

Лекция 1

Язык программирования C++

Язык C++

- Компилируемый статически типизированный язык программирования общего назначения
- Совместим с языком C
 - Но не C99
- Поддержка разных парадигм
 - Процедурное, объектно-ориентированное, обобщенное, функциональное, метапрограммирование

Программа Hello, World!

```
/*  
    Программа, выводящая строку "Hello, world!" в стандартный поток вывода  
*/  
  
#include <iostream>  
  
void main()  
{  
    // Вывод в стандартный поток вывода  
    std::cout << "Hello, world!" << std::endl;  
}
```

Константы

Константы

- Числовые константы
 - Целые числа и числа с плавающей запятой
- Логические константы
 - true и false
- Символьные константы
- Строковые константы
- [Документация](#)

Числовые константы

- Десятичные
 - 12345, -34021
 - 999999**L**, 99983**UL**
- Шестнадцатеричные
 - **0x**FeedBeef, **0x**328aadb
- Восьмеричные
 - **003**, **0723**
- Вещественные
 - 1.35, 8.45f
 - 2e+10f, -3.835e-6L

Логические константы

- Логическая константа **true** служит для обозначения логического значения «Истина», а константа **false** – значения «Ложь»

Пример – определение чётности числа

```
#include <iostream>

int main()
{
    int n;
    std::cout << "Enter a number:";
    std::cin >> n;

    bool isEven = false;
    if (n % 2 == 0)
        isEven = true;

    if (isEven)
        std::cout << n << " is an even number\n";
    else
        std::cout << n << " is an odd number\n";

    return 0;
}
```


СИМВОЛЬНЫЕ КОНСТАНТЫ

- Записывается в виде символа, обрамленного одиночными кавычками
 - 'A', '1'
- Значение символьной константы – **числовой код** символа из набора символов на данной машине
- Некоторые символы записываются в виде эскапе-последовательностей, начинающихся с символа \ul>- '\\', '\\0', '\\n', '\\177', '\\xff'

Строковые константы (строковые литералы)

- Ноль или более символов, заключенных в двойные кавычки
 - `"Hello, world\n"`
 - `""`
 - `"Hello " "world\n"` эквивалентно `"Hello world\n"`
- Во внутреннем представлении строковая константа – массив символов, завершающийся символом с кодом 0 (`'\0'`)
- Есть возможность объявления **«сырых строковых литералов»** (raw string literals), не требующих использования escape-последовательностей

```

#include <iostream>
#include <string>

void main()
{
    char letterA = 'A';
    char eol = '\n';

    // Символы, вроде " и \ внутри строковых литералов необходимо экранировать
    std::string filePath = "c:\\path\\to\\file.txt";

    // Либо использовать raw string literals
    std::string filePath1 = R"(c:\path\to\file.txt)";

    // Можно сцеплять несколько строковых литералов в один
    std::string multiLineString =
        "<html>\n"
        "\t<body>\n"
        "\t\t<p style=\"color:red;\"></p>"
        "\t</body>"
        "</html>";

    // При помощи raw string literal можно упростить задание строк, содержащих спецсимволы
    std::string htmlPage = R"marker(<html>
    <body>
        <p style="color:red;">Hello, world</p>
    </body>
</html>)marker";
}

```

Содержимое строки между "<идентификатор>(" и "<идентификатор>" воспринимается без преобразований

Что выведет программа?

```
#include <iostream>
#include <string>

int main()
{
    std::string x = "Apple" "\n" "Dog" ", " "Banana\n";

    // Что выведет программа?
    std::cout << x << "----";

    return 0;
}
```

```
Apple
Dog, Banana
---
```

Представление строковой константы в памяти

H e l l o , W o r l d \0

72	101	108	108	111	44	32	87	111	114	108	100	0
----	-----	-----	-----	-----	----	----	----	-----	-----	-----	-----	---

Типы данных

Типы данных языка C++

- Целые числа различных размеров со знаком или без
 - int, short, char
- Числа с плавающей запятой различной размерности
 - float, double, long double
- Логический тип
 - bool
- Перечисляемые типы (enum)
- Структуры (struct)
- Объединения (union)
- Массивы

Базовые типы данных

- Типы данных целых чисел
 - char
 - int
 - модификаторы
 - short/long
 - unsigned/signed
- Логический тип
 - bool
- Типы данных вещественных чисел
 - float
 - double

Объявления переменных

- Переменные объявляются раньше их использования
 - `int lower, upper, step;`
`char c, line[1000];`
`bool success;`
- При объявлении переменные могут быть инициализированы
 - `char esc = '\\';`
`int i = 0;`
`int limit = MAXLINE + 1;`
`float eps = 1.0e-5f;`
`bool success = true;`
- Модификатор **const** указывает, что значение переменной не будет далее изменяться
 - `const double e = 2.71828182845905;`
`const char msg[] = "предупреждение: ";`
`int strlen(const char str[]);`

Объявление локальных переменных и констант

```
void main()
{
    // Объявление переменной carSpeed типа double
    double carSpeed;
    carSpeed = 45.8;

    // Объявление переменной можно совместить с ее инициализацией
    int userAge = 20;
    float x = 12.6f;

    // Объявление константной переменной - переменной,
    // значение которой не может быть изменено после инициализации
    // Константа при объявлении всегда должна быть проинициализирована
    const double SPEED_OF_LIGHT = 299792458.0;
    const int SECONDS_IN_HOUR = 3600;
    const int HOURS_IN_DAY = 24;

    // Константа может быть также проинициализирована в результате выражения
    const int SECONDS_IN_DAY = SECONDS_IN_HOUR * HOURS_IN_DAY;
}
```

Автоматическое определение типа переменной

```
void main()
{
    // double
    auto PI = 3.14159265;

    // const float
    const auto E = 2.71828f;

    // float
    auto e2 = E * 2;

    // const double
    const auto halfPI = PI / 2;

    // long double
    auto sqrtPi = sqrt(PI);
}
```

Область видимости переменной

```
void main()
{
    // Область видимости переменной ограничена блоком, внутри которого она объявлена
    std::string userName = "Ivan Petrov";
    int age = 10;

    {
        // Переменная из внутреннего блока может иметь имя, совпадающее с именем из внешнего
        // блока
        // При этом внутри этого блока она замещает собой одноименную переменную из внешнего блока
        std::string userName = "Sergey Ivanov";
        assert(userName == "Sergey Ivanov");

        // Тип переменной из вложенного блока может быть другим
        double age = 7.7;

        // Лучше избегать объявления переменных, имя которых совпадает с именем из внешнего блока
    }
    // При возврате во внешний блок видимой снова становится внешняя переменная
    assert(userName == "Ivan Petrov");
}
```

Объявление глобальных переменных

```
#pragma once
```

```
// Объявление глобальной переменной как внешней  
// Это позволяет ссылаться на нее из файлов,  
// отличных от того, где она фактически определена  
extern int someOtherGlobalVariableDeclaredInVariablesCpp;
```

variables.h

variables.cpp

```
#include "variables.h"
```

```
// Глобальная переменная. Из других файлов к ней можно обратиться объявив ее внешней (extern)  
int globalVariableDeclaredInVariablesCpp = 12345;
```

```
// Еще одна глобальная переменная. В файле variable.h она объявлена как extern  
int someOtherGlobalVariableDeclaredInVariablesCpp = 54321;
```

```
// Статическая глобальная переменная. Ее область видимости - текущий .cpp файл  
// В разных .cpp файлах одной и той же программы могут быть объявлены разные  
// статические глобальные переменные. Они будут полностью изолированы друг от друга  
static int staticVariable = 66;
```

Использование глобальных переменных

main.cpp

```
#include "variables.h"

// Переменная, объявленная вне функции является глобальной. // Ее область видимости - вся программа
int someGlobalVariable = 38;
// По умолчанию глобальные переменные инициализируются нулями
int someZeroInitializedGlobleVariable;

void main()
{
    assert(someZeroInitializedGlobleVariable == 0);

    // Локальная переменная замещает собой одноименные глобальные переменные
    int someGlobalVariable = 22;
    assert(someGlobalVariable == 22);
    // К глобальной переменной все же можно обратиться по ее полному имени
    assert(::someGlobalVariable == 38);

    // Глобальные переменные, объявленные в других cpp-файлах
    // Переменная globalVariableDeclaredInVariablesCpp объявлена в файле variables.cpp
    // Чтобы обратиться к ней из других файлов, нужно предварительно объявить
    // ее внешней при помощи ключевого слова extern
    extern int globalVariableDeclaredInVariablesCpp;
    assert(globalVariableDeclaredInVariablesCpp == 12345);

    // А эта переменная была объявлена внешней в подключенном нами заголовочном файле variables.h
    assert(someOtherGlobalVariableDeclaredInVariablesCpp == 54321);
}
```

Ключевое слово `typedef`

- Язык Си++ предоставляет оператор **`typedef`**, позволяющий давать типам данных новые имена
 - После этого новое имя типа может использоваться в качестве **синонима** оригинала
- Причины использования **`typedef`**
 - Решение проблемы переносимости
 - На разных платформах/компиляторах один и тот же тип может иметь различный размер
 - Желание сделать текст программы более

Пример использования оператора typedef

```
typedef int Length;  
Length len, maxlen;  
len = 1;  
  
typedef double real;  
  
typedef int int32;  
typedef short int16;  
typedef char int8;  
  
int32 counter = 0;  
  
real x = 0.3;
```


Using – альтернатива typedef

```
int main()  
{  
    using Coordinate = double;  
  
    Coordinate x0 = 35;  
  
    return 0;  
}
```

Целочисленные типы данных

- Служат для хранения целых чисел различного размера
 - char
 - short (short int)
 - int
 - long (long int)
- Целые числа могут быть как со знаком, так и без него
 - signed
 - unsigned
- Гарантируется следующее соотношение размеров целочисленных типов:
 - `sizeof(char) <= sizeof(short)`
 - `sizeof(short) <= sizeof(int)`
 - `sizeof(int) <= sizeof(long)`

Знаковые и беззнаковые целые числа

- Типы `int` и `short` (без модификатора) являются знаковыми
 - `int` = signed `int`
 - `short` = signed `short`
- Тип `char`, **как правило**, тоже знаковый
 - `char` = signed `char`
 - Это поведение может изменяться при помощи настроек некоторых компиляторов

Представление целых чисел в памяти компьютера

- Тип `char` занимает одну ячейку памяти (байт) размером, как правило, 8 бит
 - Возможны системы, в которых разрядность байта не равна 8 битам
- Типы `short` и `int`, занимают размер, кратный размеру типа `char`
 - Размер типа `short` \leq Размер типа `int`
 - При этом число записывается в позиционной системе счисления с основанием $2^{\text{разрядность байта}}$
 - Порядок записи байтов, представляющих число в памяти, зависит от архитектуры системы
 - Little-endian, big-endian, middle-endian

Пример представления числа 666 в виде типа short и int

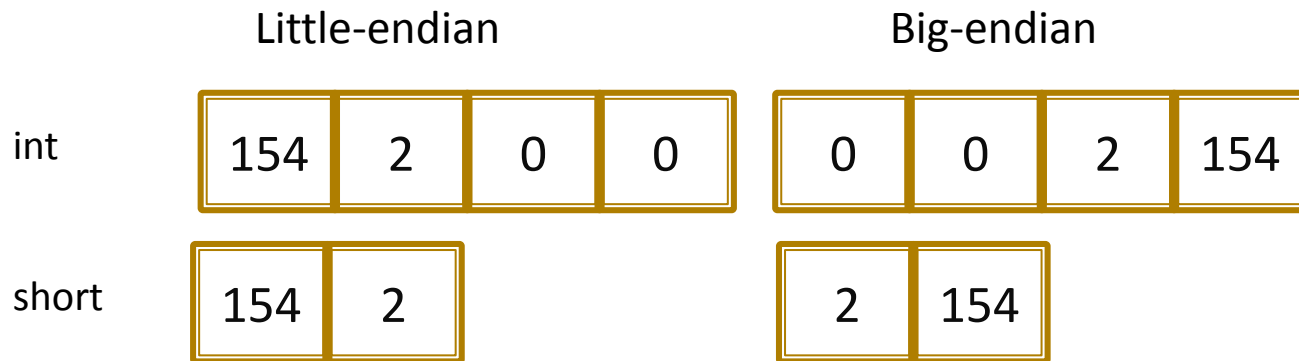
Дано:

разрядность типа char = 8 бит

разрядность типа short = 16 бит

разрядность типа int = 32 бита

$$666_{256} = 2 * 256 + 154 * 1$$



Типы данных с плавающей запятой

- Позволяют задавать вещественные числа различного размера и точности
 - float
 - double
 - long double
- Гарантированы следующие соотношения размеров вещественных типов данных
 - `sizeof(float) <= sizeof(double)`
 - `sizeof(double) <= sizeof(long double)`

Пример использования вещественных чисел

```
const float PI = 3.1415927f;

double sin60 = 0.86602540378443864676372317075294;

double FahrengeitToCelsius(double fahr)
{
    return (fahr - 32) * 5.0 / 9.0;
}

float DegreesToRadian(float degrees)
{
    return degrees * PI / 180.0f;
}
```

Перечислимый тип данных

Перечисляемые типы данных (перечисления)

- Позволяет задать ограниченный набор именованных целочисленных значений
 - День недели
 - Состояние конечного автомата
 - Модель компьютера и т.д
- Особенности
 - Имена в различных перечислениях должны отличаться друг от друга
 - Значения внутри одного перечисления могут совпадать:
 - `enum Status {Ok, Failure, Success = Ok};`

Пример использования перечислимых типов

```
#include <iostream>

enum WeekDay
{
    SUNDAY = 0,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
};

int main()
{
    WeekDay d = SUNDAY;
    std::cout << "Today is " << d << "\n";

    d++;
    std::cout << "Tomorrow will be " << d << "\n";
    return 0;
}
```

```
Today is 0
Tomorrow will be 1
```

Scoped enum

- Появились в C++11
- Ограничивают область видимости значений перечислимого типа именем перечисления
- Позволяют преодолеть ограничение традиционного enum-а на уникальность значений

Проблема традиционного enum-а

```
enum TrafficLightColor
{
    Red, Yellow, Green
};
```

// Не скомпилируется, т.к. значение Red уже используется TrafficLightColor

```
enum CarColor
{
    Black, Red, White
};
```

// Вот так скомпилируется

```
enum CarColor
{
    RedCarColor, BlackCarColor, WhiteCarColor
};
```

Решение со Scoped enum

```
enum class TrafficLightColor // или enum struct TrafficLightColor
{
    Yellow, Red, Green
};

enum class CarColor
{
    Red, Black, White
};

void main()
{
    TrafficLightColor color1 = TrafficLightColor::Red;
    CarColor color2 = CarColor::Red;
}
```

Пример использования ЛОГИЧЕСКОГО ТИПА ДАННЫХ

```
double CalculateCircleRadius(double area)
{
    bool argumentIsValid = (area >= 0);
    if (argumentIsValid)
    {
        return sqrt(area / 3.14159265);
    }
    else
    {
        return -1;
    }
}
```

Набор используемых СИМВОЛОВ

- Используются почти все графические символы ASCII таблицы (кроме @ и \$)
- Язык является чувствительным к регистру символов
 - Для записи операторов используются строчные буквы
 - Для записи идентификаторов – цифры, заглавные и строчные буквы и символ подчеркивания
 - Идентификатор не может начинаться с цифры

Основные операторы языка Си

- Общие
 - Арифметические операторы и оператор присваивания
 - Логические операторы и операторы сравнения
 - Оператор sizeof
- Управление ходом выполнения программы
 - Условные операторы
 - Операторы циклов
 - Оператор множественного выбора
- Операторы для работы с массивами, структурами и объединениями
- Операторы для работы с указателями

Арифметические операторы

■ Бинарные

- +
- -
- *
- /
- % (остаток от деления – применяется только к целым)
 - `int i = 10 % 3; /* i = 1; */`
- Деление целых сопровождается отбрасыванием дробной части
 - `float f = 8 / 3; /* f = 2.0 */`

■ Унарные (ставятся перед операндом)

- +
 - `int i = +1;`
- -
 - `int j = -8;`

Пример

```
int main()
{
    /*
    Вася купил 10 килограммов яблок,
    а бананов - на 5 килограммов больше, чем яблок.
    Груш он купил столько же, сколько яблок и бананов вместе.

    Во сколько раз больше масса всех фруктов, чем яблок?
    */
    double apples = 10;
    double bananas = apples + 5;
    double pears = apples + bananas;

    double totalFruit = apples + bananas + pears;

    double ratio = totalFruit / apples;
    std::cout << "Total fruit to apples ratio is " << ratio << "\n";
}
```

Операторы отношения

- Операторы отношения

- >
- >=
- <
- <=

- Операторы сравнения на равенство

- ==
- !=

- Логические операторы

- && - логическое **И**

- `char ch = getchar();`
`int isDigit = (ch >= '0') && (ch <= '9');`

- || - логическое **ИЛИ**

- `char ch = getchar();`
`if ((ch == ' ') || (ch == '\n') || (ch == '\t'))`
`printf("Разделитель");`

- ! – логическое **НЕ**

- `if (!valid)` эквивалентно `if (valid == 0)`

- Вычисления операторов **&&** и **||** прекращаются как только станет известна истинность или ложность результата

Пример: нахождением максимума из 3-х чисел

```
int main()
{
    /*
    Пользователь вводит 3 числа: a, b и c
    Найти вывести максимальное из этих чисел
    */
    std::cout << "Enter 3 integer numbers: ";

    int a, b, c;
    std::cin >> a >> b >> c;

    int maximum = a;
    if (b > maximum)
        maximum = b;
    if (c > maximum)
        maximum = c;

    std::cout << "Maximum is " << maximum << "\n";
}
```

Пример: определение ВИСОКОСНОСТИ ГОДА

```
// Определить, является ли високосным введённый пользователем год
int main()
{
    std::cout << "Enter year: ";

    int year;
    std::cin >> year;

    bool isLeap = (year % 4 == 0) &&
        ((year % 100 != 0) || (year % 400 == 0));
    if (isLeap)
        std::cout << year << " is a leap year\n";
    else
        std::cout << year << " is not a leap year\n";
}
```

Операторы инкремента и декремента

- Увеличивают или уменьшают значение операнда на 1
 - ++
 - --
- Имеют две формы
 - Префиксная форма (возвращает новое значение аргумента)
 - `int i = 0;`
`int j = ++i; /* i = 1; j = 1; */`
 - Постфиксная форма (возвращает старое значение аргумента)
 - `int i = 0;`
`int j = i--; /* i = -1; j = 0; */`
- Операторы инкремента и декремента можно применять только к переменным
 - `int i = (j + y)++; /* ошибка */`

```
int main()
{
    /*
    Пользователь вводит натуральное число.
    Вывести количество единиц в десятичной записи этого числа
    */
    std::cout << "Enter a natural number: ";
    int number;
    std::cin >> number;

    int n = number;
    int ones = 0;
    do
    {
        int units = n % 10;
        if (units == 1)
            ++ones;
        n /= 10;
    } while (n != 0);
    std::cout << "Decimal notation of " << number
        << " contains " << ones << " 1s\n";
}
```

Побитовые операторы

- Данные операторы позволяют осуществлять операции над отдельными битами целочисленных операндов
 - **&** - побитовое **И**
 - `int i = 0xde & 0xf0; /* i = 0xd0 */`
 - **|** - побитовое **ИЛИ**
 - `int i = 0xf0 | 0x03; /* i = 0xf3 */`
 - **^** - побитовое **исключающее ИЛИ**
 - `int i = 0x03 ^ 0x02; /* i = 0x01 */`
 - **<<** - **сдвиг влево**
 - `int i = 1 << 3; /* i = 8 */`
 - **>>** - **сдвиг вправо**
 - `int i = 0xd0 >> 4; /* i = 0x0d */`
 - **~** - побитовое **отрицание** (унарный оператор).
 - `char i = ~0x1; /* i = 0xfe (0xfe = 11111110b) */`

Пример: функция getbits

```
/* getbits: получает n бит, начиная с p-й позиции */  
unsigned getbits(unsigned x, int p, int n)  
{  
    return (x >> (p+1-n)) & ~(~0 << n);  
}
```

										n = 7									
1	1	0	0	1	0	1	0	1	0	1	1	0	1	1	1				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
						p													

$(x \gg (9 + 1 - 7)) =$

0	0	0	1	1	0	0	1	0	1	0	1	0	1	1	0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

$\sim 0 =$

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

$\sim 0 \ll 7 =$

1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

$\sim(\sim 0 \ll 7) =$

0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

$(x \gg (9 + 1 - 7)) \& \sim(\sim 0 \ll 7) =$

0	0	0	1	1	0	0	1	0	1	0	1	0	1	1	0			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
&	&	&	&	&	&	&	&	&	&	&	&	&	&	&				
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1			
=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=			
0	0	0	0	0	0	0	0	0	1	0	1	0	1	1	0			

Операторы и выражения присваивания

- Служат для присваивания переменным значения некоторого выражения
 - `i = 3;`
 - `i += 8;`
 - `i <<= 1;`
 - `j %= 3;`
- Типом и значением выражения присваивания является тип и значение левого операнда после завершения присваивания
 - ```
while ((c = getchar()) != EOF)
{
 // do something
}
```

# Пример: функция `bitcount`

```
/* bitcount: подсчет количества единичных битов в числе x */
int bitcount(unsigned x)
{
 int b;
 for (b = 0; x != 0; x >>= 1)
 {
 if (x & 0x01)
 ++b;
 }
 return b;
}
```

# Преобразование типов в стиле C

- Происходит, когда операнды оператора принадлежат к разным типам
- Неявное преобразование
  - `int i = 7.0 + 3 - 2.0f;`
- Явное преобразование
  - `int i = (int) (7.0 + 3 - 2.0f);`
- Если один из аргументов является знаковым целым, а второй беззнаковым, результатом будет целое число **без знака**

# Опасность неявного приведения типов

```
int CenterPictureOnTheScreen(
 int pictureWidth, unsigned screenWidth)
{
 return (screenWidth - pictureWidth) / 2;
}

int main(int argc, char * argv)
{
 unsigned screenWidth = 100;

 unsigned pic1Width = 50;
 int pic1X = CenterPictureOnTheScreen(pic1Width, screenWidth);
 // pic1x = 25: ok

 unsigned pic2Width = 150;
 int pic2X = CenterPictureOnTheScreen(pic2Width, screenWidth);
 // pic2x = 2147483623: error
 return 0;
}
```

# Решение проблемы – явное приведение типов

```
int CenterPictureOnTheScreen(
 int pictureWidth, unsigned screenWidth)
{
 return ((int)screenWidth - pictureWidth) / 2;
}

int main(int argc, char * argv)
{
 unsigned screenWidth = 100;

 unsigned pic1Width = 50;
 int pic1X = CenterPictureOnTheScreen(pic1Width, screenWidth);
 // pic1x = 25: ok

 unsigned pic2Width = 150;
 int pic2X = CenterPictureOnTheScreen(pic2Width, screenWidth);
 // pic2x = -25:ok
 return 0;
}
```

# Недостатки оператора преобразования типов в стиле C

- Несмотря на свою простоту данный способ преобразования типов обладает рядом недостатков
  - Допускаются потенциально некорректные преобразования типов, зачастую без информирования разработчика
  - Сложно найти в тексте программы

# Пример

```
void Test(double doubleValue)
{
 int intValue = (int)&doubleValue;
 ...
}
```

```
struct Point
{
 double x;
 double y;
};
```

```
void Test1(const Point * p)
{
 /* программист отвлекся и вместо
 int x = (int)p->x;
 написал: */
 int x = (int)p;
}
```



# Преобразование типов в стиле C++

- В языке C++ введены 4 оператора приведения типов
  - `static_cast<Type>(arg)`
  - `dynamic_cast<Type>(arg)`
  - `const_cast<Type>(arg)`
  - `reinterpret_cast<Type>(arg)`
- Каждый из данных операторов применяется для определенного преобразования типов в конкретной ситуации
- В программах на C++ следует отдавать предпочтение данным операторам

# Оператор `static_cast`

- Применяется для статического преобразования одного типа к другому
- Также может применяться для статического преобразования типов указателей в пределах иерархии классов

# Пример

```
void Test(double doubleValue)
{
 // Ошибка КОМПИЛЯЦИИ
 int intValue = static_cast<int>(&doubleValue);
}

struct Point
{
 double x;
 double y;
};

void Test1(const Point * p)
{
 int x = static_cast<int>(p->x); // ok
 int y = static_cast<int>(p); // ошибка КОМПИЛЯЦИИ
}
```

# Оператор `dynamic_cast`

- Применяется для динамического преобразования типов в пределах иерархии классов
  - (об этом позже)

# Оператор `const_cast`

- Применяется для снятия константности с константного выражения

```
int k = 0;
const int * pConstK = &k;
int * pK = const_cast<int*>(pConstK);

const int & constRefK = k;
int & refK = const_cast<int&>(constRefK);
```

# Оператор reinterpret\_cast

- Может применяться для преобразования между целочисленными типами и указателями, а также между указателями на

**несвязанные друг с другом типы данных**

```
int main(int argc, char * argv)
{
 float x = 30.3;
 unsigned y = *reinterpret_cast<int*>(&x);

 // берем четвертый бит двоичной записи числа 30.3
 unsigned z = y & (1 << 4);

 return 0;
}
```

# Условное выражение

- Условное выражение имеет вид:  
выр1 ? выр2 : выр3
  - Сначала вычисляется выражение 1
  - Если оно истинно (не равно нулю), то вычисляется выражение 2 и его значение становится значением всего условного выражения
  - В противном случае вычисляется выражение 3 и становится значением всего условного выражения
- Пример
  - $z = (a > b) ? a : b; /* z = \max(a, b) */$





# Управление выполнением программы

# Инструкции и блоки

- Выражение (например,  $x = 0$ ) становится инструкцией, если в конце поставить точку с запятой
  - $x = 0$ ;
  - `printf("Hello");`
  - В Си точка с запятой является заключающим символом инструкции, а не разделителем, как в языке Паскаль.
- Фигурные скобки `{` и `}` используются для объединения объявлений и инструкций в **составную инструкцию**, или **блок**
  - с т.з. синтаксиса языка блок воспринимается как одна инструкция

# Блоки и область видимости

- Переменные видимы внутри того блока, где она объявлена
- При покидании своего блока видимости переменная уничтожается, а занимаемая ею область памяти – освобождается
  - (автоматическое управление памятью)

```
int main(int argc, char * argv)
{
 int a = 0;
 if (argc > 1)
 {
 int b = argc - 1;
 }
 return 0;
}
```

# Конструкция if-else

- Оператор `if` позволяет выполнить тот или иной участок кода в зависимости от значения некоторого выражения
  - `if (<выражение>)`  
    `<инстр.1>`  
    `else`  
    `<инстр.2>`
  - `if (<выражение>)`  
    `<инстр>`

# Конструкция else-if

- Позволяет осуществлять многоступенчатое решение

- `if (выражение)`  
    *инструкция*  
`else if (выражение)`  
    *инструкция*  
`else if (выражение)`  
    *инструкция*  
`else if (выражение)`  
    *инструкция*  
`else`  
    *инструкция*

# Пример, бинарный поиск

```
/* binsearch: найти x в v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, const int v[], int n)
{
 int low, high, mid;

 low = 0;
 high = n - 1;
 while (low <= high)
 {
 mid = (low + high) / 2;
 if (x < v[mid])
 high = mid - 1;
 else if (x > v[mid])
 low = mid + 1;
 else /* совпадение найдено */
 return mid;
 }
 return -1; /* совпадения нет */
}
```

# Оператор switch

- Используется для выбора одного из нескольких путей
  - Осуществляется проверка на совпадение значения выражения с одной из некоторого набора целых констант, и выполняет соответствующую ветвь программы
- Инструкция `break` выполняет **выход из блока switch**

```
switch (выражение)
{
 case конст-выр: инструкции
 case конст-выр: инструкции
 default: инструкции
}
```

```
#include <string>
#include <iostream>
#include <cassert>

enum class WeekDay
{
 Sunday, Monday, Tuesday, Wednesday,
 Thursday, Friday, Saturday
};

std::string WeekDayToString(const WeekDay & weekDay)
{
 switch (weekDay)
 {
 case WeekDay::Sunday: return "Sunday";
 case WeekDay::Monday: return "Monday";
 case WeekDay::Tuesday: return "Tuesday";
 case WeekDay::Wednesday: return "Wednesday";
 case WeekDay::Thursday: return "Thursday";
 case WeekDay::Friday: return "Friday";
 case WeekDay::Saturday: return "Saturday";
 default:
 assert(!"This is not possible");
 return "";
 }
}

void main()
{
 std::cout << WeekDayToString(WeekDay::Sunday) << std::endl;
}
```



```
#include <iostream>
```

```
int main() /* подсчет цифр, символов-разделителей и прочих символов */
{
```

```
 int numSpaces = 0;
```

```
 int numDigits[10] = {};
```

```
 int numOther = 0;
```

```
 char ch;
```

```
 while (std::cin.get(ch))
```

```
 {
```

```
 switch (ch)
```

```
 {
```

```
 case '0': case '1': case '2': case '3': case '4':
```

```
 case '5': case '6': case '7': case '8': case '9':
```

```
 ++numDigits[ch - '0'];
```

```
 break;
```

```
 case ' ': case '\n': case '\t':
```

```
 ++numSpaces;
```

```
 break;
```

```
 default:
```

```
 ++numOther;
```

```
 }
```

```
 }
```

```
 std::cout << "Digits:";
```

```
 for (int n : numDigits)
```

```
 {
```

```
 std::cout << " " << n;
```

```
 }
```

```
 std::cout << ", whitespaces: " << numSpaces << ", other: " << numOther << std::endl;
```

```
 return 0;
```

```
}
```

# Циклическое выполнение

# Что такое циклическое выполнение

- **Цикл** – последовательность из нескольких операторов, указываемая в программе один раз, которая выполняется несколько раз подряд
  - Допускается существование **бесконечного цикла**
- **Тело цикла** - последовательность операторов, предназначенная для многократного выполнения в цикле

# Циклическое выполнение в языке Си

- Циклическое выполнение в языке Си осуществляется при использовании следующих операторов цикла:
  - `while`
  - `for`
  - `do..while`
- Внутри циклов могут использоваться операторы управления работой цикла:
  - **`break`** для досрочного выхода из цикла
  - **`continue`** для пропуска текущей итерации

# Оператор `while`

- Оператор **`while`** служит для организации **циклов с предусловием**
  - цикл, который выполняется, пока истинно некоторое условие, указанное перед его началом
- Синтаксис
  - `while (выражение)`  
*инструкция*
  - *Инструкция* (тело цикла) выполняется до тех пор, пока ***выражение*** принимает ненулевое значение

# Пример: нахождение наибольшего общего делителя

```
// Поиск наибольшего общего делителя чисел a и b
{
 unsigned a = 714;
 unsigned b = 312;
 cout << "Greatest Common Denominator of " << a << " and " << b << " is ";
 while (b != 0)
 {
 swap(a, b);
 b = b % a;
 }
 cout << max(a, 1u) << endl;
}
```

# Оператор for

- Оператор **for** служит для организации **ЦИКЛОВ СО СЧЕТЧИКОМ**
- Синтаксис
  - `for (выр1; выр2; выр3)`  
*инструкция*
  - **Выражение1** выполняется один раз перед началом цикла
    - Например, оператор инициализации счетчика цикла
  - Выполнение **инструкции** (тело цикла) продолжается до тех пор, пока **выражение2** имеет ненулевое значение
    - если **выражение2** отсутствует, то выполнение цикла продолжается **бесконечно**
  - После каждой итерации цикла выполняется **выражение3**
    - Например, изменение счетчика цикла

# Простой цикл for

```
void main()
{
 // Выводит 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
 // Область видимости переменной i ограничена телом цикла
 for (int i = 0; i < 10; ++i)
 {
 cout << i << ", ";
 }
 cout << endl;

 for (int i = 100; i >= 0; i -= 2)
 {
 cout << i << ", ";
 }
 cout << endl;
}
```



# Range-based for

- Версия цикла for, предназначенная для перебора элементов некоторого диапазона
  - Массивы, строки, контейнеры стандартной библиотеки, пользовательские типы данных
- Синтаксис:
  - for (*тип идентификатор* : диапазон)  
*инструкция*

# Пример: обход элементов массива

```
// Обход элементов массива
{
 int numbers[] = { 10, 15, 17, 33, 18 };
 int sum = 0;
 int product = 1;
 cout << "Array items: ";
 for (int number : numbers)
 {
 cout << number << ", ";
 sum += number;
 product *= number;
 }
 cout << endl << "\tSum: " << sum << endl << "\tProduct: " << product << endl;
}
```

```

// Обход символов строки и контейнера map
{
 string pangram = "the quick brown fox jumps over the lazy dog";

 map<char, int> characterOccurrences;
 for (char ch : pangram)
 {
 cout << ch;
 if (isalpha(ch))
 {
 // Возвращенное значение частоты встречаемости символа увеличиваем на 1
 ++characterOccurrences[ch];
 }
 }

 cout << "Character occurrence in \"" << pangram << "\":" << endl;
 for (const auto & charOcc : characterOccurrences)
 {
 cout << "\t" << charOcc.first << ": " << charOcc.second << endl;
 }

 /*
 Выше написанный цикл аналогичес следующему:
 for (auto it = characterOccurrences.begin(); it != characterOccurrences.end(); ++it)
 {
 const auto & charOcc = *it;
 cout << "\t" << charOcc.first << ": " << charOcc.second << endl;
 }
 */
}

```

# Оператор do-while

- Оператор **do-while** служит для организации **циклов с постусловием**
  - цикл, в котором условие проверяется **после** выполнения тела цикла
  - тело **всегда выполняется** хотя бы один раз
- Синтаксис
  - `do`  
*инструкция*  
`while (выражение) ;`
  - *Инструкция* выполняется до тех пор, пока *выражение* принимает **ненулевое** значение

# Пример

```
// Ввод продолжается, пока пользователь не введет bye
{
 string userInput;
 do
 {
 cout << R"(Enter text or "bye" to finish:)";
 getline(cin, userInput);
 cout << "You entered: " << userInput << endl;
 } while (userInput != "bye");
}
```

# Бесконечные циклы for, while, do-while

```
{
 // Генератор псевдослучайных чисел с использованием Вихря Мерсенна (Mersenne Twister)
 mt19937 generator;
 // Адаптер для получения равномерно распределенных чисел в диапазоне [1; 10]
 uniform_int_distribution<int> dist(1, 10);
 for (;;)
 {
 cout << "Next random number: " << dist(generator) << endl;
 cout << "Type q or Q to quit the game: ";
 string userInput;
 getline(cin, userInput);
 if (userInput == "q" || userInput == "Q")
 {
 break;
 }
 }
}

// Также можно использовать цикл while:
// while (true)
// {
// тело цикла
// }
// либо цикл do-while:
// do
// {
// тело цикла
// } while(true);
}
```

# Вложенные циклы

- Один цикл может быть вложен в другой
  - При этом выполнение внутреннего цикла выполняется как часть оператора внешнего цикла

# Инструкции **break** и **continue**

- Инструкция **break** осуществляет немедленный выход из тела цикла, внутри которого она находится
  - Также инструкция **break** осуществляет выход из оператора **switch**
- Инструкция **continue** осуществляет пропуск оставшихся операторов тела цикла, внутри которого она находится, и переход на следующую итерацию цикла
  - В циклах **while** и **do-while** осуществляется переход к проверке условия
  - В цикле **for** осуществляется переход к приращению переменной цикла



# Пример: поиск простых чисел

```
cout << "Primes: ";
for (int i = 2; i < 100; ++i)
{
 bool isPrime = true;
 // Наивный метод определения простоты числа i
 // проверяем i на делимость на любое из чисел диапазона [2; sqrt(i)]
 for (int j = 2; j * j <= i; ++j)
 {
 // Если найден множитель числа i, выходим из цикла при помощи break
 if (i % j == 0)
 {
 isPrime = false;
 break;
 }
 }
 if (isPrime)
 {
 cout << i << ", ";
 }
}
```

# Инструкция goto

- Инструкция **goto** позволяет осуществить переход на заданную метку внутри текущей функции
- Синтаксис:
  - `goto метка;`
- Как правило, использование инструкции goto усложняет структуру программы и без крайней необходимости ею пользоваться не стоит
  - Если Вы все еще думаете об использовании этого оператора – **использовать его все**

# Пример

```
/* поиск совпадающих элементов в массивах */
for (i = 0; i < n; ++i)
{
 for (j = 0; j < m; ++j)
 {
 if (a[i] == b[j])
 goto found;
 }
}
/* нет одинаковых элементов */
...
found:
/* обнаружено совпадение: a[i] == b[j] */
...
```

# Структуры

---

# Структуры

- **Структура** - это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем.
  - Структуры помогают в организации сложных данных, позволяя группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое

```
// Структура Point, задающая точку на плоскости
```

```
struct Point
```

```
{
```

```
 int x;
```

```
 int y;
```

```
};
```

```
// Поля глобально объявленной структуры по умолчанию инициализируются нулями
```

```
Point globalPoint;
```

```
void main()
{
 // Объявляем переменную pt, а затем инициализируем ее поля одно за другим
 Point pt;
 pt.x = 10;
 pt.y = 20;

 // Объявление переменной-структуры можно совместить
 // с инициализацией ее полей
 Point pt0 = { 33, 24 };
 assert(pt0.x == 33 && pt0.y == 24);
 // Еще один способ инициализации структуры при ее объявлении
 Point pt1{ 14, -22 };
 assert(pt1.x == 14 && pt1.y == -22);

 // Недостающие поля при инициализации заполняются нулями
 Point pt2 = { 21 };
 assert(pt2.x == 21 && pt2.y == 0);
 Point pt3 = {};
 assert(pt3.x == 0 && pt3.y == 0);

 // Поля глобальных и статических переменных-структур по умолчанию
 // инициализируются нулями
 static Point pt4;
 assert(pt4.x == 0 && pt4.y == 0);
 assert(globalPoint.x == 0 && globalPoint.y == 0);
}
```

```
struct Triangle
{
 Point vertex1;
 Point vertex2;
 Point vertex3;
};

// Инициализация структур, содержащих вложенные структуры
void main()
{
 Triangle t1 =
 {
 {0, 0},
 {20, 100},
 {30, 15}
 };
 assert(t1.vertex1.x == 0 && t1.vertex1.y == 0);
 assert(t1.vertex2.x == 20 && t1.vertex2.y == 100);
 assert(t1.vertex3.x == 30 && t1.vertex3.y == 15);

 // Структура, все поля которой будут проинициализированы нулями
 Triangle t2 = {};
 assert(t2.vertex1.x == 0 && t2.vertex1.y == 0);
 assert(t2.vertex2.x == 0 && t2.vertex2.y == 0);
 assert(t2.vertex3.x == 0 && t2.vertex3.y == 0);
}
```



```
// Структуры в качестве параметров функций и возвращаемых значений
double CalculateDistance(const Point & pt1, const Point & pt2)
{
 return hypot(pt1.x - pt2.x, pt1.y - pt2.y);
}
Point CalculateTriangleCenter(const Triangle & triangle)
{
 return {
 (triangle.vertex1.x + triangle.vertex2.x + triangle.vertex3.x) / 3,
 (triangle.vertex1.y + triangle.vertex2.y + triangle.vertex3.y) / 3,
 };
}

void main()
{
 Triangle t0 = {
 { 0, 0 }, { 10, -20 }, { 20, 20 }
 };
 auto center = CalculateTriangleCenter(t0);
 assert(center.x == 10 && center.y == 0);

 // При передаче в функцию можно создать экземпляр структуры без объявления переменной
 // В этом случае в функцию будет передана ссылка временный объект
 center = CalculateTriangleCenter({ { 0, 0 }, { -20, 10 }, { 20, 20 } });
 assert(center.x == 0 && center.y == 10);

 Point pt0{ 1, 1 };
 Point pt1{ 4, 5 };
 double distance = CalculateDistance(pt0, pt1);
 // Проверка чисел с плавающей запятой на приблизительное равенство
 assert(abs(distance - 5.0) <= DBL_EPSILON);
}
```

```
enum class Month
{
 January, February, March,
 April, May, June,
 July, August, September,
 October, November, December
};
```

```
struct Date
{
 int day;
 Month month;
 int year;
};
```

// Person - пример более сложной структуры

```
struct Person
{
 std::string name;
 std::string address;
 Date birthday;
 int height;
};
```

```
// Проверка двух дат на равенство
bool Equals(const Date & d1, const Date& d2)
{
 return (d1.day == d2.day) && (d1.month == d2.month) && (d1.year == d2.year);
}
// Проверка двух людей на идентичность
bool Equals(const Person & p1, const Person & p2)
{
 return (p1.name == p2.name) && (p1.address == p2.address) &&
 Equals(p1.birthday, p2.birthday) && p1.height == p2.height;
}
void main()
{
 Person person1 = {
 "Ivanov Ivan", "Suvorova Street, 17",
 { 10, Month::March, 1975 }, 185
 };
 Person person2 = {
 "Sergeev Egor", "Sovetskaya Street, 24",
 { 11, Month::February, 1990 }, 116
 };
 Person person3 = {
 "Ivanov Ivan", "Suvorova Street, 17",
 { 10, Month::March, 1975 }, 185
 };
 assert(!Equals(person1, person2));
 assert(!Equals(person2, person3));
 assert(Equals(person1, person3));
}
```

# Объединения

---

# Объединения

- **Объединение** - это тип данных, который может содержать (в разные моменты времени) объекты различных типов и размеров
  - Объединения позволяют хранить разнородные данные в одной и той же области памяти без включения в программу машинно-зависимой информации

```
#include <iostream>

enum class NumericType
{
 INTEGER,
 REAL,
};

struct Numeric
{
 NumericType type;

 union
 {
 int intValue;
 double realValue;
 }value;
};

void PrintNumeric(const Numeric & n)
{
 if (n.type == NumericType::INTEGER)
 std::cout << n.value.intValue << std::endl;
 else
 std::cout << n.value.realValue << std::endl;
}

void main()
{
 Numeric a, b;

 a.type = NumericType::INTEGER; a.value.intValue = 5;
 b.type = NumericType::REAL; b.value.realValue = 3.8;

 PrintNumeric(a);
 PrintNumeric(b);
}
```

|     |
|-----|
| 5   |
| 3.8 |

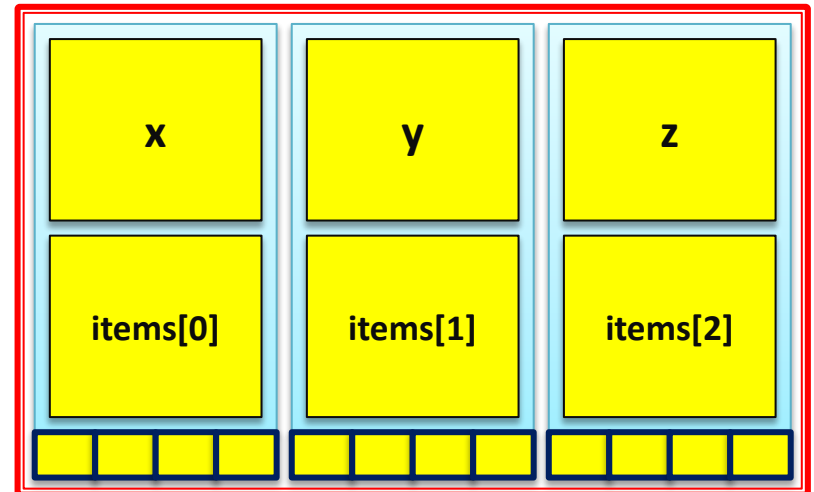
# Пример 2

```
union Vector3D
{
 struct
 {
 float x, y, z;
 };
 float items[3];
};
...
```

```
Vector3D v1;
v1.x = 0;
v1.y = 10;
v1.z = -0.3;
```

```
v1.items[0] = 3;
```

Vector3D



# Массивы

---



# Массивы

- Массивы позволяют объявить несколько (один и более) последовательных объектов, объединенных под одним именем, и осуществлять к ним индексированный доступ
  - В качестве индексов используются целые числа, или типы, приводимые к целым
  - Размер массива задается статически на этапе компиляции и не может быть изменен в ходе работы программы
  - Индекс начального элемента массива равен **нулю**
  - Есть возможность объявления многомерных массивов

```
#include <cassert>
#include <string>

int g_globalArray[3];
void main()
{
 // Глобальные переменные-массивы по умолчанию инициализируются нулем
 assert(g_globalArray[0] == 0 && g_globalArray[1] == 0 && g_globalArray[2] == 0);

 // Массив из 3-х элементов. Элементы не проинициализированы
 float floatNumbers[3];
 floatNumbers[0] = 1.0; floatNumbers[1] = 3.5; floatNumbers[2] = -4.5;

 // Массив при объявлении может быть проинициализирован
 double doubleNumbers[3] = { 3.8, 2.1, 3.53 };

 // Элементы массива, не указанные при инициализации, равны нулю
 double zeroFilledArray[3] = { 3.5, 7.2 };
 assert(zeroFilledArray[2] == 0.0);

 // Элементы проинициализированы нулями
 double zeroInitializedArray[3] = { };

 // Если не указать размер массива при инициализации,
 // он будет определен автоматически
 double arrayOf5Items[] = { 3.5, 8.7, 2.3, -1.25, 0.0 };

 std::string name = "John", surname = "Doe";
 // При инициализации элементов массив могут также использоваться выражения
 std::string userNames[] = { "Ivan", "Sergey", name + " " + surname };
 // Так можно определить количество элементов в массиве
 assert(std::end(userNames) - std::begin(userNames) == 3);
 assert(userNames[2] == "John Doe");
}
```

# Массивы символов

```
void ArrayOfChars()
{
 // Константный массив из 5 элементов
 const char name[] = { 'J', 'o', 'h', 'n', '\0' };

 // Неконстантный массив из 4 элементов
 char surname[] = "Doe";

 // Константный массив из 6 элементов
 const char hello[6] = "Hello";
}
```

# Определение размера массива

```
#include <cassert>
#include <string>

int g_globalArray[3];

// При помощи такой функции можно определить количество элементов в массиве
template<typename T, std::size_t N>
constexpr std::size_t SizeOfArray(T(&)[N])
{
 return N;
}

void main()
{
 std::string userNames[] = { "Ivan", "Sergey", "Stepan" };
 assert(std::end(userNames) - std::begin(userNames) == 3);
 assert(SizeOfArray(userNames) == 3);

 const char arr1[] = { 'J', 'o', 'h', 'n', '\0' };
 assert(std::end(arr1) - std::begin(arr1) == 5);

 char arr2[] = "Doe";
 assert(SizeOfArray(arr2) == 4);

 const char arr3[6] = "Hello";
 assert(SizeOfArray(arr3) == 6);
}
```

# Многомерные массивы

```
#include <cassert>

typedef double Matrix2x2[2][2];

void main()
{
 Matrix2x2 mat = {
 {1.0, 2.5},
 {4.5, 3.2}
 };
 assert(mat[0][0] == 1.0);
 assert(mat[0][1] == 2.5);
 assert(mat[1][0] == 4.5);
 assert(mat[1][1] == 3.2);
}
```

# Передача массива в функцию

```
#include <cassert>

typedef double Matrix2x2[2][2];

void Fn(Matrix2x2 mat)
{
 mat[0][0] = 3.0;
}

struct WrappedMatrix2x2
{
 Matrix2x2 items;
};

void Fn2(WrappedMatrix2x2 mat)
{
 mat.items[0][0] = 3.0;
}

void main()
{
 Matrix2x2 mat = {
 {1.0, 2.5},
 {4.5, 3.2}
 };
 assert(mat[0][0] == 1.0);

 // При передаче массива в функцию Fn будет передан оригинал
 Fn(mat);
 // Модификация элементов массива внутри Fn изменит переданный
 массив
 assert(mat[0][0] == 3.0);

 WrappedMatrix2x2 wrappedMat = {
 {
 { 1.0, 2.5 },
 { 4.5, 3.2 }
 }
 };
 // При передаче структуры в функцию Fn2
 // будет передана копия структуры
 Fn2(wrappedMat);
 // Модификация элементов массива внутри Fn2
 // на оригинальный массив влияния не окажет
 assert(wrappedMat.items[0][0] == 1.0);
}
```

# Указатели, динамическая память

# Указатели

- Указатель – используются для хранения адресов переменных в памяти
- Основные области применения
  - Работа с динамической памятью
  - Работа с массивами
  - Передача параметров в функцию по ссылке
  - Организация связанных структур данных (списки, деревья)



```

#include <stdio.h>

typedef struct tagPoint
{
 int x, y;
}Point;

void PrintPoint(Point *pPoint)
{
 printf("point is (%d, %d)\n", pPoint->x, (*pPoint).y);
}

void Swap(int *a, int *b)
{
 int temp = *a;
 *a = *b;
 *b = temp;
}

int main()
{
 int value = 0;
 int one = 1, two = 2;
 int *pValue = &value;
 Point pnt = {10, 20};

 printf("value is %d\n", value);
 *pValue = 1;
 printf("now value is %d\n\n", value);

 printf("one=%d, two=%d\n", one, two);
 Swap(&one, &two);
 printf("now one=%d, two=%d\n\n", one, two);

 PrintPoint(&pnt);

 return 0;
}

```

```

value is 0
now value is 1

one=1, two=2
now one=2, two=1

point is (10, 20)

```

# Хранение данных

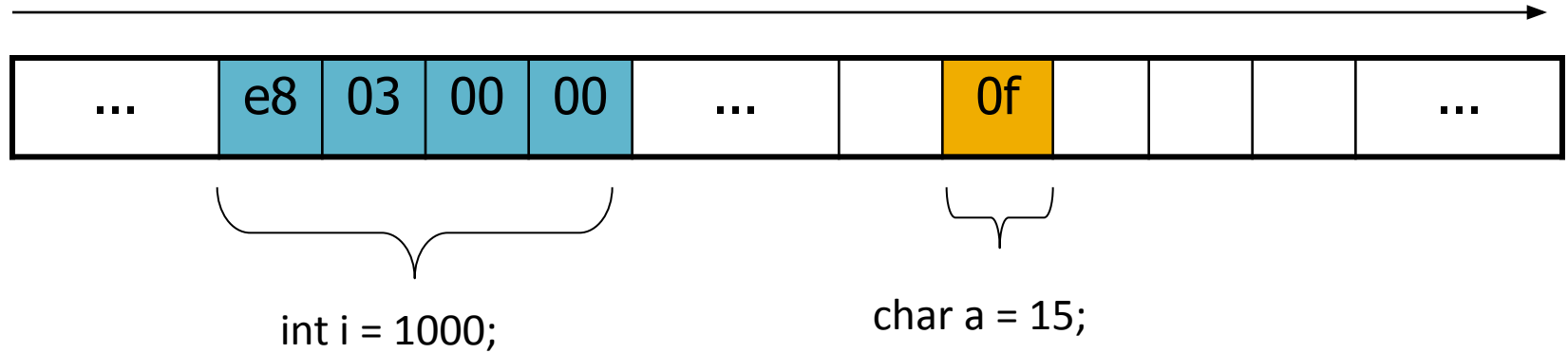
- В C++ есть три разных способа выделения памяти для объектов
  - **Статическое**: пространство для объектов создаётся в области хранения данных программы в момент компиляции;
  - **Автоматическое**: объекты можно временно хранить в [стеке](#); эта память затем автоматически освобождается и может быть использована снова, после того, как программа выходит из блока, использующего её.
  - **Динамическое**: блоки памяти нужного размера могут запрашиваться во время выполнения программы с помощью оператора `new` в области памяти, называемой [кучей](#). Эти блоки освобождаются и могут быть использованы снова после вызова для них оператора `delete`.

# Организация памяти в языке

## C++

- С точки зрения языка C++ память представляет собой массив последовательно пронумерованных ячеек памяти, с которыми можно работать по отдельности или связными кусками
  - Порядковый номер ячейки называется ее **адресом**
  - Эта память используется для хранения значений переменных.
  - Переменные различных типов могут занимать различное количество ячеек памяти, и иметь различные способы представления в памяти

# Пример



# Что такое указатель?

- **Указатель** – это переменная, которая может хранить адрес другой переменной в памяти заданного типа
  - Указатели – мощное средство языка C++, позволяющее эффективно решать различные задачи
  - Использование указателей открывает доступ к памяти машины, поэтому пользоваться ими следует аккуратно

# Объявление указателя

- Указатель на переменную определенного типа объявляется следующим образом:

**<тип> \* <идентификатор>;**

- Например:  
`int *pointerToInt;`
- Указатель, способный хранить адрес переменной любого типа имеет тип **void\***:
  - `void * pointerToAnyType;`
- Как и к обычным переменным, к указателям можно применять модификатор **const**:
  - `const int * pointerToConstInt;`
  - `char * const constPointerToChar = &ch;`
  - `const double * const constPointerToConstDouble = &x;`
  - `float * const constPointerToFloat = &y;`
  - `const void * pointerToConstData;`

# Получение адреса переменной

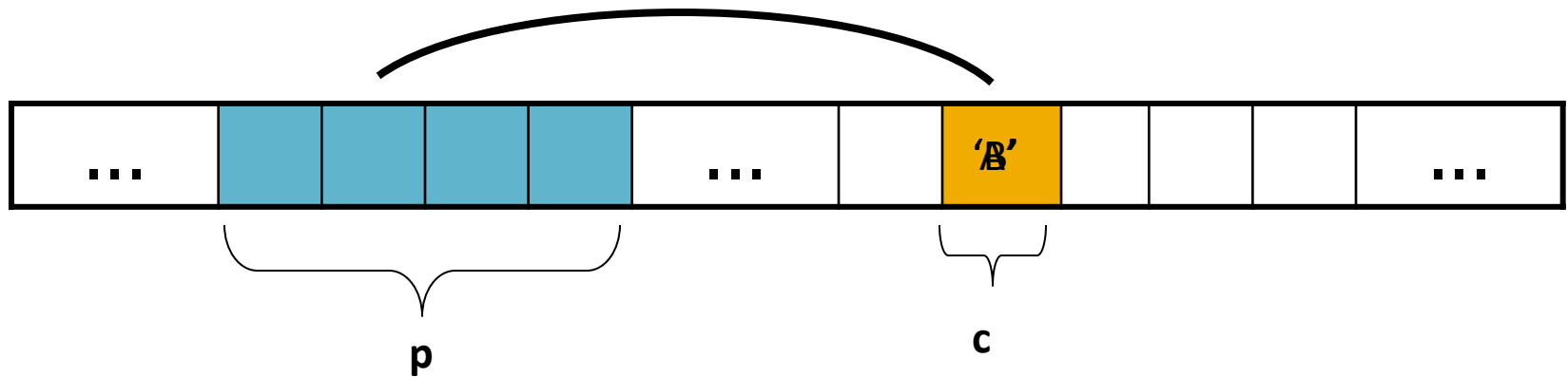
- Для взятия адреса переменной в памяти служит унарный оператор **&**
  - Этот оператор возвращает адрес переменной, который может быть присвоен указателю совместимого типа
  - Оператор взятия адреса применим только к переменным. Его нельзя применять к числовым константам, литералам, выражениям или регистровым переменным

# Оператор косвенного доступа

- Для доступа к значению, на которое ссылается указатель, необходимо его **разыменование** (dereferencing), осуществляемое при помощи **унарного оператора \***

- ```
int * p = &i;  
*p = 5;
```


Пример



```
char c = 'A' ;
```

```
char *p = &c ;
```

```
*p = 'B' ;
```

Инициализация указателей

- Значение **неинициализированного** указателя не определено
 - **Разыменование такого указателя приводит к неопределенному поведению**
 - Лучше присвоить указателю нулевое значение (или символическую константу **NULL**), чтобы подчеркнуть, что он не ссылается ни на какую переменную:
 - `char * p1 = 0;`
`char * p2 = NULL;`
- В стандарт C++11 введено специальное ключевое слово `nullptr`, обозначающее нулевой указатель
- Разыменование нулевого указателя также приводит к неопределенному поведению, однако появляется возможность проверки значения указателя:
 - `if (p != nullptr) // или просто if (p)`

NULL (или 0) vs nullptr

- В программах на C++11 следует использовать `nullptr` вместо `NULL` или `0`
 - У `nullptr` отсутствует неявное преобразование к целочисленным типам
 - При этом сохраняется неявное преобразование к типу `bool` (значение `false`)

```
#include <iostream>

void Print(void * p)
{
    std::cout << "Printing a pointer: " << p << "\n";
}

void Print(int i)
{
    std::cout << "Printing an integer: " << i << "\n";
}

void Print(bool b)
{
    std::cout << "Printing a boolean: " << (b ? "true" : "false") <<
"\n";
}

int main(int argc, char* argv[])
{
    Print(NULL); //
    Printing an integer: 0
    Print(nullptr); // Printing a
    pointer: 00000000
    bool nullptrAsBool = nullptr;
```

Копирование указателей

- Как и в случае обычных переменных, значение одного указателя может быть присвоено другому при помощи оператора =
 - Следует помнить, что в этом случае **копируется адрес переменной, а не ее значение**
 - Для копирования значения переменной, на которую ссылается указатель, необходимо применить **оператор разыменования ***
 - ```
char a = 'A';
char b = 'B';
char c = 'C';
char *pa = &a;
char *pb = &b;
char *pc = &c;
pa = pb; // pa и pb теперь хранят адрес b
*pa = *pc; // b теперь хранит значение 'C'
```

# Указатели и аргументы функций

- В языке Си параметры в функцию передаются по значению.
  - Указатели – единственный способ изменить значение параметра изнутри функции
  - В языке С++ появилась возможность передачи параметров по ссылке

```
void swap(int *pa, int
*pb)
{
 int tmp = *pa;
 *pa = *pb;
 *pb = tmp;
}
```

```
void swap(int &pa, int
&pb)
{
 int tmp = pa;
 pa = pb;
 pb = tmp;
}
```

# Указатели на функции

- В Си можно объявить указатель на функцию и работать с ним как с обычной переменной, сохраняя возможность вызова функции по указателю на нее
  - Данная возможность позволяет иметь несколько реализаций алгоритма, имеющих общий интерфейс
  - Для получения указателя следует воспользоваться оператором взятия адреса &
    - В C++ можно использовать имя функции
- В C++ есть возможность использовать функциональные объекты
- В C++11 появились безымянные функции (lambda-функции)
  - У Lambda-функций, не имеющих состояния, есть возможность получения адреса

```
#include <stdio.h>
typedef bool (*OrderedFunction)(int a, int b);

void BubbleSort(int array[], int size, OrderedFunction fn)
{
 bool sorted;
 do
 {
 sorted = true;
 for (int i = 0; i < size - 1; ++i)
 {
 if (!fn(array[i], array[i + 1]))
 {
 int tmp = array[i];
 array[i] = array[i + 1];
 array[i + 1] = tmp;
 sorted = false;
 }
 }
 --size;
 } while(!sorted && (size > 1));
}

bool IsOrdered(int a, int b) {return a <= b; }

int main()
{
 int arr[5] = {3, 5, 1, 7, 9};
 BubbleSort(arr, 5, IsOrdered);
 return 0;
}
```



```

#include <stdio.h>
typedef bool (*OrderedFunction)(int a, int b);

void BubbleSort(int array[], int size, OrderedFunction fn)
{
 bool sorted;
 do
 {
 sorted = true;
 for (int i = 0; i < size - 1; ++i)
 {
 if (!fn(array[i], array[i + 1]))
 {
 int tmp = array[i];
 array[i] = array[i + 1];
 array[i + 1] = tmp;
 sorted = false;
 }
 }
 --size;
 } while(!sorted && (size > 1));
}

void main()
{
 int arr[5] = {3, 5, 1, 7, 9};
 BubbleSort(arr, 5, [](int a, int b){return a <= b;});
}

```

```
#include <functional>
#include <utility>

typedef std::function<bool(int a, int b)> OrderedFunction;

void BubbleSort(int array[], int size, OrderedFunction const& isOrdered)
{
 bool sorted;
 do
 {
 sorted = true;
 for (int i = 0; i < size - 1; ++i)
 {
 if (!isOrdered(array[i], array[i + 1]))
 {
 std::swap(array[i], array[i + 1]);
 sorted = false;
 }
 }
 --size;
 } while (!sorted && (size > 1));
}

void main()
{
 int arr[5] = { 3, 5, 1, 7, 9 };
 BubbleSort(arr, 5, [](int a, int b){return a <= b; });
}
```

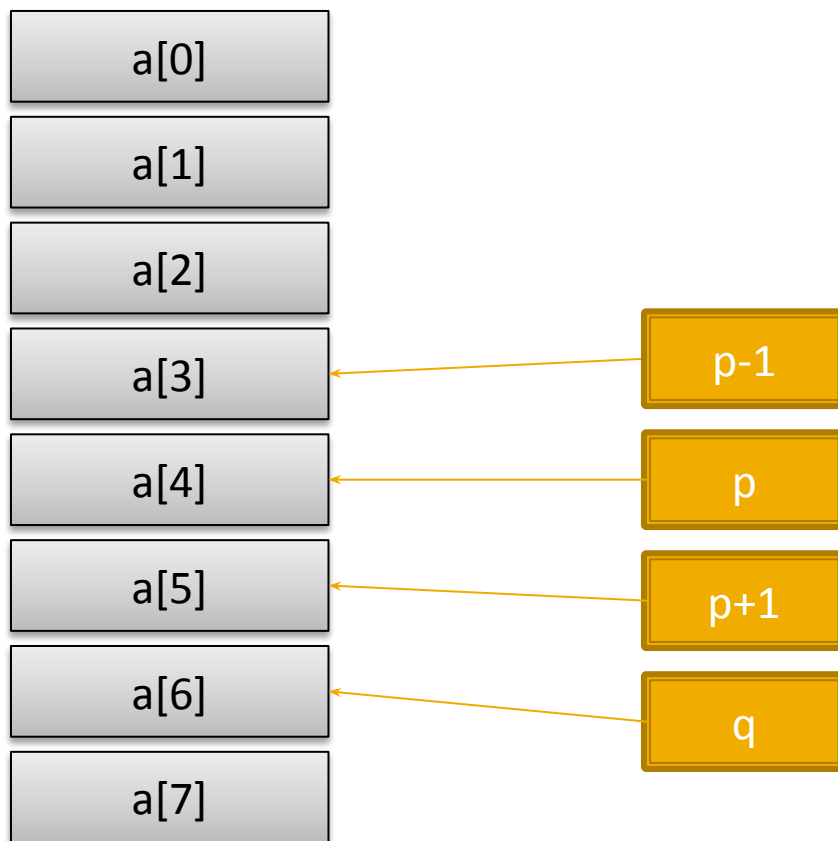
# Указатели и массивы

- Указатели и массивы в C/C++ тесно связаны
  - Имя массива является синонимом расположения его начального элемента
    - `int arr[10];`  
`int *p = arr; // эквивалентно int *p = &arr[0];`
  - Индексация элементов массива возможна с помощью указателей и адресной арифметики

# Адресная арифметика

- Если  $p$  – указатель на некоторый элемент массива, то
  - $p+1$  – указатель на следующий элемент
  - $p-1$  – указатель на предыдущий элемент
  - $p+j$  – указатель на  $j$ -й элемент после  $p$
  - $p[j]$  разыменовывает  $j$ -й элемент относительно  $p$
- Если  $p$  и  $q$  – указатели на некоторые элементы **одного массива**, то
  - $p-q$  - равно количеству элементов после  $q$ , которое необходимо добавить, чтобы получить  $p$
  - $p < q$  принимает значение 1, если  $p$  указывает на элемент, предшествующий  $q$ , в противном случае - 0
  - $p == q$ , принимает значение 1 если  $p$  и  $q$  указывают на один и тот же элемент, в противном случае - 0

# Адресная арифметика в действии



$p = \&a[4]$

$q = \&a[6]$

$q - 2 \Rightarrow p$

$q - p \Rightarrow 2$

$p + 2 \Rightarrow q$

$p < q \Rightarrow \text{true}$

$p + 1 == q - 1$

$p[3] \Rightarrow a[7]$

$p[-2] \Rightarrow a[2]$

$\&a[8] - \&a[0] \Rightarrow 8$

# Примеры

```
int arr[10];

// получаем указатель на начальный элемент
массива
int *p = arr; // эквивалентно int *p = &arr[0];

// следующие две строки эквивалентны
*(p + 4) = 5;
arr[4] = 5;

/* несмотря на то, что в массиве всего 10
элементов,
допускается получать указатель на ячейку,
следующую
за последним элементом массива */
p = &a[10];
*(p - 1) = 3; // эквивалентно arr[9] = 3;
```

# Указатели на `char`

- Строковые константы – массивы символов с завершающим нулем
- Передача строковой константы в функцию (напр. `printf`) осуществляется путем передачи указателя на ее начальный элемент

# Особенности

- Присваивание символьных указателей, **не копирует строки**
  - `char * p = "Hello";`  
`char * p1 = p;` // p и p1 указывают на одно и то же место в памяти
- Символьный массив и символьный указатель – различные понятия
  - `char msg[] = "Hello";` // массив
    - Символы внутри массива могут изменяться
    - `msg` всегда указывает на одно и то же место в памяти
  - `char *pmsg = "Hello";` // указатель
    - Попытка изменить символы через `pmsg` приведет к неопределенному поведению
    - `pmsg` – указатель, можно присвоить ему другое значение в ходе работы программы



# Массивы указателей

- Указатели, как и другие переменные можно группировать в массивы
  - `int main(int argc, char* argv[])`
  - `const char * a[] = {"Hello", "World!"};  
printf("%s %s\n", a[0], a[1]);  
a[0] = "Goodbye";`
- Массивы указателей могут использоваться как альтернатива двумерных массивов

# Указатели на указатели

- В C и C++ возможны указатели, ссылающиеся на другие указатели
  - `char arr[] = "Hello";`  
`char *parr = arr;`  
`char **pparr = &parr; // pparr – хранит адрес указателя parr`  
`(*pparr)[0] = 'h'; // arr = "hello"`  
`pparr[0][1] = 'E'; // arr = "hEllo";`

# Инкремент и декремент указателя

- Когда указатель ссылается на определенный элемент массива, имеют смысл операции инкремента и декремента указателя
  - `char str[] = "Hello, world!";`  
`char *p = str;` // p указывает на символ H  
`p++;` // p указывает на символ e  
`*p = 'E';` // заменяем символ e на E

```
#include "stdio.h"
```

```
// возвращаем адрес найденного символа в строке или nullptr в случае
отсутствия
```

```
const char* FindChar(const char str[], char ch)
```

```
{
 const char * p = str;

 while (*p != '\0')
 {
 if (*p == ch)
 return p;
 ++p;
 }
```

```
 return nullptr;
}
```

```
int main()
```

```
{
 const char str[] = "Hello, world!\n";

 const char *pw = FindChar(str, 'w');
 if (pw)
 printf("%s", pw);

 return 0;
}
```

**Output:**

world!

# Указатели и динамическая память

- Часто возможны ситуации, когда размер и количество блоков памяти, необходимых программе, не известны заранее
- В этом случае прибегают к использованию динамически распределяемой памяти
  - Приложение может запрашивать блоки памяти необходимого размера из области, называемой кучей (heap)
  - Как только блок памяти становится не нужен его освобождают возвращая память

# Операторы `new` и `delete`

- В состав языка C++ вошли операторы `new` и `delete`, осуществляющие работу с динамической памятью на уровне языка
  - Оператор `new` выделяет память под элемент или массив элементов
    - Тип `*p = new Тип()`
    - Тип `*p = new Тип(инициализатор,...)`
    - Тип `*p = new Тип[кол-во элементов]`
  - Оператор `delete` освобождает память, выделенную ранее оператором `new`
    - `delete pObject;`
    - `delete [] pArray;`

# Прочие средства работы с динамической памятью

- В стандартной библиотеке языка C для работы с динамической памятью служат функции:
  - malloc
  - calloc
  - realloc
  - free
- Существуют средства работы с динамической памятью, зависящие от используемой ОС или используемых компонентов

# Функции `memcpy`, `memset` и `memmove`

- Функция `memcpy` осуществляет копирование блока памяти из одного адреса в другой
  - `void memcpy(void *dst, const void *src, size_t count)`
- Функция `memmove` аналогична `memcpy`, но корректно работает, если блоки перекрываются
  - `void memmove(void *dst, const void *src, size_t count)`
- Функция `memset` заполняет область памяти определенным значением типа `char`
  - `void memset(void *dst, int c, size_t count)`



# Пример

```
int n = 30;

// выделяем память под n элементов типа
int
int * arr = (int*)malloc(sizeof(int) * n);

memset(arr, 1, sizeof(int) * n);

arr[0] = 5;

free(arr);
arr = NULL;
```

# Указатели на структуры и объединения

- Указатели на структуры объявляются аналогично указателям на другие типы
- Для доступа к элементам структуры может применяться оператор ->

```
■ struct Point
 {
 int x, y;
 };
```

```
Point p = {10, 20};
Point *pPoint = &p;
(*pPoint).x = 1;
pPoint->y = 2;
```

# Правила корректной работы с динамической памятью

- Объекты, выделенные при помощи оператора `new` должны быть удалены при помощи оператора `delete`
  - `MyType * pObj = new MyType;`  
`delete pObj;`
- Массивы объектов, выделенные при помощи оператора `new []` должны быть удалены при помощи оператора `delete []`
  - `MyType * pArray = new MyType[N];`  
`delete [] pArray;`
- Блоки памяти, выделенные при помощи функции `malloc` (`realloc`, `calloc`) должны быть освобождены при помощи функции `free`
  - `void * p = malloc(1000);`  
`free(p);`
- Использование «непарных» средств освобождения памяти приведет к неопределенному поведению

# Проблемы ручного управления памятью

- «Висячие ссылки» (dangling pointer)
  - После удаления объекта все указатели на него становятся «висячими»
    - Область памяти может быть отдана ОС и стать недоступной, либо использоваться новым объектом
    - Разыменованное или попытка повторного удаления приведет либо к аварийной остановке программы, либо к неопределенному поведению
  - Причина возникновения: неправильная оценка времени жизни объекта – команда удаления объекта вызывается до окончания его использования в программе

# Проблемы ручного управления памятью (продолжение)

## ■ Утечка памяти (Memory Leak)

### ■ Причины:

- Программист не удалил объект после завершения использования
- Ссылающемуся на объект указателю присвоено новое значение, тогда как на объект нет других ссылок
  - Объект становится недоступен программно, но продолжает занимать память

### ■ Следствие

- Программа все больше и больше потребляет памяти, замедляя работу системы, пока не исчерпает доступный объем адресного пространства и не завершится с ошибкой

# Примеры некорректной работы с динамической памятью

```
int main(int argc, char* argv[])
{
 int * pIntArray = new int[100];
 free(pIntArray); // Неопределенное поведение: использование free вместо delete []

 int * pAnotherIntArray = new int[10];
 delete pAnotherIntArray; // Неопределенное поведение: использование delete вместо delete []

 // Выделяем в куче один объект float, инициализируя его значением 100
 float * pFloat = new float(100);
 delete [] pFloat; // Неопределенное поведение: использование delete [] вместо delete

 char * myString = new char[100];
 delete [] myString;
 delete [] myString; // Неопределенное поведение: повторное удаление массива

 char * anotherString = new char[10];
 delete [] anotherString;
 anotherString[0] = 'A'; // Неопределенное поведение: доступ к элементам удаленного массива

 void * pData = malloc(100);
 free(pData);
 free(pData); // Неопределенное поведение: повторное удаление блока данных
}
```

# Еще примеры некорректной работы с динамической памятью

```
int main(int argc, char* argv[])
{
 int * someInt = new int(11);
 someInt = new int(12); // Утечка памяти: старое значение указателя потеряно, память не освободить
 delete someInt;

 int someValue = *(new int(35)); // Утечка памяти: выделили в куче, разыменовали, адрес потеряли

 int * p = new int(10);
 if (getchar() == 'A')
 {
 return 0; // Утечка памяти: забыли вызывать delete p перед выходом из функции
 }
 delete p;

 return 0;
}
```

# Как не прострелить себе ногу, программируя на C++

- Работа с указателями – сильная сторона C++, требующая большой внимательности и ответственности
- Сведите к минимуму ручную работу с указателями и динамической памятью
  - Используйте контейнеры стандартной библиотеки C++ как альтернативу динамическим массивам
  - Используйте классы стандартных «умных указателей» для владения объектами в динамической памяти
  - Используйте иные проверенные временем библиотеки (например boost)
  - Пишите свои умные обертки, автоматизирующие владение ресурсами



# А как у них?

- Есть ЯВУ, использующие [сборку мусора](#) (Garbage collection), например Java, C#, JavaScript, D, Lisp, ActionScript, Objective C и др.
  - Освобождение памяти от неиспользуемых объектов возлагается на среду исполнения
  - Свобода программиста по работе с указателями, адресной арифметикой в таких языках либо отсутствует, либо сильно ограничена
  - Возможны кратковременные замедления в работе программы в неопределенные моменты на время сборки мусора
  - Эффективная работа сборщика мусора возможна только при достаточном количестве свободной памяти

# Автоматический сборщик мусора – не панацея

- Сборка мусора автоматизирует лишь работу с памятью, но не с другими ресурсами (файлы, подключения к БД)
- В некоторых языках есть возможность выполнить некоторый код непосредственно перед удалением объекта ([финализатор](#)) сборщиком мусора
  - Для управления ресурсами не годится, т.к. объект может использоваться (либо не удаляться сборщиком мусора) гораздо дольше, чем владеемый им ресурс, поэтому ресурсами приходится управлять вручную
- Утечки памяти все равно возможны, если ссылка на ненужный более объект хранится в используемом объекте
  - В некоторых языках есть [слабые ссылки](#)

# Ссылки

---

# Ссылки

- Ссылку можно рассматривать как еще одно имя объекта
- Синтаксис
  - `<Тип> &` означает ссылку на `<Тип>`
- Применение
  - Задание параметров функций
  - Перегрузка операций

# Ссылки в качестве параметров функций

- Функция принимает не копию аргумента, а ссылку на него
  - При сложных типах аргументов (классы, структуры) это может дать прирост в скорости вызова функции
    - Не тратится время на создании копии
    - Простые типы, как правило, эффективнее передавать по значению
      - char, int, float, double
  - Изменение значения формального параметра внутри функции приводит к изменению значения переданного аргумента
    - Альтернативный способ возврата значения из функции
    - Возврат нескольких значений одновременно

# Константные ссылки в качестве параметров функций

- Параметр, переданный в функцию по константной ссылке, доступен внутри нее только для чтения
- Если функция не изменяет значение своего аргумента, то имеет смысл передавать его по константной ссылке
  - Простые типы данных следует передавать по значению

# Пример 1

```
#include <stdio.h>

void Swap(int & a, int & b)
{
 int tmp = a;
 a = b;
 b = tmp;
}

int main()
{
 int a = 1, b = 3;
 printf("a=%d, b=%d\n", a, b);
 Swap(a, b);
 printf("a=%d, b=%d\n", a, b);
}
```

## OUTPUT

a=1, b=3

a=3, b=1

# Пример 2

```
struct Point
{
 int x, y;
};

void Print(Point const& pnt)
{
 printf("(x:%d, y:%d)\n", pnt.x, pnt.y);
}

int main()
{
 Point pnt = {10, 20};
 Print(pnt);

 return 0;
}
```



# Инициализация ссылки

- Ссылка должна быть обязательно проинициализирована
  - Должен существовать объект на который она ссылается
  - Синтаксис
    - Тип & идентификатор = значение;
- Инициализация ссылки совершенно отличается от операции присваивания
  - Будучи проинициализированной, присваивание ссылке нового значения **изменяет значение ссылаемого объекта, а не значение ссылки**

# Пример

```
#include <stdio.h>

int main()
{
 int i = 1;
 int j = 3;

 // инициализация ссылки
 int & ri = i;
 printf("i=%d, j=%d\n", i, j);

 // присваивание значения объекту, на который ссылается ri
 ri = j;
 printf("i=%d, j=%d\n", i, j);
}
```

## OUTPUT

i=1, j=3

i=3, j=3

# Ссылки на временные объекты

- При инициализации ссылки объектом другого типа компилятор создает временный объект нужного типа и использует его для инициализации ссылки
  - На данный временный объект может ссылаться только константная ссылка
  - То же самое происходит при инициализации ссылки значением константы
  - Изменение значения объекта в данном случае не отражается на значении временного объекта
- Время жизни временного объекта равно области видимости созданной ссылки

# Пример 1

```
int a = 1;
int & refA = a; // ссылка на a

printf("a = %d\n", a);
++refA;
printf("Now a = %d\n\n", a);

const double & refDoubleA = a; // ссылка на временный объект
printf("refDoubleA = %f\n", refDoubleA);

// изменение a не оказывает влияния на refDoubleA
++a;
printf("Now a = %d, refDoubleA = %f\n", a, refDoubleA);
```

## OUTPUT:

```
a = 1
```

```
Now a = 2
```

```
refDoubleA = 2.00000
```

```
Now a = 3, refDoubleA = 2.00000
```

# Пример 2

```
#include <iostream>

int Add(int x, int y)
{
 return x + y;
}

int main(int argc, char* argv[])
{
 int & wontCompile = Add(10, 20); // Ошибка компиляции

 const int & result = Add(10, 20); // ОК

 std::cout << result << "\n";
 return 0;
}
```

# Пространства имен

---

# Пространства имен

- **Пространства имен** позволяют логически сгруппировать классы, переменные и функции в некоторые именованные области
  - Позволяют избежать конфликта имен идентификаторов в различных модулях проекта
  - Разбивают программу на функциональные единицы
- Доступ к идентификатору внутри пространства имен:
  - `<namespace>::<identifier>`
  - Либо:
    - `using namespace <namespace>; <identifier>;`
  - Либо:
    - `using <namespace>::<identifier>; <identifier>;`
  - **Не рекомендуется использовать `using namespace` в заголовочных файлах**

```
#include <stdio.h>

namespace math
{
 int calculateX2(int x)
 {
 return x * x;
 }
}

namespace graphics
{
 namespace shapes
 {
 struct rectangle
 {
 int x, y, w, h;
 };
 struct circle
 {
 int x, y, r;
 };
 }
}

namespace sound_player
{
 void PlaySound()
 {
 // sound playing code is placed here
 }
}

using namespace sound_player;

int main()
{
 int x = 5;
 int x2 = math::calculateX2(x);
 graphics::shapes::rectangle rect = {0, 0, 40, 30};
 PlaySound();

 using graphics::shapes::rectangle;
 rectangle r1;
 return 0;
}
```



# Стандартная библиотека шаблонов STL

# Стандартная библиотека шаблонов (STL)

- Программная библиотека, содержащая большое количество готового к использованию обобщенного кода
  - Контейнеры
  - Итераторы
  - Алгоритмы
  - Умные указатели
  - Поддержка многопоточности, случайных чисел
  - Потоки ввода/вывода
  - Поддержка функционального программирования
  - И многое другое
- Все контейнеры, алгоритмы и итераторы в STL объявлены в пространстве имен `std`
  - Стандарт запрещает программисту объявлять свои типы в данном пространстве имен

# Контейнеры

- Классы, предназначенные для хранения элементов определенного типа
  - STL содержит классы обобщенных реализаций различных контейнеров, которые можно использовать с элементами различных типов
- В STL поддерживаются 2 вида контейнеров
  - Последовательные
  - Ассоциативные

# Основные контейнеры STL

- Последовательные контейнеры
  - Строка (**basic\_string**, **string**, **wstring**)
  - Вектор (**vector**)
  - Двусвязный список (**list**)
  - Двусторонняя очередь (**deque**)
- Ассоциативные контейнеры
  - Отображение (**map**, **multimap**, **unordered\_map**, **unordered\_multimap**)
  - Множество (**set**, **multiset**, **unordered\_set**, **unordered\_multiset**)
- Контейнеры-адаптеры
  - Стек (**stack**)
  - Очередь (**queue**)
  - Очередь с приоритетом (**priority\_queue**)

# Строка std::string

- Контейнер, предназначенный для хранения строк произвольной длины
  - В качестве элементов строк могут выступать элементы типа `char` (`string`), `wchar_t` (`wstring`) или определяемые пользователем типы (`basic_string`)
  - Достоинства:
    - Автоматизация управления памятью
    - Набор операций для работы со строками
  - Для работы с данным классом строк необходимо подключить заголовочный файл `<string>`

# Создание строки

```
string emptyString;
string hello = "Hello";
auto goodbye = "Goodbye"s;

const char chars[] = {'0', 'n', 'e'};
// Создание строки из массива символов заданной длины
string one(chars, std::size(chars)); // One

string aaaa(4, 'a'); // aaaa
```

# Размер и вместимость

```
auto text = "This is a very long string"s;
assert(text.Length() == 26);
assert(text.size() == text.Length());
assert(text.capacity() >= text.Length());
```

```
auto oldCapacity = text.capacity();
text.erase(19, 7); // erase " string"
assert(text == "This is a very long"s);
assert(text.capacity() == oldCapacity);
assert(text.Length() == 19);
```

```
assert(!text.empty());
text.clear();
assert(text.Length() == 0);
assert(text.empty());
assert(text.capacity() == oldCapacity);
```

# Сравнение строк

```
assert("bbb"s > "aaa"s);
assert("xyz"s == "xyz"s);
assert("Abc"s > "Abb"s);
```

```
string s = "Hello";
assert("Hello" == s);
```



# Конкатенация строк

```
string hello("Hello");
string world("world");
```

```
string helloWorld = hello + " " + world;
// "Hello world"
```

```
string s;
s.append(hello).append(" ").append(world);
// "Hello world"
```

# Извлечение подстроки

```
auto helloWorld = "Hello world"s;

assert(helloWorld.substr(0, 5) == "Hello"s);
assert(helloWorld.substr(6, 5) == "world"s);
assert(helloWorld.substr(6) == "world"s);
assert(helloWorld.substr() == "Hello world"s);
```

```
auto s = "This is a wonderful "s;
s.append(helloWorld, 0, 4);
assert(s == "This is a wonderful Hell");
```

```
auto helloWorld1 = helloWorld.substr(0, 6);
helloWorld1.append(helloWorld, 6);
assert(helloWorld1 == helloWorld);
```

# Поиск внутри строки

```
string s("Hello world");

assert(s.find('w') == 6);
assert(s.find('x') == string::npos);
assert(s.find_first_of("aeiouy"s) == 1); // e

assert(s.find("world"s) == 6);
assert(s.find('o') == 4); // first 'o' letter
assert(s.rfind('o') == 7); // last 'o' letter

assert(s.find('o', 5) == 7);
```

# Замена внутри строки

```
auto s = "Hello world"s;
s.replace(0, 5, "Goodbye"s);
assert(s == "Goodbye world"s);
```

```
auto s1 = "This is a cat"s;
s1.replace(10, 3, s, 8, 5);
assert(s1 == "This is a world");
```

**string\_view**

---

# string view

- Объект, ссылающийся на неизменяемую последовательность символов в памяти
- Не владеет символьными данными
  - При разрушении `string_view` массив не удаляется
  - После разрушения массива символов использовать ссылавшийся на него `string_view` нельзя
- Легковесный
  - Как правило, указатель на начало + длина

# Конструирование string\_view

```
const char arr[] = { 'H', 'e', 'l', 'l', 'o' };
string_view v(arr, 5);
assert(v.length() == 5);
assert(v.data() == &arr[0]); // view uses arr's data

string s("Hello");
string_view sv = s;
assert(sv.data() == s.data());
assert(sv.length() == s.length());

sv = string_view(s.data(), 4);
assert(sv.length() == 4);
```

# Пример

```
auto url =
 "http://en.cppreference.com/w/cpp/string/basic_string_view"s;

string_view scheme(&url[0], 4);
assert(scheme == "http");

string_view domain(&url[7], 19);
assert(domain == "en.cppreference.com");
```



# Вектор std::vector

- Контейнер для хранения динамического массива элементов произвольного типа
  - Автоматизация процесса управления памятью
    - Везде, где возможно, рекомендуется использовать класс `vector` как альтернативу динамическому выделению массивов объектов при помощи оператора `new []`
    - К элементам массива предоставляется индексированный доступ
  - Для использования данного класса необходимо подключить заголовочный файл `<vector>`

# Пример

```
#include <vector>
#include <string>

using namespace std;
int main(int argc, char *argv[])
{
 // создаем массив целых чисел, содержащий 100 элементов
 vector<int> vectorOfInt(100);
 vector<string> vectorOfString;

 vectorOfInt.push_back(10);
 vectorOfString.push_back("Hello");

 std::string hello = vectorOfString[0];
 size_t numberOfItems = vectorOfString.size();

 return 0;
}
```

```
#include <vector>
#include <iostream>
#include <string>
using namespace std;

struct Point
{
 int x, y;
};

struct PointDb1
{
 double x, y;
};

void main()
{
 vector<Point> points = {
 {0, 0}, {20, 10}, {30, 30}
 };

 for (auto & pt : points)
 {
 cout << pt.x << ", " << pt.y << endl;
 }

 PointDb1 center = {0, 0};
 for (auto & pt : points)
 {
 center.x += pt.x;
 center.y += pt.y;
 }
 center.x /= points.size();
 center.y /= points.size();

 cout << center.x << ", " << center.y << endl;
}
```

# Двусвязный список `std::list`

- Реализовывает двусвязный список элементов произвольного типа
  - К элементам списка осуществляется последовательный доступ при помощи итераторов
  - Вставка и удаление элементов из произвольного места списка осуществляется за **постоянное время**
  - Для начала работы с данным контейнером необходимо подключить заголовочный файл `<list>`

# Пример

```
#include <list>
#include <string>
#include <iostream>

using namespace std;
int main(int argc, char *argv[])
{
 list<string> listOfStrings;
 listOfStrings.push_back("One");
 listOfStrings.push_back("Two");
 listOfStrings.push_back("Three");

 for (auto & item : listOfStrings)
 {
 cout << item << ", ";
 }
 return 0;
}
```

```
#include <list>
#include <string>
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;
typedef list<string> StringList;

StringList PopulateNamesList()
{
 StringList maleNames; maleNames.emplace_back("Ivan"); maleNames.emplace_front("Sergey");
 StringList femaleNames; femaleNames.emplace_back("Irina"); femaleNames.emplace_front("Anna");

 StringList allNames(move(maleNames));
 allNames.insert(allNames.end(), femaleNames.cbegin(), femaleNames.cend());
 return allNames;
}

void PrintNamesList(StringList const& names)
{
 for (auto const& name : names)
 {
 cout << name << ", ";
 }
 cout << endl;
}

void PrintNamesListByCopyingItemsToCout(StringList const& names)
{
 copy(names.begin(), names.end(), ostream_iterator<string>(cout, ", "));
 cout << endl;
}

void main()
{
 StringList names = PopulateNamesList();
 PrintNamesList(names);
 PrintNamesListByCopyingItemsToCout(names);
}
```

# Двусторонняя очередь (double-ended queue) std::deque

- Аналогична вектору, но обеспечивает эффективную вставку и удаление элементов не только в конце, но и в начале очереди
  - В отличие от вектора **не гарантируется** последовательное хранение элементов в памяти
  - Гарантируется константный доступ к элементу по его индексу
  - Для использования необходимо подключить заголовочный файл <deque>

# Классы std::map и std::multimap

- Ассоциативный контейнер, хранящий пары «ключ» - «значение»
  - Позволяет отображать элементы одного типа в элементы другого или того же самого типа
  - map – все ключи уникальные
  - multimap – допускается дублирование ключей
- Для подключения данных классов необходимо подключить заголовочный файл <map>
- Требования к ключам:
  - Наличие операции отношения <



# Пример

```
#include <map>
#include <string>
#include <iostream>

using namespace std;
int main(int argc, char *argv[])
{
 map<string, string> dictionary;
 dictionary.insert(pair<string, string>("Cat", "Кошка"));
 dictionary.insert(make_pair("Snake", "Змея"));
 dictionary["Dog"] = "Собака";
 dictionary ["Mouse"] = "МЪШЪ";

 cout << dictionary["Dog"] << "\n";
 return 0;
}
```

# Пример – подсчет частоты встречаемости символов

```
#include <map>
#include <unordered_map>
#include <string>
#include <iostream>

using namespace std;

void main()
{
 map<char, int> charFreq;
 //Можно использовать unordered_map
 //unordered_map<char, int> charFreq;
 string text = "a quick brown fox jumps over the lazy dog";
 for (auto ch : text)
 {
 if (isalpha(ch))
 {
 ++charFreq[ch];
 }
 }

 for (auto & chInfo : charFreq)
 {
 cout << chInfo.first << ": " << chInfo.second << endl;
 }

 cout << endl;
}
```

# Классы std::unordered\_map и std::unordered\_multimap

- Ассоциативный контейнер, хранящий пары «ключ» - «значение»
  - Элементы хранятся не отсортированы никоим образом, но сгруппированы в bucket-ы согласно хеш-значению ключей
  - `unordered_map` – все ключи уникальные
  - `unordered_multimap` – допускается дублирование ключей
- Для подключения данных классов необходимо подключить заголовочный файл `<unordered_map>`
- Требования к ключам:
  - Наличие операции сравнения `==`
  - Возможность вычислить хеш от значения ключа

```
#include <unordered_map>
#include <string>
#include <iostream>

using namespace std;

enum class Gender { Male, Female, };

string GenderToString(Gender gender)
{
 return (gender == Gender::Male) ? "male" : "female";
}

void main()
{
 unordered_map<string, Gender> nameToGender = {
 {"Sarah", Gender::Female},
 {"John", Gender::Male},
 {"Leonardo", Gender::Male},
 {"Richard", Gender::Male},
 {"Tanya", Gender::Female}
 };
 cout << "Tanya is a "
 << GenderToString(nameToGender.at("Tanya")) << " name.\n";
 cout << "John is a " << GenderToString(nameToGender.at("John")) << " name.\n";
}
```

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

struct Point
{
 int x;
 int y;
};

// Структура с перегруженным оператором (), позволяющая вычислить
// хеш-значение для заданной структуры person
// Она необходима для использования Point в качестве ключей в unordered map
struct PointHasher
{
 size_t operator()(const Point& pt) const
 {
 return hash<int>()(pt.x) ^ hash<int>()(pt.y);
 }
};

// Оператор сравнения, позволяющий сравнивать структуры типа Point.
// он необходим для использования Point в качестве ключей
bool operator==(const Point& pt1, const Point& pt2)
{
 return pt1.x == pt2.x && pt1.y == pt2.y;
}
```

```
void main()
{
 // отображение Point в string, позволяющее узнать название
 // объекта на картинке в точке с заданными координатами
 unordered_map<Point, string, PointHasher> pointDescriptions;

 pointDescriptions.emplace(Point{10, 20}, "apple");
 pointDescriptions[Point{11, 20}] = "apple";

 cout << pointDescriptions.at(Point{ 10, 20 }) << endl; // apple

 auto it = pointDescriptions.find(Point{ 11, 34 });
 if (it != pointDescriptions.end()) // вернет false
 {
 cout << it->second << endl;
 }
 else
 {
 cout << "No point description at {11, 34}" << endl;
 }
}
```

# Классы множеств std::set и std::multiset

- Ассоциативный контейнер, хранящий множество элементов определенного типа
  - set – дублирование элементов не допускается
  - multiset – дублирование элементов допускается
- Для использования данных классов необходимо подключить заголовочный файл <set>
- Требования к элементам – наличие операции отношения <
  - Возможно реализовать проверку упорядоченности иным способом при помощи объекта-параметра шаблона

# Пример

```
#include <set>
#include <string>
#include <iostream>

using namespace std;
int main(int argc, char *argv[])
{
 set<int> primeNumbers;
 primeNumbers.insert(2);
 primeNumbers.insert(3);
 primeNumbers.insert(5);
 if (primeNumbers.find(3) != primeNumbers.end())
 {
 cout << "3 is a prime number\n";
 }

 set<string> maleNames;
 maleNames.insert("John");
 maleNames.insert("Peter");

 return 0;
}
```



# Итераторы

- Итератор – объект, позволяющий программисту осуществлять перебор элементов контейнера вне зависимости от деталей его реализации
  - Например, осуществлять вставку диапазона элементов одного контейнера в другой
- Итераторы используются в STL для доступа к элементам контейнеров
  - Обобщенные реализации алгоритмов используют итераторы для обработки элементов контейнеров
    - Итератор – связующее звено между контейнером и алгоритмом

# Алгоритмы

- Обобщенные функции, реализующие типичные алгоритмы над элементами контейнеров
  - Сортировка, поиск, поэлементная обработка
- Алгоритмы в STL не работают с контейнерами напрямую
  - Вместо этого алгоритмы используют итераторы, задающие определенные элементы или диапазоны элементов контейнера
- Для работы с алгоритмами STL необходимо подключить заголовочный файл `<algorithm>`

# Пример: сортировка массива с использованием STL

```
#include <algorithm>
#include <functional>

int main()
{
 int array[5] = {3, 5, 1, 7, 9};

 // Сортируем массив по возрастанию
 std::sort(&array[0], &array[5]);

 // Сортируем по убыванию
 std::sort(&array[0], &array[5],
std::greater<int>());

 return 0;
}
```

```
#include <algorithm>
#include <functional>
#include <string>

struct Student
{
 std::string name;
 int age;
};

bool CompareStudentsByAge(Student const& s1, Student const& s2)
{
 return s1.age < s2.age;
}

int main()
{
 Student students[] = { {"Ivan", 20}, {"Alexey", 21}, {"Sergey", 19}, };

 std::sort(std::begin(students), std::end(students),
 [](Student const& s1, Student const& s2){
 return s1.name < s2.name;
 }); // Alexey, Ivan Sergey

 std::sort(&students[0], &students[sizeof(students)/sizeof(*students)],
 CompareStudentsByAge); // Sergey, Ivan, Alexey

 return 0;
}
```

# Пример

```
#include <string>
#include <vector>
#include <list>
#include <iterator>

using namespace std;
int main(int argc, char *argv[])
{
 vector<string> names;
 names.push_back("Peter");
 names.push_back("Ivan");
 names.push_back("John");

 list<string> namesList;

 sort(names.begin(), names.end());

 copy(names.begin(), names.end(), back_inserter(namesList));

 return 0;
}
```

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

using namespace std;

bool IsEven(int value)
{
 return (value % 2) == 0;
}

void FindFirstEvenValueInArray()
{
 int numbers[] = { 1, 3, 9, 10, 17, 12, 21 };

 auto it = find_if(cbegin(numbers), cend(numbers), IsEven);

 if (it != cend(numbers))
 {
 cout << "First even number in array is " << *it << endl;
 }
}
```

```
#include <algorithm>
#include <iostream>
#include <string>

void SearchingForRabbit()
{
 string animals[] = {
 "fox", "wolf", "snake", "turtle", "bear", "rabbit", "hare"};

 if (find(begin(animals), end(animals), "rabbit") != end(animals))
 {
 cout << "There is a rabbit among the animals" << endl;
 }
 else
 {
 cout << "There are no any rabbits" << endl;
 }
}
```

```
#include <algorithm>
#include <iostream>
#include <string>

using namespace std;

struct Person
{
 string name;
 int age;
};

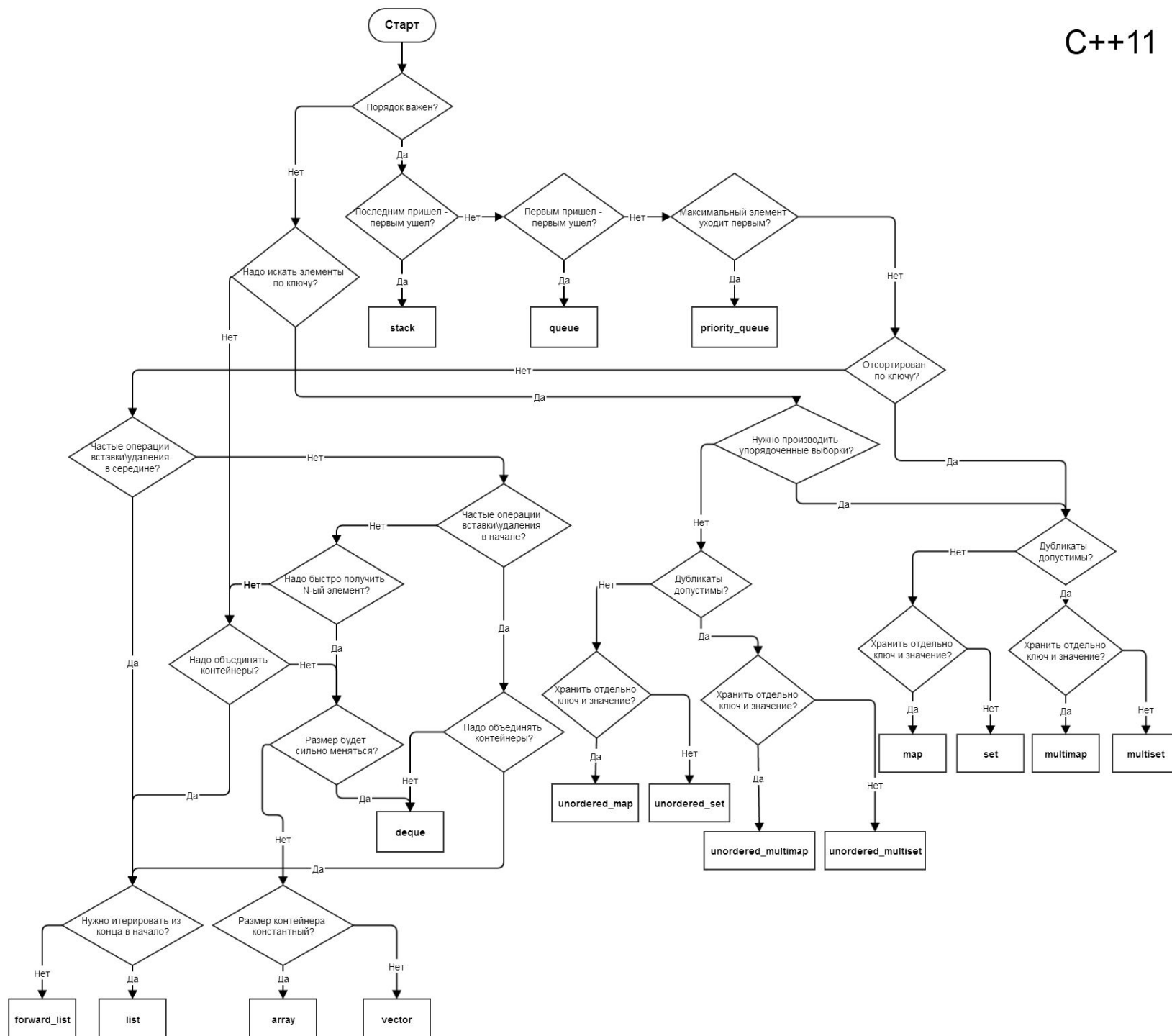
void TestWhetherThereIsAtLeastOneAdult()
{
 Person people[] = {
 { "Ivan", 4 }, { "Sergey", 16 }, { "Stepan Anatolievich", 65 },
 { "Maria Semenovna", 36 }, { "Egor", 13 }
 };

 if (any_of(begin(people), end(people), [](Person const& person){
 return person.age >= 18;
 })))
 {
 cout << "At least one person is an adult" << endl;
 }
 else
 {
 cout << "There are no adults" << endl;
 }
}
```



# Контейнеры STL и умные указатели

- Контейнеры STL автоматически освобождают занимаемую своими элементами память
- `std::vector` – рекомендуемая альтернатива динамическому массиву
- `std::unique_ptr` – умный указатель
- `std::shared_ptr` – умный указатель с подсчетом ссылок
- Прочее
  - `boost::scoped_ptr`
  - `boost::intrusive_ptr`
  - `boost::scoped_array`
  - `boost::shared_array`



# Ссылки

- [Алгоритм выбора контейнера STL](#)