А чем программы-то писать?

Почему именно плюсы?

- Популярны в среде спортивного программирования (исторически сложилось).
- Программы работают относительно быстро.
- Низкоуровневый контроль над работой программы.
- Мощные средства для ускорения написания кода.

И где писать на плюсах?

Хоть в блокноте! Лишь бы компилятор был.

Компиляторы:

- MinGW (Windows)
- GNU GCC (Linux, Mac)

Среды разработки:

- Dev-C++
- CodeBlocks (слегка круче)
- MS Visual Studio (совсем круто, но местами медленно)

Типичная олимпиадная программа

```
#include <iostream>
#include <cstdio>   // Подключаемые библиотеки
#include <string>
using namespace std;   // Пишем, чтобы не писать “std::” перед каждым вторым словом.

#define FILE_NAME "problem"   // Макросы

const int TARGET = 1;   // Константы и глобальные переменные

bool thisIsIt(int number) {   // Функции и структуры
    return number == TARGET;
}

                    // Основная функция
int main() {
    freopen(FILE_NAME".in", "r", stdin);
    freopen(FILE_NAME".out", "w", stdout);   // Настройки ввода/вывода.
    ios_base::sync_with_stdio(false);
    cin.tie(0);

    int number;
    string message = "I love numbers!";
    cin >> number;
    if (thisIsIt(number)) {
        cout << "This is " << TARGET << ", I'm sure.\n";
    }
    cout << message << '\n';

    return 0;   // Если не вернуть ноль, то может быть плохо.
}
```

Основные типы данных

- **char** (целое число/ASCII символ, 1 байт, -128..+127)
- **short (int)** (целое число, 2 байта, ~-32768..~+32767)
- **int** (целое число, 4 байта, ~-2e9..~+2e9)
- **long long (int)** (целое число, 8 байт, ~-8e18..~+8e18)
- **bool** (логическое значение, 1 байт, false/true)
- **double** (вещественное число, 8 байт, ~14-15 значащих цифр)

Для каждого целочисленного типа существует альтернативный **unsigned** тип (беззнаковый), принимающий только неотрицательные значения.

```
int x = -2;
usigned int y = 2;
```

Основные операторы

- Операторы присваивания: **=**, **+=**, **&=**, …
- Инкремент/декремент: **++**, **--**.
- Арифметические операторы: **+**, **-**, *****, **/**, **%** (остаток от деления).
- Логические операторы: **&&**, **||**, **!**.
- Операторы сравнения: **<**, **<=**, **==**, **>=**, **>**.
- Побитовые операторы: **&** (побитовое И), **|** (побитовое ИЛИ), **^** (побитовое исключающее ИЛИ), **~** (побитовое отрицание).
- Оператор приведения: **(newtype)**.
- Тернарное условие: **? :**
- Другие специальные операторы: для работы с памятью, для обращения к членам структуры, для работы с указателями, запятая, в конце концов.

Условия и циклы

- **if .. else** - условный оператор
- **while** – оператор цикла с предусловием
- **do .. while** – оператор цикла с постусловием
- **for** – оператор слегка продвинутого цикла с предусловием

```
int x = 12;
if (x % 3 == 2) {
    while (x) {
        --x;
    }
} else if (x % 3) {
    do {
        x -= 2;
    } while (x > 0);
} else {
    for (int i = 0; i < 10; ++i) {
        x -= i;
    }
}
```

Операторы прерывания

- **continue** – прерывает текущую итерацию цикла и переходит к следующей
- **break** – прерывает цикл
- **return** – прерывает работу функции

```
int main() {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        if (i % 2) {
            continue;
        }
        if (i == 6) {
            break;
        }
        sum += i;
    }
    cout << sum;
    return 0;
}
```

Ввод/вывод

В библиотеках реализованы 2 типа ввода/вывода:

- Форматный (наследие C) (**<cstdio>**)

```
int x, y;
scanf("%d%d", &x, &y);
printf("%d\n", x + y);
```

+ Гибкая система спецификаторов.
+ Трудоёмкий вывод бывает проще писать через форматный вывод.

- Потоковый (стильно, модно, молодёжно) (**<iostream>**)

```
int x, y;
cin >> x >> y;
cout << x + y;
```

+ Пишется попроще и побыстрей.
+ Некоторая универсальность. Решили изменить int на long long? Тут ничего не надо исправлять!
+ Больше типов объектов, которые вводятся/выводятся (например, контейнер string), а также можно научить потоки вводить/выводить объекты других типов.

Ввод/вывод

Для того чтобы контролировать формат потокового вывода, пользуйтесь библиотекой **<iomanip>**.

```
double x;
x = 1.200003;
cout << x << endl;
cout << setprecision(9) << x << endl;
cout << fixed << x << endl;
```

```
1.2
1.200003
1.200003000
```

Самый популярный способ подключить файлы для ввода/вывода:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Однако после выполнения этих строк программа теряет доступ к стандартным потокам ввода/вывода (т.е. к консоли), поэтому для использования консоли вам придётся убрать эти строки.

Ввод/вывод

Потоковый ввод/вывод часто оказывается медленней, чем форматный (особенно на старых компиляторах). Чтобы ускорить потоки до уровня форматов, используйте следующие магические строки:

```
ios_base::sync_with_stdio(false);
cin.tie(0);
```

Первая строка отключает синхронизацию между форматным и потоковым вводом/выводом; после неё форматный ввод/вывод будет работать некорректно, если мешать его с потоковым.

В интерактивных задачах лучше не использовать вторую строку, т.к. после неё ввод и вывод перестают должным образом синхронизироваться.

Ввод/вывод

Для ускорения вывода используйте вывод символа **‘\n’** вместо вызова манипулятора **endl**.

```
int two = 2;
cout << two << '\n';
cout << "This is two\n";
```

Однако в интерактивных задачах вам всё равно придётся использовать **endl**.

```
int two = 2;
cout << two << endl;
cout << "This is two" << endl;
```

Работа с памятью

Программы используют три вида памяти:

- Автоматическая (переменные, объявленные внутри любой функции, а также параметры функций и возвращаемые переменные функций)

Стоит помнить, что память в принципе не бесконечна, а автоматическая так вообще обычно ограничена считанными мегабайтами.

- Статическая (переменные, объявленные глобально или с помощью ключевого слова **static**)
- Динамическая (переменные, память для которых была выделена с помощью оператора **new** или с помощью функции **malloc()**)

```
int number1;
static int number2;
int* number3 = new int;
```

Указатели

Указатели – переменные, хранящие адрес некоторой ячейки в памяти.

Указатели есть по сути обычные числа, с которыми также работают некоторые арифметические операторы.

```
int variable = 13;
int* pointer;
pointer = &variable;
cout << *pointer << endl;
```

После выделения динамической памяти под переменную как раз возвращается указатель.

```
int* pointer = new int(3);
cout << *pointer;
delete pointer;
```

Массивы

Массивы = указатели, для которых выделили много памяти.

```
int autoArray[15];
for (int* ptr = autoArray; ptr != autoArray + 15; ++ptr) {
    *ptr = 2;
}
cout << autoArray[6] << endl;
int* dynamicArray = new int[15];
for (int* ptr = dynamicArray; ptr != dynamicArray + 15; ++ptr) {
    *ptr = 3;
}
cout << dynamicArray[6] << endl;
delete[] dynamicArray;
```

Размеры массивов неизменны. Единственное, что можно сделать – выделить новый участок памяти, перекинуть туда старые элементы и освободить старый.

Учтите, что первый способ создания массива из представленного примера пользуется автоматической памятью. Старайтесь избегать этого и объявлять массивы динамически/статически, чтобы не вызвать stack overflow.

Массивы

Следующий пример демонстрирует массивы, созданные в статической памяти.

```
int globalArray[10];

int main() {

    static int staticArray[10];
```

Созданные таким образом массивы изначально заполнены нулями (для массивов структур там вызывается конструктор по умолчанию, об этом позже). Впрочем, и для инициализации обычных массивов нулями тоже необязательно по ним пробегаться в цикле.

```
int arr[666] = {};
```

Массивы

Многомерные массивы тоже есть. Но легко и просто создавать их только в автоматической и статической памяти.

```
int arrOfSomething[10][20][30][40];
int arrOfZeroes[10][20][30][40] = {};
```

Да, и помните, что если вы создали массив в автоматической или динамической памяти, не проинициализировав его, то он совсем необязательно будет заполнен нулями. Там могут содержаться совсем случайные значения.

Ссылки

Ссылка – обёртка указателя, с которой можно работать, как с обычной переменной.

Работая с ссылкой, вы на самом деле работаете с переменной, на которую она указывает.

```
int number = 1;
int& link = number;
cout << number << ' ' << link << '\n';
link = 2;
cout << number << ' ' << link << '\n';
number = 3;
cout << number << ' ' << link << '\n';
```

```
1 1
2 2
3 3
```

После инициализации ссылки её нельзя изменить, заставить указывать на другую переменную.

Функции

Одним main-ом сыт не будешь – стоит иногда заниматься написанием других функций.

```
int fun(int x, int y) {
    x += y;
    y += x;
    return y;
}

int main() {
    int a = 2;
    a += fun(a, 3);
    cout << a << '\n';
    return 0;
}
```

Любая функция принимает на вход некоторые параметры определённых типов (ну почти любая функция) и возвращает некоторое значение определённого типа (по достижению ключевого слова **return**). Либо функция ничего не вернёт, если в качестве типа возвращаемого значения указать **void**.

Функции

Функция «видит» только объявленные глобально и объявленные локально (т.е. внутри функции) переменные.

Параметры функции – это локальные переменные, в которые перед работой функции <u>копируются</u> некоторые другие переменные откуда-то извне (откуда она вызывается).

Чтобы изменить внутри функции внешние переменные, передавайте их по ссылке или по указателю.

```
void fun(int x) {
    x += 2;
}

int main() {
    int a = 2;
    fun(a);
    cout << a << '\n';
    return 0;
}
```

```
2
```

```
void fun(int& x) {
    x += 2;
}

int main() {
    int a = 2;
    fun(a);
    cout << a << '\n';
    return 0;
}
```

```
void fun(int* x) {
    *x += 2;
}

int main() {
    int a = 2;
    fun(&a);
    cout << a << '\n';
    return 0;
}
```

```
4
```

Функции

Функция может вызывать саму себя и любые другие функции, объявленные выше.

```
void write(int x) {
    cout << x << '\n';
}

void go(int x) {
    if (x <= 0) {
        return;
    }
    go(x - 1);
    write(x);
}

int main() {
    go(5);
    return 0;
}
```

```
1
2
3
4
5
```

Однако помните, что для каждого нового вызова функции выделяется некоторая память из числа автоматической (которая освобождается только после завершения её работы).

Поскольку количество доступной автоматической памяти обычно сильно ограничено, чрезмерное количество вызовов может привести к **stack overflow** и вылету программы.

Функции

```
void write(int x) {
    cout << x << '\n';
}

void go1(int x);

void go2(int x) {
    if (x <= 0) {
        return;
    }
    go1(x - 1);
    write(x);
    write(x);
}

void go1(int x) {
    if (x <= 0) {
        return;
    }
    go2(x - 1);
    write(x);
}

int main() {
    go1(5);
    return 0;
}
```

Можно сделать и так, чтобы функции могли вызывать друг друга, прописав перед реализацией функций их прототипы.

```
1
2
2
3
4
4
5
```

Функции

Функции можно перегружать, т.е. можно объявить несколько функций под одним и тем же именем, но с разной реализацией. Такие функции обязаны отличаться либо количеством аргументов, либо при одинаковом их количестве типами этих аргументов.

```
void write(int x) {
    cout << x << '\n';
}

void write(int x, int y) {
    cout << x << ' ' << y << '\n';
}

void write(int x, int* y) {
    cout << x << ' ' << *y << 'P' << '\n';
}

void write(int x, double y) {
    cout << x << ' ' << y << 'D' << '\n';
}

int main() {
    int a = 3, b = 4;
    double d = 5.5;
    write(a);
    write(a, b);
    write(a, &b);
    write(a, d);
    return 0;
}
```

Функции

```
void writeLol(int k) {
    while (k--) {
        cout << "LOL\n";
    }
}

void writeKek(int k) {
    while (k--) {
        cout << "KEK\n";
    }
}

void writeSomething(int k, void (*write)(int)) {
    while (k--) {
        write(k + 1);
    }
}

int main() {
    void (*ptr)(int) = writeKek;
    writeSomething(2, ptr);
    writeSomething(3, writeLol);
    return 0;
}
```

Наконец, существуют так называемые указатели на функции – переменные... указывающие на функции.

Зачем? Например, их можно передавать в другие функции!

Структуры

Структура – совокупность переменных, объединённых под общим именем.

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

int main() {
    Point p;
    p.x = 2;
    p.y = 3.5;
    cout << p.x << ' ' << p.y << '\n';
    return 0;
}
```

Переменные в структуре (называемые полями) хранятся в соседних ячейках памяти (как и элементы массива, например).

Структуры

Помимо переменных, структура может содержать функции для их обработки (называемые методами). Они ведут себя как обычные функции, только:

- Вызываются от соответствующих структур.
- «Видят» все поля структуры, от которой были вызваны, и могут их модифицировать (если только не прописать ключевое слово **const**).

```
struct Point {
    int x, y;

    void read() {
        cin >> x >> y;
    }

    int square() const {
        return x * y;
    }
};

int main() {
    Point p;
    p.read();
    cout << p.square() << '\n';
    return 0;
}
```

Структуры

Существует специальные типы методов, предназначенные для инициализации и корректного удаления структуры.

- Конструктор запускается при создании структуры.

Конструктор можно перегрузить, как и обычные функции.

```
struct Pair {
    int a, b;

    Pair() :
        a(1), b(2) {}

    Pair(int x, int y) {
        a = x, b = 2 * y;
    }

    void write() const {
        cout << a << ' ' << b << '\n';
    }
};

int main() {
    Pair p(10, 10);
    p.write();
    return 0;
}
```

```
10 20
```

Структуры

Конструктор, не принимающий аргументов, называется конструктором по умолчанию. Если при создании экземпляра структуры не определяется, какой конструктор нужно вызвать, вызывается он.

```
int main() {
    Pair p;
    p.write();
    Pair pp[4];
    for (int i = 0; i < 4; ++i) {
        pp[i].write();
    }
    return 0;
}
```

```
1 2
1 2
1 2
1 2
1 2
```

Любой массив структур, содержащих прописанный конструктор по умолчанию, изначально инициализируется соответствующим образом, в отличие от массивов обычных чисел.

Структуры

- Деструктор вызывается перед освобождением памяти, выделенной под структуры.

Деструктор не принимает никаких параметров и ничего не возвращает.

```
int counter = 0;

struct Counted {
    Counted() {
        ++counter;
    }

    ~Counted() {
        --counter;
    }
};

int main() {
    cout << counter << '\n';
    {
        Counted c;
        cout << counter << '\n';
    }
    cout << counter << '\n';
    return 0;
}
```

```
0
1
0
```

Константы

Язык позволяет объявить переменные, изменить которые после инициализации нельзя.

Можно объявить константную ссылку на переменную, тогда в то время как переменную можно будет изменять напрямую, через ссылку это сделать будет невозможно. Так же по аналогии можно объявить константный указатель.

```
const int a = 20;
int b;
const int& bl = b;
const int* bp = &b;
```

Структуры тоже можно сделать константными, тогда нельзя будет вызвать методы, их модифицирующие (но можно вызывать «константные», помеченные модификатором **const**).

Статические переменные

В функции можно объявить статические переменные: такие переменные не будут очищены после конца первого вызова функции и будут доступны всем следующим вызовам.

```
void inc() {
    static int x = 0;
    ++x;
    cout << x << '\n';
}

int main() {
    for (int i = 0; i < 5; ++i) {
        inc();
    }
    return 0;
}
```

```
1
2
3
4
5
```

Статические переменные (не объявленные глобально) видны только внутри своей функции.

Также они используют статическую память.

Статические переменные

```
struct Counted {
    Counted() {
        ++counter;
    }

    ~Counted() {
        --counter;
    }

    static int counter;
};

int Counted::counter = 0;

int main() {
    cout << Counted::counter << '\n';
    {
        Counted a, b, c;
        cout << Counted::counter << '\n';
    }
    cout << Counted::counter << '\n';
    return 0;
}
```

По аналогии с функциями, структуры так же могут иметь статические поля. Для всех экземпляров одной структуры будет существовать единственная статическая переменная, к которой все эти экземпляры будут иметь доступ.

```
0
3
0
```

Перегрузка операторов

Операторы **+**, **-**, **&&**, **++** и все остальные - это обычные функции. И как обычные функции, вы можете перегрузить операторы, научить плюс, минус или восклицательный знак работать с вашими, например, структурами.

Операторы можно перегрузить либо как метод структуры, либо как просто обычную функцию (некоторые операторы можно перегрузить лишь одним из этих способов).

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

Point operator + (const Point& l, const Point& r) {
    Point res;
    res.x = l.x + r.x;
    res.y = l.y + r.y;
    return res;
}


int main() {
    Point a, b, c;
    a.x = 0;
    a.y = 5;
    b.x = 6;
    b.y = 9;
    c = a + b;
    return 0;
}
```

Контейнеры

Контейнеры – структуры данных, уже реализованные в стандартной библиотеке.

Иногда недостаточно обычного массива для хранения кучи элементов, нужна саморасширяющаяся структура данных – вектор, например (**vector**, **<vector>**). Иногда недостаточно простого саморасширяющегося массива, нужна функциональность очереди (**queue**, **<queue>**). Иногда недостаточно простой очереди, необходима очередь с приоритетами (**priority_queue**, **<queue>**). Иногда…

```
vector<int> vct = { 3, 8, 17, 5 };
for (int x = 4; x < 100; x *= (x % 5 + 2)) {
    vct.push_back(x);
}
sort(vct.begin(), vct.end());
for (vector<int>::iterator it = vct.begin(); it != vct.end(); ++it) {
    cout << *it << ' ';
}
```

Контейнеры

Для удобного перебора элементов того или иного контейнера расширили возможности оператора **for**. Вместо того, чтобы вручную прогонять итератор (указатель на элемент контейнера) по контейнеру, можно приказать программе в простой форме пройтись по всем элементам и вернуть от каждого ссылку или копию.

```
for (vector<int>::iterator it = vct.begin(); it != vct.end(); ++it) {
    ++*it;
    cout << *it << ' ';
}
cout << endl;
for (const int& el : vct) {
    cout << el << ' ';
}
cout << endl;
for (int& el : vct) {
    ++el;
    cout << el << ' ';
}
cout << endl;
for (int el : vct) {
    cout << el << ' ';
}
cout << endl;
```

Автоматическое определение типа

Вы можете использовать ключевое слово **auto**, когда по каким-то причинам не хотите прописывать тип очередной объявляемой переменной и если уверены, что компилятор правильно определит этот тип.

Использование этого слова не особенно приветствует, так как затрудняет чтение и восприятие кода.

```
auto num = 3 + 2;
auto it = vct.begin();
for (auto& el : vct) {
    ++el;
    cout << el << ' ';
}
cout << endl;
```

Библиотеки

C++ довольно богат встроенными функциями и типами, облегчающими жизнь спортивного программиста, которые расположены в различных подключаемых библиотеках. Все они подключаются в начале программы с помощью директивы **#include**.

Если вам лень запоминать, что где лежит, можно воспользоваться нижеследующей строчкой вместо всех подключений. Это может не работать в старых компиляторах.

```
#include <bits/stdc++.h>
```

Необходимые и рекомендуемые к использованию библиотеки:

- **<cstdio>**

Предоставляет возможности форматного ввода/вывода, доступа к файлам. Также предоставляет функции для ввода из строк/вывода в строки (реализованных, как массивы **char**).

Библиотеки

- **<iostream>**

Предоставляет возможности потокового ввода/вывода, доступа к файлам.

- **<iomanip>**

Предоставляет манипуляторы для форматирования потокового ввода/вывода.

- **<sstream>**

Позволяет использовать строки-потоки, потоковая альтернатива форматному вводу/выводу в строки.

- **<cctype>**

Предоставляет удобные функции для определения типа символа (цифра, буква и т.д.), а также для изменения регистра букв.

Библиотеки

- **<cmath>**

Предоставляет широкий набор математических функций, работающих с вещественными числами.

- **<cstdlib>**

Содержит некоторые полезные функции (мгновенного завершения программы, конвертации си-строки в число), а также базовые инструменты для генерации рандомных чисел.

- **<ctime>**

Содержит функции для работы со временами, а также функции для измерения времени работы программы.

- **<climits>**

Содержит константы минимальных/максимальных значений целочисленных типов.

Библиотеки

- **<algorithm>**

Предоставляет широкий набор реализованных алгоритмов и просто удобных функций. Различные варианты поиска, тестов, подсчёта на массивах и контейнерах; перемещение, удаление, сортировка, перемешивание отрезков массивов и контейнеров; порядковая статистика, бинарный поиск, генерация перестановок; **swap**; минимум, максимум из двух элементов и на отрезке.

- **<random>**

Предоставляет продвинутые инструменты для генерации псевдослучайных чисел различных типов, в различных диапазонах и различными методами.

Библиотеки

- **<string>** - контейнер для хранения строк
- **<vector>** - вектор, динамически расширяемый массив
- **<list>** - двусвязный список
- **<queue>** - очередь; приоритетная очередь
- **<stack>** - стек
- **<deque>** - дек
- **<set>**, **<map>** - красно-чёрные деревья
- **<unordered_set>**, **<unordered_map>** - хэш-таблицы