Дисклеймер.

Автор не несет ответственности за любой ущерб, причиненный Вам при использовании данного документа. Автор напоминает, что данный документ может содержать ошибки и опечатки, недостоверную и/или непроверенную информацию. Если Вы желаете помочь в развитии проекта или сообщить об ошибке/опечатке/неточности:

GitHub проекта

Автор в ВК

Содержание

1 Базовые конструкции программирования: синтаксис и семантика языков высокого уровня; переменные, типы, выражения и присваивания; простейший ввод/вывод.

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур, называемых следованием, ветвлением и циклом. Этот результат установлен Боймом и Якопини путем преобразования любой программы в эквивалентную, состоящую только из комбинаций данных структур. Следование, ветвление и цикл — базовые конструкции структурного программирования.

Оператор — фраза алгоритмического языка, определяющая законченный этап обработки данных. В состав операторов входят ключевые слова, данные, выражения и др.

Следование (или последовательность выполнения) — способ упорядочения инструкций программы в процессе её выполнения.

```
//code
    a = 1, b = 2, a+b;
    или
    a = 1;
    b = 2;
    a+b;
//end of code
```

Ветвление задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

```
//code
    if (a < b)
    {
        a = 12;
    }
    else
    {
        b = 6;
    }
//end of code
```

Цикл задает многократное выполнение оператора, пока выполняется условие.

```
//code
    while (i > 0)
    {
        cout << "vsya fignya";
        i-- // декремент для выхода из цикла
    }
//end of code
```

Особенностью данных конструкций является только один вход и выход, что позволяет произвольно вкладывать их друг в друга.

В большинстве языков существует несколько реализаций базовых конструкций. Например, в C++ существует 3 вида циклов и 2 вида ветвлений.

Синтаксис языка программирования — правила построения сообщений в системе.

Семантика в программировании — система правил определения поведения отдельных языковых конструкций. Семантика определяет смысловое значение предложений алгоритмического языка.

Синтаксис проверяется на ранних этапах трансляции.

Переменная — поименованная либо адресуемая иным способом область памяти, адрес которой можно использовать для осуществления доступа к данным. Данные, находящиеся в переменной (по данному адресу памяти) называются значением данной переменной.

Типы данных:

Типы данных в C:

char — символьный тип;

int — целочисленный тип;

float — вещественный тип;

double — вещественный двойной точности;

void — не имеющий значения.

Модификаторы типа:

unsigned — беззнаковый тип (не имеет отрицательной части).

short — "короткий" тип (двухбайтовый).

long — "длинный" тип (4-10 байтовый).

Выражения

Выражение - конструкция на языке программирования, предназначенная для выполнения вычислений. Выражение состоит из операндов, объединен-

ных знаками операций. Различают арифметические, логические и символьные выражения.

Первичные выражения — элементы, из которых строятся более сложные выражения. К таковым относятся литералы (основные константные выражения (100, 'c')), имена (названия переменных, функций), уточняемые имена (std::cout, где std — область (namespace, пространство имен), а ::cout — глобальная функция).

Операторы или логические операторы — операции сравнения, позволяющие строить логические выражения (==, <, >, >=, <=, !=)

Сложные логические выражения — простые логические выражения, объединенные *операциями* (|| — «or», && — «and», ! — «not»). Результатом данных операций также будут являться логические значения (true, false).

Не уверен в полноте этого блока. В интернете информации немного, если кто-нибудь что-то добавит — буду очень рад

Присваивания

Присваивание — механизм, позволяющий динамически изменять связи объектов данных (переменных) с их значениями. На физическом уровне результат проведения операции состоит в проведении записи или перезаписи ячеек памяти или регистров процессора.

Является одной из центральных конструкций в императивных языках программирования.

```
//code
double a = 10.34; long int b = 2016;
int arr[10] = {0}; string say = "Hello, world!";
//end of code
```

Некоторые языки допускают множественность целевых объектов (C++ в т.ч.):

```
//code
int a, b, c;
a = b = c = 10;
//при этом компилятор видит это так:
a = (b = (c = 10));
//в C++ оператор присваивания возвращает присваиваемое значение.
//end of code
```

Некоторые языки допускают параллельное присваивание (Python, Ruby, но не C++!):

```
//code
a, b = 10, 20 #эта информация дана для ознакомления. Можно будет
блеснуть знаниями перед Тереховым :D
//end of code
```

Также большинством языков поддерживаются составные операции присваивания (+=, -=, *=, /=, %=)

Ввод и вывод

Так как C++ является наследником языка C, то поддерживает два типа простейшего ввода и вывода: традиционный (в стиле C) и ввод-вывод с помощью потоков STL.

Традиционный ввод-вывод реализуется при помощи заголовочного файла stdio.h в C или cstdio в C++. Он предоставляет необходимый набор функций для ввода-вывода при помощи scanf() и printf().

При запуске консольного приложения неявно открываются три потока: stdin — для ввода с клавиатуры, stdout — для буферизованного вывода на монитор и stderr — для небуферизованного вывода на монитор сообщений об ошибках. Эти три потока определены посредством <cstdio>.

```
//code on "pure C"
#include <stdio.h> //cstdio для C++

int main()
{
    int i; float a;
    printf("Введите i и a\n"); /* вывод приглашения к вводу */
    scanf("%d%f", &i, &a); /* непосредственно ввод */
    return 0;
}
//end of code
```

Файловый ввод-вывод в традиционном стиле реализуется при помощи того же заголовочного файла и fprintf() и fscanf(). Синтаксис примерно тот же, за исключением того, что в качестве аргумента обязательно должен быть передан объект, связанный с файлом из которого производится чтение (или в который производится запись).

```
//code
#include <cstdio>
using namespace std;

int main()
{
    FILE * fin = fopen("filein.txt", "r");
    FILE * fout = fopen("fileout.txt", "w");

    int data;
    fscanf(fin, "%d", &data);
    // в переменную data ушло десятичное целое
    fprintf(fout, "%d\n", data*data);
    // а теперь мы пишем её квадрат в файл fileout.txt

    return 0;
}
//end of code
```

Ввод-вывод через потоки STL реализуется при помощи заголовочного файла iostream. Он предоставляет необходимый набор функций для ввода-вывода посредством cin и cout.

При запуске консольного приложения неявно открываются четыре потока: cin — для ввода с клавиатуры, cout — для буферизованного вывода на монитор, cerr — для небуферизованного вывода на монитор сообщений об ошибках и clog — буферизованный аналог cerr. Эти четыре потока определены посредством <iostream>.

```
//code
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, world!";
    int a;
    cin >> a;
}

//end of code
```

Файловый ввод-вывод реализуется при помощи заголовочного файла fstream. Не забываем определить объекты ввода и вывода.

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

int main()
{
    ifstream fin("filein.txt");
    ofstream fout("fileout.txt");
    int data;
    fin >> data;
    // в переменную data ушло десятичное целое
    fout << data*data;
    // а теперь мы пишем её квадрат в файл fileout.txt

    return 0;
}
//end of code
```

2 Условные предложения и итеративные конструкции

Условные предложения или **ветвления** задают выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

В языке C++ имеется 2 вида ветвлений:

- Конструкция if-else:

```
//code
if (a > b)
{ //Терехов пользуется подобным оформлением скобок
    a = a/2;
    b = b*2;
} //Не рискуйте: будьте как Терехов :)
else if (a < b)
{
    a = a*2;
    b = b/2;
}
else
{
    a = b;
}
//end of code
```

Просьба понимать, что в данном коде оператор else if не является отдельной командой. Это всего лишь комбинация else и нового if.

- Конструкция switch-case:

Является вариантом множественного выбора.

```
//code
int number;
cin >> number;
switch (number)
{
    case 1:
        cout << "Hello, world!";
        break;
    case 2:
        cout << "Goodbye, world!";
        break;
    case 3:
        cout << "Мне уже лень, придумайте сами";
        break;
    default :
        cout << "Терехов лучший! :)";
}
//end of code
```

Стоит помнить, что переменная, созданная внутри условного оператора, в нем и умрет.

Break в switch нужно ставить для предотвращения «провалов управления» — ситуации, когда выполнится не только команды в «нужном» case, но и во всех последующих.

Итеративная конструкция или **цикл** задает многократное выполнение оператора, пока выполняется условие.

- Конструкция for.

```
//code
for (int i = 0; i < 10; i++)
    cout << "Итерация номер" << i << endl;
//end of code
```

Поток выполнения данного оператора:

1) При первом вхождении объявляется переменная i, которой присваивается значение.

2) Проверяется соответствие условию i $<$ 10. Если true, то происходит выполнение команд, входящих в тело цикла. Если false, то выход из цикла.

3) Затем выполняется инкремент i++ (или любое другое выражение, записанное на этом месте).

4) Передача управления в точку 2).

Кстати, программист волен по своему усмотрению пропускать записи в скобках после for(). К примеру, for(; ;) является бесконечным циклом.

- Конструкция while.

```
//code
while (true)
{
    cout << "Come on, Java! Это Java!";
}
//на самом деле, это C++, а кто понял пасхалку, тот молодец.
//end of code
```

Поток выполнения:

1) Проверка условия в скобках.

2) true — выполнить тело цикла, false — не выполнять.

- Конструкция do ... while.

```
//code
do
{
    i = 1;
    cout << "Это выполнится хотя бы " << i << "раз";
    i++;
}
while (i < 2);
//end of code
```

Поток выполнения:

1) Выполнить тело цикла.
2) Проверить условие в скобках.
3) Если true, то передать управление в точку 1).

Операторы break и continue:

break работает в любых типах цикла и операторе switch, досрочно прекращая выполнение цикла и передавая управление за его пределы.

continue досрочно начинает следующую итерацию цикла.

3 Функции и передача параметров; структурная декомпозиция

Функция — это фрагмент кода или алгоритм, реализованный на языке программирования, с целью выполнения определённой последовательности операций. Функции позволяют сделать программу модульной, то есть разделить программу на несколько маленьких подпрограмм (функций), которые в совокупности выполняют поставленную задачу. Еще один огромнейший плюс функций в том, что их можно многократно использовать.

Функции в C++ могут быть определены не только в тексте программы, но и в заголовочных файлах. К примеру, функции pow() и sqrt() определены в заголовочном файле <cmath>.

Определяемые в тексте программы функции подразделяются, в свою очередь, на функции, возвращающие значения, и функции, значения не возвращающие. Функция, не возвращающая значения, называется **процедурой**.

Общий синтаксис:

```
//code
тип_возвращ_значения имя_функции(параметры)
{
    тело_функции;
    return возвращаемое_значение;
}
//end of code
```

Возвращаемые значения функций могут быть любыми, за исключением массивов. Однако вы всегда можете вернуть массив, «обернув» его в структуру.

Пример процедуры:

```
//code
void procedure(int parametr)
{
    parametr++;
}
//end of code
```

Понятно, что практического смысла такая функция не несет. Процедуры стоит использовать для взаимодействия с внешними объектами: вывода (но ни в коем случае не ввода!) на экран или файл с предварительной обработкой, изменения массивов или переменных, передаваемых по ссылке.

Пример функции, возвращающей значение:

```
//code
```

```
double summator(double value1, double value2)
{
    return value1+value2;
}
//end of code
```

Вызов функции:

```
//code
a = summator(b, c);
//end of code
```

При этом управление будет передано функции, созданы локальные (т.е. не видимые за пределами функции (инкапсуляция, аднака)) копии переменных b и c, их значения присвоены переменным value1 и value2 (которые, как вы уже догадались, и являются этими локальными копиями). Будет выполнены все операторы внутри функции, оператор return передаст управление обратно в «родительскую» функцию, инициализировав присвоение возвращаемого значения переменной a.

В качестве параметров функции может быть передано что угодно, в том числе массивы (будут переданы по ссылке), структуры и даже сами функции!

Функции в C++ можно объявлять в самом файле программы (до или после функции main), или же во внешних файлах *.cpp, которые затем будут связаны.

- Все функции в одном файле.

1) Функции перед main():

В этом случае не требуется прототипирования функций, поскольку они обрабатываются до их вызова из функции main(), т.е. во время компиляции main() компилятор уже знает о всех составляющих программу функциях. Такой способ имеет свои минусы: сама функция всегда должна следовать перед своим вызовом, да и объявить взаимно рекурсивные функции (функции, вызывающие друг друга) не получится.

2) Функции после main():

Хорошему стилю программирования соответствует способ объявления функций после main(). Однако в таком случае требуется прототипирование функций.

```
//code
#include <iostream>

void say_hello(int n); //прототип
```

```
int main()
{
    std::cout << "Сколько раз сказать?";
    int count;
    std::cin >> count;
    say_hello(count); //теперь компилятор знает
    // о существовании этой функции
    return 0;
}

void say_hello(n)
{
    for (int i = 0; i < n; i++)
        std::cout << "Hello, world!\n";
}
//end of code
```

Вообще прототип требует лишь объявления типов переменных и имен функций, так что прототип в предыдущем примере можно переписать так:

```
//code
void say_hello(int);
//end of code
```

Такой фокус (как оказалось) прокатит и с массивами, и с указателями:

```
//code
int many_money(double [], int, int);
int many_money(int *, char [], short int);
//end of code
```

- Функции в разных файлах.

Внимание! В этом разделе информация дана для ознакомления и на случай «вдруг спросит». Рассказывайте это на экзамене только в том случае, если вы понимаете, что говорите.

Понятно, что большие проекты не пишутся в один файл. Здесь будет разобран простой пример программы, в которой функция вынесена в отдельный файл.

Структура такой программы следующая: в директории с программой создаются 3 (или больше) файлов:

1) Файл с функцией main(). Назовем его main.cpp.

2) Файл с прототипами используемых функций с расширением *.h. Назовем его hello.h.

3) Файл(ы) с функцией(ями). Их может быть сколь угодно большое количество, главное, чтобы все функции были прототипированы в hello.h. В нашей программе такой файл один и мы назовем его hello.cpp.

Содержимое файлов:

hello.h

```
//code
#ifndef hello //не забываем стража
#define hello

void say_hello();

#endif
//end of code
```

hello.cpp

```
//code
#include <iostream>

void say_hello()
{
    std::cout << "Hello, world!\n";
}
//end of code
```

main.cpp

```
//code
#include "hello.h" //не забываем эту строчку
//она указывает препроцессору, в какой файл смотреть
//для объявления функций

int main()
{
    say_hello();
    return 0;
}
//end of code
```

В процесс компиляции такой программы добавляется еще один шаг: **линковка** — сборка исполняемого модуля из нескольких объектных.

Как все это происходит. Этапы:

1) Препроцессинг. Препроцессор обрабатывает файл main.cpp, «натыкается» на #include "hello.h" и включает его содержимое в код основного файла.

2) Компиляция. Компилятор компилирует файлы main.cpp и hello.cpp. В большинстве IDE это должно происходить автоматически, но если вы пользуетесь консольным компилятором, вам придется указать это прямо. На выходе компилятор выдает 2 **объектных файла**. Объектные файлы можно назвать программами в полном смысле этого слова (это уже бинарные коды), если не учитывать то, что по отдельности они бесполезны: один из них содержит вызов функции say_hello(), определенной в другом файле, а второй вообще не запустится, ибо в нем нет функции main(). Короче — эти файлы были бы мусором, если бы не

3) Линковка. Линковщик получает на вход два объектных файла и делает из них один, полноценный.

Обычно всю эту работу берет на себя IDE, но знать подробности не помешает.

Структурная декомпозиция — разделение целого на части. В программировании этот термин означает разделение сложного проекта (программы) на более мелкие части. Структурная декомпозиция является основой принципа «разделяй и властвуй».

Разделение программы на множество мелких частей, каждая из которых способна независимо выполнять свой этап общей работы — первый способ декомпозиции. Основой этого принципа является проектирование «сверху вниз».

Второй способ декомпозиции — разделение системы по признаку принадлежности её элементов различным абстракциям данной проблемной области. В таком случае «мир» программы будет представлять собой совокупность взаимодействующих объектов. Такой подход называется объектно-ориентированным.

Структурная декомпозиция в основном представляется в виде графа, где задача разбивается на несколько подзадач, каждая из которых в свою очередь подразделяется еще на несколько подзадач и так далее.

Немного о принципе «Разделяй и властвуй». Данный принцип заключается в рекурсивном разбиении решаемой задачи на две и более подзадач такого же типа, но меньшего размера, и комбинирования их решений для получения ответа к исходной задаче. Если говорить более приземленно, то мы должны разбивать задачу на подзадачи до тех пор, пока все подзадачи не станут элементарными, а затем решить их и «соединить» ответы для получения ответа к исходной задаче.

Типичными примерами являются двоичный поиск и сортировка слиянием или QuickSort.

Основой для принципа «Разделяй и властвуй» служат функции.

Результирующая последовательность действий по решению задачи называется **алгоритмом**.

4 Алгоритмы и решение задач: стратегии решения задач, роль алгоритмов в решении задач, стратегии реализации алгоритмов, стратегии отладки, понятие алгоритма, свойства алгоритмов

Алгоритм — набор инструкций, описывающий порядок действий исполнителя для достижения результата решения задачи за конечное число действий, при любом наборе исходных данных. Понятие алгоритма относится к базисным понятиям математики.

Слово «алгоритм» происходит от имени индийского ученого Абу-Абдуллаха аль-Хорезми, который в VIII веке впервые описал десятичную систему счисления и ввел понятие нуля как значения пропущенных разрядов.

Современное формальное понятие алгоритма было дано в 30-50-х годах XX века в трудах Тьюринга, Поста, Чёрча, Маркова и других.

Первым программистом считается баронесса Ада Лавлейс, впервые описавшая алгоритм вычисления чисел Бернулли на аналитической машине Чарльза Беббиджа.

Решение задач — мыслительный процесс направленный на достижение цели, заданной в рамках проблемной ситуации (задачи).

Процесс решения задачи:

1) Нахождение проблемы;

2) Постановка задачи;

3) Решение задачи;

Стратегии решения задач:

1) Решение задач «из начала в конец». Применяется, когда решение задачи с начала более обосновано.

2) «Из конца в начало». Данная стратегия применяется в случае, если из конечной точки ведет меньше путей решения, нежели из изначальной.

3) Индуктивный метод. Рассуждение от частного к общему.

4) Дедуктивный метод. Рассуждение от общего к частному.

В общем случае стратегия решения задачи может быть представлена как алгоритм. Цель такого алгоритма — поиск окончательного или конкретного решения. В зависимости от сложности задачи алгоритмы могут отличаться по сложности, однако известно одно: стратегию решения любой задачи можно представить в качестве алгоритма, выполнив который, исполнитель получит верный ответ. Чем больше вы знаете об алгоритмах, тем больше у вас шансов найти хорошее решение проблемы. Во многих случаях новая задача легко сводится к старой, но для этого нужно иметь фундаментальное понимание

старых задач.

Полученный алгоритм (программа) должен обладать следующим набором свойств:

- дискретность (алгоритм разбит на отдельные шаги - команды);
- однозначность (каждая команда определяет единственно возможное действие исполнителя);
- понятность (все команды алгоритма входят в систему команд исполнителя);
- результативность (исполнитель должен решить задачу за конечное число шагов).

Большая часть алгоритмов обладает также свойством массовости (с помощью одного и того же алгоритма можно решать множество однотипных задач).

Отладка — этап в разработке программы, состоящий в выявлении и устранении программных ошибок, факт существования которых уже установлен.

Отладка бывает двух видов:

1) Синтаксическая отладка. Синтаксические ошибки выявляет компилятор, поэтому исправлять их достаточно легко.

2) Семантическая (смысловая) отладка. Ее время наступает тогда, когда синтаксических ошибок не осталось, но результаты программа выдает неверные. Здесь компилятор сам ничего выявить не сможет.

Стратегии отладки

Принципы локализации ошибок:

Большинство ошибок обнаруживается вообще без запуска программы - просто внимательным просматриванием текста. Чрезвычайно удобные вспомогательные средства - это отладочные механизмы среды разработки: трассировка (процесс пошагового выполнения программы), промежуточный контроль значений.

Принципы исправления ошибок еще больше похожи на законы Мерфи:

- Там, где найдена одна ошибка, возможно, есть и другие.
- Вероятность, что ошибка найдена правильно, никогда не равна ста процентам.
- Наша задача - найти саму ошибку, а не ее симптом.

Стратегии отладки:

- Метод индукции - анализ программы от частного к общему.

  Просматриваем симптомы ошибки и определяем данные, которые имеют к ней хоть какое-то отношение. Затем, используя тесты, исключаем маловероятные гипотезы, пока не остается одна, которую мы пытаемся уточнить и доказать.

- Метод дедукции - от общего к частному.

  Выдвигаем гипотезу, которая может объяснить ошибку, пусть и не полностью. Затем при помощи тестов эта гипотеза проверяется и доказывается.

- Обратное движение по алгоритму.

  Отладка начинается там, где впервые встретился неправильный результат. Затем работа программы прослеживается (мысленно или при помощи тестов) в обратном порядке, пока не будет обнаружено место возможной ошибки.

5 Базовые структуры данных: примитивные типы; массивы; структуры

Примитивный тип — тип данных, предоставляемый языком программирования как базовая встроенная единица языка.

Примитивные типы в C++:

- boolean. Размер 1 байт, принимаемые значения true-false.
- char. Размер 1 байт, принимаемые значения -128-127. Используется для представления символьных данных.
- wchar_t. Размер 2-4 байта. Расширенный char-тип.
- int. Размер 4 байта. Целочисленный тип.
- short int. Размер 2 байта. «Ужатый» целочисленный тип.
- long int. Размер 8 байт. «Расширенный» целочисленный тип.
- float. Размер 4 байта. Вещественный тип данных.
- double. Размер 8 байт. «Расширенный» вещественный тип данных.
- long double. Размер 10 байт. «Сильно расширенный» вещественный тип данных.

Массив — тип или структура данных в виде набора компонентов (элементов массива) одного типа, расположенных в памяти непосредственно друг за другом. При этом доступ к отдельным элементам осуществляется с помощью индексации, то есть ссылки на массив с указанием индекса нужного элемента.

Массивы подразделяются на статические и динамические. Статические массивы занимают определенное количество памяти в зависимости от числа выделенных элементов на всем протяжении выполнения программы, а динамические способны изменять свой размер во время выполнения программы.

Кроме того, C++ дополнительно содержит шаблонные классы vector и array, которые являются высокоуровневыми реализациями динамических и статических массивов соответственно.

```
//code

//объявление статического массива
int qwerty[5] = {0, 1, 2, 3, 4};
```

```
//объявление динамического массива
int *qwerty = new int[10];

//объявление вектора
#include <vector>
std::vector<int> qwerty(10);
//при этом значения вектора будут инициализированы нулями

//объявление аррэя
#include <array>
std::array<int, 3> a2 = {1, 2, 3};

//end of code
```

Плюсы vector'а и array'я в том, что они позволяют использовать некоторые фишки, недоступные со стандартными массивами, например, копирование массива при объявлении, сравнивание массивов и т.п.

Массивы используются для хранения и итерационной обработки данных.

Структура (в C/C++) — совокупность переменных, объединенных одним именем, предоставляющая общепринятый способ хранения информации. При объявлении структуры создается шаблон, используемый для создания объектов структуры.Переменные, образующие структуру, называются полями структуры.

Структура — первый шаг к ООП, поскольку они позволяют создавать собственные шаблоны и работать с ними. Структуры, помимо примитивных, могут содержать и составные типы: массивы, структуры.

```
//code
struct mark //структура оценок
{
    int math; //за математику
    int physic; //за физику
    int geography; //за георгафию
};

struct student //создаем структуру
{
    char *name; //имя студента
```

```
    int age; //возраст
    int group; //класс
    mark winter; //оценки за зимнюю контрольную
};

int main() //работаем с нашими структурами
{
    student Stark;
    Stark.name = "Арья"; //обращение к полю
    Stark.age = 16; //осуществляется через точку
    Stark.group = 10;
    Stark.winter.math = 5; //структура в структуре
    Stark.winter.physic = 4; //why not?
    Stark.winter.geography = 5;
    return 0;
}
//end of code
```

6 Базовые структуры данных: строки и операции над строками

Строки состоят из символов.

Символ — элементарная единица, некий набор которых несет определенный смысл.

В C++ существует специальный символьный тип — char.

Строки в C++ представляются в виде массивов символов, завершающихся знаком конца строки \0 — нуль-терминатором.

Символьные строки состоят из набора символьных констант, заключенных в кавычки. При объявлении строкового массива требуется учитывать наличие в конце строки нуль-терминатора.

```
//code
char string[16] = "Мы сдали матан!"
std::cout << string;
//end of code
```

Разберем подробнее, что происходит при выполнении данного участка кода:

1) Создается массив char длиною 16 элементов. Кстати, если добавить еще один символ в строку, компилятор выкинет ошибку, пожаловавшись на «слишком длинный массив».

2) Элементам массива соответственно присваиваются значения символов строки.

3) Указатель на первый элемент массива «скармливается» cout'у. Cout понимает, что перед ним массив char'ов и пускается в последовательный вывод элементов этого массива до тех пор, пока не наткнется на нуль-терминатор. Наткнувшись на него, cout понимает, что массив закончился, и прекращает вывод. Кстати, подобное ограничение работает даже в том случае, когда после нуль-терминатора имеются еще ячейки массива. Таким образом, вывод кода

```
//code
char string[16] = "Мы сдали матан!"
string[7] = '\0';
std::cout << string;
//end of code
```

будет таким: «Мы сда».

При инициализации строки не обязательно указывать её размер — компилятор сделает это автоматически.

Считывание строк с клавиатуры можно осуществить несколькими способами:

1) С помощью функции gets(string) — считает в переменную string все символы, введенные до нажатия клавиши Enter.

2) С помощью std::cin. Учитывайте, что cin читает только первое слово и считает его за первую строку. Остальные слова остаются во входной очереди.

3) С помощью функции cin.get(string, 16). Учитывайте, что она оставляет символ конца строки во входной очереди. Чтобы считать его, используйте cin.get() без параметров.

4) С помощью функции cin.getline(string, 16). Её поведение аналогично предыдущей за исключением того, что она считывает в том числе и символ конца строки.

Функций для работы со строками много. Перечислим лишь основные:

- strlen(имя_строки) — определяет длину указанной строки, без учёта нуль-символа.
- strcpy(s1,s2) — выполняет побайтное копирование символов из строки s2 в строку s1.
- strcat(s1,s2) объединяет строку s2 со строкой s1. Результат сохраняется в s1.
- strcmp(s1,s2) сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1>s2, <0 — если s1<s2. Регистр учитывается.
- stricmp(s1,s2) сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1>s2, <0 — если s1<s2. Регистр не учитывается.

Шаблонный класс string.

Реализует строки как классы. Описан в заголовочном файле <string>.

Данный класс здорово облегчает жизнь программисту:

```
//code
#include <iostream>
#include <string>

int main()
{
```

```
    std::string str1 = "Darth Vader";
    std::string str2;

    str2 = str1; //копируем значение str1 в str2

    if (str1 == str2) //сравниваем строки
        std::cout << "Строки равны";

    std::string str3 = str1 + " " + srt2; //конкатенация.
}
//end of code
```

И так далее. Таким образом, класс string является мощным инструментом и здорово облегчает жизнь программисту.

7 Представление данных в памяти компьютера: биты, байты, слова; представление символьных данных

Любые данные в памяти компьютера представляются в виде последовательности нулей и единиц. Такая последовательность называется **кодом**. Последовательность именно нулей и единиц была выбрана потому, что её реализация максимально проста: есть сигнал — нет сигнала. Но, кстати, это не самое производительное решение. Наибольшей плотностью записи информации обладает система с основанием e. Таким образом, из целочисленных систем наибольшей плотностью записи информации обладает троичная система. Теоретически доказано, что троичный компьютер быстрее двоичного примерно в полтора раза (неожиданно, не правда ли?). Однако реализация такой системы очень сложна. В мире был создан всего один такой компьютер — Сетунь-70, разработанный научным сотрудником МГУ Брусенцовым Н.П. на основе троичных ферритодиодных ячеек собственной разработки.

Бит — двоичное число, принимающее значение 0 или 1. Наименьшая ячейка памяти компьютера.

Машинное слово — это упорядоченное множество двоичных разрядов, используемое для хранения команд программы и обрабатываемых данных. Машинное слово имеет тот же размер, что и разрядность процессора.

Байт — машинное слово минимальной размерности, адресуемое в процессе обработки данных.

Размеры байтов и машинных слов были введены корпорацией IBM в 1970-х годах:

- Байт — 8 бит.
- H (полуслово) — 16 бит, последний бит всегда 0.
- W (слово) — 32 бита, последние два бита 00.
- DW (двойное слово) — 64 бита, последние три бита 000.

Все современные процессоры являются 64-х битными. Данную архитектуру впервые реализовала в своих процессорах компания AMD в 2003 году, поэтому её чаще всего называют amd64 или x86-64, подразумевая обратную совместимость с 32-хбитной архитектурой x86, которую впервые реализовала Intel в процессорах 80?86 серии.

Представление текстовой информации:

Представление и обработка текстовой информации основана в C++ на базовом типе char, байт которого может хранить либо сам символ текста, либо его двоичное представление.

Для подобного представления придуманы **кодировки**, в которых каждый символ кодируется определенной последовательностью битов.

Международным стандартом в этом смысле стала кодировка ASCII. Сначала она была семибитовой и кодировала 128 символов, часть которых составляют буквы латинского алфавита, знаки препинания и спецсимволы. Однако затем она была расширена до 8 бит (256 символов); первые 128 символов кодируются так же, как и в семибитовой ASCII, а оставшиеся 128 символов могут быть использованы для регионального компонента кодировки.

Понятно, что 256 символами невозможно охватить все языки. Поэтому были разработаны другие кодировки, которые (конечно же!) несовместимы друг с другом. Однако сейчас развивается кодировка UTF-8 (Unicode Transformation Format,) (и её «дочки» UTF-16, UTF-32). Суть данной кодировки в том, что она охватывает все символы всех языков мира, символы вымерших языков, математические и музыкальные символы, смайлы и даже клинопись и алфавит Брайля!

Алгоритм кодирования в UTF-8 стандартизован и состоит из 3-х пунктов:

1) Определить количество байт, требуемых для кодирования данного символа.

2) Подготовить старшие биты первого байта.

3) Заполнить оставшиеся биты всех последующих байтов.

Суть кодирования в том, что первые n бит содержат информацию о региональной кодировке и числе байтов, которым кодируется один символ. К примеру, если первый бит равен нулю, то кодировка полностью аналогична ASCII. Если же он равен единице, то подключается еще один байт, содержащий региональные коды. Аналогично со следующий битом. Завершающим битом первого байта должен быть бит-терминатор, означающий конец кода размера. Затем идут значащие биты, при этом каждый новый байт начинается с префикса 10, означающего продолжение.

Пример кодировки одного символа, где «x» — значащий бит.

```
(1 байт)  0aaa aaaa
(2 байта) 110x xxxx 10xx xxxx
(3 байта) 1110 xxxx 10xx xxxx 10xx xxxx
(4 байта) 1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx
(5 байт)  1111 10xx 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx
(6 байт)  1111 110x 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx
```

Сейчас UTF-8 признана международным форматом кодировки, на который перешли все современные системы. В операционных системах семейств Linux

и BSD, Mac OS, Android, и так далее Unicode используется по умолчанию. Единственным «динозавром», по прежнему использующий кодировку собственного производства CP-1251 остаётся Windows, но, к счастью, и она начинает помаленьку переезжать на UTF-8.

8 Представление числовых данных и системы счисления

Кодирование числовой информации:

Числа в памяти компьютера хранятся в двоичной системе счисления. При этом отрицательные числа хранятся при помощи **дополнительного кода** — такого представления двоичного числа, при котором выполняются следующие операции:

1) добавим слева еще одну цифру — знак числа, принимающую всего два значения: 0 — плюс, 1 — минус;

2) положительные числа представляются обычным образом;

3) каждая цифра отрицательного числа заменяется на дополнение ее до n-1, где n - основание системы счисления (для десятичной системы - это дополнение до 9, то есть цифра, которая в сумме с исходной дает 9);

4) к полученному числу добавляется 1.

Таким образом, в двоичном двухбайтном целом (short int) положительные числа занимают диапазон [0x0000, 0x7FFF] ([0, 32767]), а отрицательные — [0x8001-0xFFFF] ([-32767, -1]). При этом число 0x8000 не определено (-0).

Такой финт позволяет нам складывать числа в дополнительном коде по правилам сложения чисел без знака и получать корректный результат, выраженный двоичным кодом.

Для представления чисел с плавающей точкой используется следующая запись:

Нулевой бит — знак. С 1 по 8 — порядок. С 9 по 31 — мантисса. (для 32-хбитного числа).

Нулевой бит — знак. 1, 2, 3 биты фиксированы. С 4 по 9 — порядок. Все остальное — мантисса. (для 64-хбитного числа).

Различают следующие типы чисел:

Нормализованное число: $\pm exp > 0$ и затем любой набор битов.

Ненормализованное число: ± 0 и затем любые символы не равные 0.

Ноль: ± 0 и нули.

∞: $\pm 1111...$ и затем нули.

NaN (Not a Number): $\pm 111...1$ и затем ненулевые символы.

Системы счисления:

Для человека самой понятной системой счисления навсегда останется 10-чная, так как именно она имеет основание, равное числу пальцев на руке человека. Компьютерам, по понятным причинам, понятнее двоичная систе-

ма счисления. В качестве компромисса между компьютером и человеком используется 16-теричная система счисления, которая объединяет в себе легкость перевода в двоичную и достаточную легкость для чтения и понимания человеком. Кроме того, в шестнадцатеричной системе счисления легко представлять данные в машинном виде: 1 байт = 2 цифрам.

9 Обзор операционных систем: роль и задачи операционных систем; простое управление файлами

Операционная система — комплекс взаимосвязанных программ, предназначенных для управления ресурсами вычислительного устройства и организации взаимодействия с пользователем.

Функции операционных систем:

1) Исполнение запросов программ (ввод и вывод данных, запуск и остановка других программ, выделение и освобождение дополнительной памяти и др.).

2) Загрузка программ в оперативную память и их выполнение.

3) Стандартизованный доступ к периферийным устройствам (устройства ввода-вывода).

4) Управление оперативной памятью (распределение между процессами, организация виртуальной памяти).

5) Управление доступом к данным на энергонезависимых носителях (таких как жёсткий диск, оптические диски и др.), организованным в той или иной файловой системе.

6) Обеспечение пользовательского интерфейса (управление вводом-выводом).

7) Сохранение информации об ошибках системы.

Дополнительные функции:

1) Параллельное или псевдопараллельное выполнение задач (многозадачность).

2) Эффективное распределение ресурсов вычислительной системы между процессами.

3) Разграничение доступа различных процессов к ресурсам.

4) Организация надёжных вычислений (невозможности одного вычислительного процесса намеренно или по ошибке повлиять на вычисления в другом процессе), основана на разграничении доступа к ресурсам.

5) Взаимодействие между процессами: обмен данными, взаимная синхронизация.

6) Защита самой системы, а также пользовательских данных и программ от действий пользователей (злонамеренных или по незнанию) или приложений.

7) Многопользовательский режим работы и разграничение прав доступа.

Роль операционных систем:

1) Универсальный механизм сохранения данных.

2) Распределение полномочий.

3) Управление процессами выполнения отдельных программ.

Первой действительно полноценной операционной системой была UNIX, разработанная в 1969 году Кеном Томпсоном, Деннисом Ритчи и Брайаном Керниганом. Существовавшие до этого момента решения были либо плохо масштабируемы (затруднена функция переноса системы), либо сложны, и, конечно же, полностью несовместимы между собой.

Одним из преимуществ новой ОС была принципиально новая и простая метафорика: всего два ключевых понятия: вычислительный процесс и файл.

Одной из важнейших функций операционной системы является управление файлами.

Файл — именованная область данных на носителе информации.

Для упорядоченной работы с файлами используются **файловые системы** — порядки, определяющие способ организации, хранения и именования данных на носителях информации. Также она связывает носитель и API для доступа к файлам. Файловая система не является частью ОС, она «лежит» на диске и требует драйверов для обращения к себе.

Свойства файла:

- Имя файла;
- Расширение файла;
- Основные атрибуты (владельцы Windows могут вспомнить галочку «скрытый»);
- Время (создания, изменения, последнего обращения);
- Владелец и группа;
- Права доступа (rwxr-xr- -).

Особенности реализации: в UNIX-подобных операционных системах процессы и устройства также представляются в виде файлов.

Буферизация — метод организации обмена, который подразумевает использование буфера для временного хранения данных. Бывает прозрачная, когда устройства не подозревают о существовании буфера между ними (например, операция кэширования перед записью на диск) и непрозрачная, когда сторонам для обмена требуются знания о буфере.

10 Введение в распределенные вычисления: предпосылки возникновения и история сетей и Интернета

Распределенное вычисление — вычисление какой-либо сложной программы посредством её разбиения на несколько малых кусков и их вычисление на нескольких разных машинах.

Особенностью распределенных многопроцессорных вычислительных систем является легкая масштабируемость, в отличии от локальных компьютеров.

Распределенная ОС, динамически и автоматически распределяя работы по различным машинам системы, заставляет набор сетевых машин обрабатывать информацию параллельно. При этом каждый компьютер в сети обладает собственной операционной системой, самостоятельно управляющей процессами и памятью на данной машине, однако они все объединены коммуникационными протоколами для организации взаимодействия процессов, выполняющихся на разных компьютерах сети. В таком случае каждая ОС каждого компьютера называется сетевой ОС.

Необходимость в сетях возникла еще в 50-х годах прошлого века, однако связано оно было не с распределенными вычислениями и не с вычислениями вообще. В 1958 году министерство обороны США приняло решение создать систему раннего оповещения о ракетной атаке со стороны СССР. Так как наблюдательные пункты были разбросаны по всей стране, возникла необходимость в сети, способной быстро передавать информацию. В 1962 году Джордж Ликлайдер из Массачусетского университета выступил с серией заметок о социальном взаимодействии при помощи компьютерных сетей. В этом же году его пригласили на должность первого руководителя исследовательского компьютерного проекта при Министерстве обороны США.

Так в конце 1969 года увидела свет первая компьютерная сеть APRANet, состоящая всего из 4-х компьютеров. В основе данной сети лежала идея **пакетной коммутации**, предложенная еще Полом Бэреном в середине 50-х годов. В течении следующих пяти лет к APRANet были подключены еще несколько сотен компьютеров.

Параллельно развивались и другие проекты сетей, однако препятствием между их совместной работой было то, что все они работали по-разному. Был необходим единый протокол, который бы позволил общаться разным компьютерам и сетям совместно. И в 1983 году был разработан подобный протокол, получивший название TCP/IP. Машины APRANet были переведены на него в том же году. Так был выработан стандарт, в соответствии с которым до сих

пор развивается сеть Интернет.

К возникшей сети затем подключалось все больше и больше компьютеров и в результате мы имеем то, что имеем — сеть Интернет.

11 Человеко-машинное взаимодействие: введение в вопросы проектирования

Интерфейс (чаще всего понимается пользовательский интерфейс) — совокупность средств и методов, с помощью которых пользователь взаимодействует с компьютером. Интерфейс может быть однонаправленным — вывод информации или двунаправленным — возможно обратное взаимодействие.

Юзабилити — субъективная оценка эффективности и удобства использования какого-либо интерфейса. Важными характеристиками, повышающими юзабилити, являются понятность, удобство и дружественность к пользователю.

Интерфейс предполагает использование различных систем ввода и вывода для взаимодействия с ним. Пробежимся кратко по истории интерфейса:

1) Ламповые пульты и перфоленты для вывода и перфокарты для ввода — самый старый и самый неудобный интерфейс. Но он выполнял свое главное призвание — обеспечивал человеко-машинное взаимодействие.

2) Дисплеи и устройства печати — с появлением этих устройств стало возможным выводить на экран образы (а, как известно, большая часть людей лучше воспринимает образы, а не текст). Оттуда же пошло правило «машина для человека».

Следует различать 2 понятия: UX (User Experience Design), который представляет собою дизайн «чтобы было удобно» и UI (User Interface Design) — более узкое понятие, включающее в себя дизайн элементов управления, кнопок и т.п. Грубо говоря, UX — это расположение некрасивых элементов интерфейса таким образом, чтобы пользователю было удобно его использовать и UI — приведение данных элементов интерфейса в красивый вид, как то: раскраска, наложение теней, выравнивание.

Основные правила создания интерфейса:

1) Организованность элементов интерфейса. Это означает, что они должны быть логически структурированы и взаимосвязаны.

2) Группировка элементов интерфейса. Подразумевает объединение в группы логически связанных элементов (меню, формы).

3) Выравнивание элементов интерфейса. Сложно представить, что плохо выровненный интерфейс может быть для кого-то удобным!

4) Единый стиль элементов интерфейса. Стилевое оформление играет не последнюю роль, ведь именно оно сохраняется в памяти пользователя.

5) Наличие свободного пространства. Это позволяет разграничивать информационные блоки, сосредотачивая внимание на чем-то одном.

Нужно осознавать, что 99% успешности программы — это интерфейс, так как пользователь не видит «потрохов» программы и собирается взаимодействовать с вашей программой, а не копаться в её внутренностях. Поэтому дизайну также нужно уделить немало времени.

Хочется дополнительно отметить, что существует такой вид интерфейса, как **командная строка** или **терминал**. Данный интерфейс имеет не очень высокое юзабилити, интуитивно не понятен и не удобен в использовании для новичков. Однако, разобравшись в нем, можно признать, что он гораздо удобнее графического, поскольку все настройки и функции лежат на поверхности. Кроме того, он имеет гораздо больший потенциал для совершения неких специальных действий, например, множественного копирования или переименования.

12 Методология разработки программного обеспечения: основные понятия и принципы проектирования; структурная декомпозиция; стратегии тестирования и отладки; разработка сценариев тестирования (test cases)

Методология — это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Методология — это реализация стандарта. Сами стандарты лишь говорят о том, что должно быть, оставляя свободу выбора и адаптации.

Конкретные вещи реализуется через выбранную методологию. Именно она определяет, как будет выполняться разработка. Существует много успешных методологий создания программного обеспечения. Выбор конкретной методологии зависит от размера команды, от специфики и сложности проекта, от стабильности и зрелости процессов в компании и от личных качеств сотрудников.

Методологии представляют собой ядро теории управления разработкой программного обеспечения. К существующей классификации в зависимости от используемой в ней модели жизненного цикла (водопадные и итерационные методологии) добавилась более общая классификация на прогнозируемые и адаптивные методологии.

Модель водопада — модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

Итеративная разработка — выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы. Проект при этом подходе в каждой фазе развития проходит повторяющийся цикл: Планирование - Реализация - Проверка - Оценка.

Прогнозируемые методологии фокусируются на детальном планировании будущего. Известны запланированные задачи и ресурсы на весь срок проекта. Команда с трудом реагирует на возможные изменения. План оптимизирован исходя из состава работ и существующих требований.

Адаптивные методологии нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения. Когда меняются требования, команда разработчиков тоже меняется. Команда, участвующая в адаптивной разработке, с трудом может предсказать будущее проекта. Существует точный план лишь на ближайшее время. Более удаленные во времени планы существуют лишь как декларации о целях проекта, ожидаемых затратах и результатах.

В основе любой методологии лежит **структурное программирование** — постепенная декомпозиция задачи и реализация её решения с использованием заранее ограниченного числа конструкций (while, if, разделение на функции и так далее).

Главный принцип любой методологии — принцип «разделяй и властвуй», то есть принцип **структурной декомпозиции**.

Этапы разработки любой сложной системы:

1) Декомпозиция, то есть разделение задачи на подзадачи. Оценка возможности упрощения тех или иных подзадач.

2) Проверка поддержки данной декомпозиции используемыми технологиями.

3) Оценка сложности каждой подсистемы.

4) Сообщение заказчику о результатах декомпозиции, при этом планируемое время разработки умножаем на два (а цену на три, хе-хе).

5) Если заказчик дает добро, принимаемся за разработку.

Таки да, исполнение всех этих пунктов не бесплатное! Это называется системный анализ и такое стоит немалых денег.

Язык UML — язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур. Он был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

Стратегия тестирования — это план проведения работ по тестированию системы или её модуля, учитывающий специфику функциональности и зависимости с другими компонентами системы и платформы. Стратегия определяет типы тестов, которые нужно выполнять для данного функционала системы, включает описание необходимых подходов с точки зрения целей тестирования и может задавать описания или требования к необходимым для проведения тестирования инструментам и инфраструктуре.

Несколько правил тестирования:

1) Знаем, что должна выдать на программа на данном тесте до запуска теста;

2) Хороший тест — тест, который сломал программу.

3) Каждый цикл проверять минимум 3 раза: без захода в цикл, с единич-

ным заходом, с множественным заходом.

4) Лучший способ отладки — чтение программы. Еще более лучший способ отладки — описание действия программы «на бумажке» или объяснение её кому-то постороннему.

Отсюда вытекают **аксиомы программирования**:

1) В каждой программе есть ошибка.

2) Если в программе нет ошибок, то ошибка в алгоритме.

3) Если нет ошибок ни в программе, ни в алгоритме, то такая программа никому нафиг не нужна!

Итоги:

1) Тесты нужно писать до программы (test first)!

2) При написании программы нужно использовать принципы структурного программирования (и никаких goto!).

3) Нужно заранее готовиться к отладочной работе (расставить контрольные точки и понять, что будет делать программа при этом и только после этого нажимать кнопку «Пуск»).

13 Среды разработки; инструменты тестирования и отладки

IDE (интегрированная среда разработки) — комплекс программных средств, используемых программистами для разработки программного обеспечения. Включают в себя (минимум):

- Текстовый редактор;
- Компилятор/интерпретатор;
- Средства автоматизации сборки;
- Отладчик (дебаггер).

Однако поистине хорошая среда разработки обязана включать в себя (или быть расширяемой до):

- Систему управления контролем версий;
- Средства облегчения использования объектно-ориентированного программирования (браузер классов, инспектор объектов, диаграмму иерархии классов);
- Средства разработки графических интерфейсов;
- Возможность написания собственных плагинов.

Отладка — этап в разработке программы, состоящий в выявлении и устранении программных ошибок, факт наличия которых уже установлен.

Тестирование — процесс выполнения данной программы на некоем наборе тестов, для которых заранее известен результат выполнения.

Отладку можно представить в виде многократного повторения трех процессов: тестирования, поиска ошибки и редактирования кода программы для её исправления.

Главным инструментом для тестирования и отладки программы является дебаггер. Он позволяет выполнять части кода, расставлять точки останова, менять значения переменных в процессе выполнения программы и так далее.

Точка останова (Break point) — некая точка, расположенная в коде программы. При достижении данной точки выполнение программы будет прервано и управление передано отладчику. А это значит, что теперь мы можем посмотреть все значения локальных переменных и понять, где ошибка. И, конечно же, изменить их.

Трассировка — построчное выполнение кода, при котором брэкпоинты как бы устанавливаются на каждой строчке. При этом доступен функционал вхождения/игнорирования if'ов.

Таким образом, отладчик позволяет находить ошибки достаточно быстро.

14 Социальный контекст компьютинга: история компьютинга и компьютеров; эволюция идей и компьютеров; социальный эффект компьютеров и Интернета; профессионализм, кодекс этики и ответственное поведение; авторские права, интеллектуальная собственность и компьютерное пиратство

Краткий курс истории компьютеров:

В 1812 году английский математик и экономист **Чарльз Бэббидж** начал работу над созданием так называемой «разностной машины», которая, по его задумке, могла не только производить арифметические действия, но и выполнять их по определенному алгоритму. К сожалению, данная машина так и не была закончена.

Однако 1842 году была опубликована рукопись «Очерк об аналитической машине Чарльза Бэббиджа», переведенная ученицей и последовательницей Беббиджа — Адой Лавлейс. Она в том числе составила несколько программ для этой аналитической машины. Поэтому леди Лавлейс считается первым программистом мира. Язык Ада, кстати, был назван в её честь.

После Бэббиджа значительный вклад внес американский изобретатель Г. Холлерит, впервые построивший ручной перфоратор и табулятор — машину для сортировки перфокарт.

Большим толчком к развитию электронно-вычислительной техники послужило создание в 1917 году катодного реле (триггера) М.А. Бонч-Бруевичем.

Первым создателем автоматической вычислительной машины считается немецкий учёный К. Цузе, построивший релейные машины Z1, Z2, Z3, Z4, Z5, однако наиболее известна Z3, так как именно на нее была впервые опубликована документация.

Независимо от Цузе в 1943 году Г. Эйкен и Т. Уотсон собрали собственную релейную машину Mark-I.

Первой ЭВМ считается ENIAC, которая выполняла операции сложения и умножения на три порядка быстрее, чем это делали релейные машины. Она была представлена в 1946 году.

Революцию совершили идеи американского математика фон Неймана. До него вычислительные машины основывались на гарвардской архитектуре, которая подразумевала разделенную командную память и память физических данных. Фон Нейманом же была предложена архитектура, в которой командная память и память данных были собраны в одном устройстве. Данная архитектура называется фон-Неймановской и все современные компьютеры построены на ней.

Развитие ламповых ЭВМ началось с постройки серии ЭВМ UNIAC, которые работали с частотой 2.5 МГц и содержали до 5000 ламп.

Первой советской вычислительной машиной считается БЭСМ, которая была построена в Институте точной математики и вычислительной техники под руководством С.А. Лебедева. Это произошло в 1952 году. С 1956 года начался серийный выпуск БЭСМ-2.

В 1955 году в США было объявлено о создании транзисторного компьютера. В течении 10 лет были созданы десятки ЭВМ на транзисторах.

В 1964 году IBM создает шесть машин модели IBM-360, которые были построены на интегральных схемах. Это ознаменовало начало эпохи компьютеров третьего поколения. Таких машин было выпущено уже более 33 тысяч только корпорацией IBM. По всему миру строятся машины, подобные данной.

Примерно в это же время появляются визуальные устройства ввода-вывода — графические дисплеи.

Четвертое поколение ЭВМ связано с развитием микропроцессорной техники. В 1971 году Intel выпускает микросхему Intel-4004, родоначальницу современного семейства x86 процессоров.

В 1972 году появился первый восьмибитный процессор Intel-8008, разрядность которого позволяла кодировать не только цифры, но и буквы алфавита и специальные символы. Таким образом, мировая индустрия стояла на пороге создания персональных компьютеров.

Первые такие ПК базировались на процессорах Intel-8080 и выпускались во второй половине 70-х годов.

Однако первым ПК считается компьютер Apple-I, собранный Стивом Джобсом и Стивом Возняком в 1976 году. Он не имел ни клавиатуры, ни корпуса, однако мог считаться полноценным компьютером. Затем появился Apple-II, который принес компании Apple мировую известность и более двух миллиардов долларов.

К 1980 году становится очевидным успех идеи ПК. Появляется множество машин, самыми известными из которых стали IBM-PC (завоевал наибольшую популярность), Apple Macintosh, Amiga (нет, не Amigo), ZX Spectrum и так далее.

К тому моменту началось постоянное наращивание производственных мощностей, частот и размеров памяти, что привело повсеместному распространению ПК.

О социальном эффекте даже писать не буду. Все знают, каков он и с чем его едят.

Трудно оценить профессионализм программиста, как и результат его работы, поскольку алгоритмов решения задачи может быть несколько и каждый будет иметь плюсы и минусы. Например, результат работы программиста можно оценить по метрике сложности программы.

Метрика сложности программы содержит:

1) Количество строк кода;

2) Число функций;

3) Число баз данных и таблиц;

4) Число окошек;

5) Число ветвлений;

Но, конечно, всё в этом мире субъективно. Однако есть так называемый кодекс этики программистов, который подразумевает следующие признаки профессионального программиста:

1) Профессионал смотрит в будущее своей программы, оценивает возможности развития, поддержки.

2) Работа профессионала кому-то нужна, кроме него самого. Профессиональный программист — не студент, который пишет программки для души, он нацеливает свой продукт на какую-то определенную аудиторию.

3) Профессионал охотно делится своими знаниями с соседями.

4) Профессионал состоит в какой либо профессиональной организации (к примеру, IEEE).

5) Профессионал всегда в ответе за свою программу.

В современном мире проблема авторского права стоит довольно остро. С появлением интернета и торрент-сетей развернулась настоящая баталия между сторонниками копирайта — соблюдения авторского права и сторонниками копилефта, которые призывают отказаться от авторского права. Первые напирают на то, что работа автора должна быть оплачена; вторые — на то, что в цифровую эпоху копирование не стоит ничего. Данная война длится уже второй десяток лет и пока перевеса нет ни на одной стороне.

Причем здесь программист? А притом, что его напрямую касается эта война, ведь компьютерная программа считается интеллектуальной собственностью и точно так же легко распространяема. В связи с этим программисту требуется знать несколько правил:

1) Написанная вами «для себя» программа принадлежит вам и вы можете распоряжаться ею, как душе угодно (но только если она не нарушает прав и свобод других людей).

2) Авторские права на программу, написанную по заказу кого-либо принадлежат, чаще всего, не вам, а этому кому-то, однако у вас есть право быть упомянутым как разработчик этой программы.

3) Лицензии для частного и коммерческого использования отличаются друг от друга. Программа, бесплатная для частного пользователя, скорее всего будет платной для компании (сразу же вспоминаем Skype).

4) Существует множество различных лицензий, под которыми вы можете выпускать свою интеллектуальную собственность.

Разберем подробнее последний пункт.

Все программное обеспечение можно разделить по признаку свободы на 2 больших группы: проприетарное и открытое.

Проприетарное программное обеспечение — программное обеспечение, являющееся частной собственностью правообладателя, который сохраняет за собой полную монополию на данное ПО. Напрямую с понятием коммерческого ПО данное понятие не связано, однако они близки по смыслу.

В чем его суть? Вы — автор, царь и бог данного ПО. Вы можете продавать его по бешеным ценам или сделать бесплатным. Вы можете выпустить обновление, не советуясь ни с кем. Вы можете прекратить поддержку данного ПО. Вы можете запретить распространять его кому-либо, кроме вас самих (что чаще всего и делают). Вы можете делать с ним *все, что угодно*, и пользователи вашего ПО будут вынуждены либо пользоваться им, либо искать аналоги.

Такой подход позволяет вам зарабатывать деньги, причем большие деньги. Билл Гейтс сделал свои миллиарды, просто продавая каждую копию операционной системы Windows. Такая копия стоит ровно столько, сколько требуется копьютеру энергии для записи ПО на диск плюс сам диск. С другой стороны, на разработку Windows было потрачено много сил и средств, и понятно, что они должны окупаться.

В противовес закрытому программному обеспечению существует **открытое программное обеспечение**. Главным признаком такого ПО является открытый исходный код и свобода копирования. То есть любой может изменить код вашей программы так, как хочется ему, или использовать вашу программу не переводя ни цента на ваш счет. Однако, накладываются и ограничения: вы не имеете права продавать программы с открытым исходным кодом

(хотя некоторые лицензии позволяют продавать ПО, созданное при помощи открытого) и выпускать его в виде проприетарного. Также некоторые лицензии обязуют вас указывать автора программы (поскольку авторские права могут быть наложены на open-source код).

С одной стороны, open source позволяет вам использовать программы, не платя разработчикам, с другой — проект, выпущенный под свободной лицензией рискует не окупиться, ведь никто не может заставить пользователей платить за использование открытого проекта. Все весьма неоднозначно, сторонники и хорошие аргументы имеются и у той, и у другой стороны.

Однако есть еще один класс людей, которые реализуют главную доктрину свободного программного обеспечения: «копия не стоит ничего» на проприетарном ПО и не только ПО. Сеть peer-to-peer и протокол BitTorrent позволяют им неограниченно обмениваться информацией друг с другом, плодя множественные копии. Отсутствие единого центра в сетях подобного ранга затрудняет борьбу с данным видом распространения ПО, которое получило название «компьютерное пиратство».

Однако даже у компьютерных пиратов есть аргументы за свое поведение. К ним можно отнести возможность покупки некачественного ПО (не реализующего необходимых функций, забагованного или же неудобного к использованию), отсутствие денег на приобретение дорогостоящего ПО у некоторых классов общества (50% аудитории торрентов — люди младше 25 лет) и так далее.

В этих вопросах все упирается в человеческую мораль, а, так как она неоднозначна, то неоднозначно и отношение людей к данному вопросу.

15 Объектно-ориентированное программирование: объектно-ориентированное проектирование, инкапсуляция и скрытие информации; разделение интерфейса и реализации; классы, наследники и наследование; полиморфизм; иерархии классов

Объектно-ориентированное программирование — метод программирования, при использовании которого главными элементами программ являются объекты. В языках программирования понятие объекта реализовано как совокупность свойств (структур данных, характерных для данного объекта), методов их обработки (подпрограмм изменения их свойств) и событий, на которые данный объект может реагировать и которые приводят, как правило, к изменению свойств объекта.

Три слона ООП:

1) **Инкапсуляция** — объединение данных и свойственных им процедур обработки в одном объекте;

2) **Наследование** — свойство, предусматривающее создание новых классов на базе существующих таким образом, чтобы класс-потомок имел все свойства класса-родителя.

3) **Полиморфизм** — свойство, позволяющее задать общие действия для класса. В более общем смысле полиморфизм можно интерпретировать как «один интерфейс, множество действий», что означает возможность создания общего интерфейса для близких по смыслу действий. Классический пример — перегрузка функций.

Эти три слона стоят на черепахе **Абстракции** — способе выделить набор общих характеристик объектов, исключая из рассмотрения те, которые общими не являются.

Фундаментальным понятием ООП является **Класс** — шаблон, на основе которого может быть создан конкретный программный объект, он описывает свойства и методы, определяющие поведение объектов данного класса. Каждый конкретный объект, имеющий структуру этого класса, называется экземпляром класса.

Наследник — класс, наследуемый от класса-предка, с сохранением свойств класса-предка и приобретением новых, «личных» свойств.

```
//code
смотри main1.txt в этой же папке
//end of code
```

public-часть кода называется **интерфейсом** и данные в ней видны за пределами класса.

private-часть кода называется **реализацией** и за пределами класса не видна.

protected-часть кода никак не называется и видна только лишь наследникам.

Такое разделение создано, во-первых, для инкапсуляции, а во-вторых, для того, чтобы никто не мог чего-нибудь начудачить в private-данных. В области public не принято записывать данные. Место данных — в private. А в public пишутся лишь операции доступа. В таком случае, если создателем программы будут изменены какие-то данные, пользователи ничего не заметят. Это снижает ошибаемость.

Иерархии классов — классификация объектных типов, при которой объекты рассматриваются как реализации классов. Различные классы связываются отношениями типа «наследует», «расширяет», «является абстракцией», «описание интерфейса».

16 Основные вычислительные алгоритмы: алгоритмы поиска и сортировки (линейный и дихотомический поиск, сортировка вставкой и выбором наименьшего элемента)

Поиск — обработка некоего множества данных с целью выявления подмножества данных, соответствующих критериям поиска.

Все алгоритмы поиска делятся на поиск в упорядоченном множестве данных и поиск в неупорядоченном множестве данных.

Линейный поиск может осуществляться на любом множестве данных. Представляет собою простой перебор (брутфорс) значений множества и их сравнивание с искомым значением. Имеет сложность алгоритма $O(n)$, поэтому применяется лишь для небольших массивов данных. Однако плюсами данного поиска являются отсутствие требований дополнительной памяти и простейшая реализация.

Дихотомический, он же бинарный, поиск — поиск методом деления отрезка поиска пополам и проверки, в какую из половин входит искомое значение. Имеет сложность алгоритма $\log n + 1$, но относительно сложную реализацию (известна история, когда автор бинарного поиска в своей книге о бинарном поиске допустил ошибку в реализации алгоритма), требует дополнительной памяти (в рекурсивной реализации) и сортированного массива данных, в котором поиск будет осуществляться.

Существуют также интерполяционный поиск и поиск с помощью хеш-таблиц.

Сортировка — последовательное расположение или разбиение на группы данных в зависимости от выбранного критерия.

Свойства сортировок:

1) Устойчивость — устойчивая сортировка не меняет положения одинаковых элементов.

2) Естественность поведения — эффективность метода при обработке частично отсортированного массива. Алгоритм ведет себя естественно, если учитывает это и работает быстрее.

3) Сложность (использование операции сравнения). Для сортировок, основанных на сравнениях, сложность алгоритма составляет минимум $n \log n$.

Виды сортировок можно прочитать в википедии.

Сортировка выбором наименьшего элемента

Шаги алгоритма:

1) Находим номер минимального значения в текущем списке;

2) Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции);

3) Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

Сложность алгоритма $O(n^2)$.

Сортировка вставками

Алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Вычислительная сложность - $O(n^2)$.

17 Основы программирования, основанного на событиях

Событийно-ориентированное программирование — парадигма программирования, в которой выполнение программы определяется событиями.

Событие в ООП — сообщение, которое возникает в различных точках исполняемого кода при выполнении определенный условий.

События предназначены для того, чтобы иметь возможность предусмотреть реакцию программного обеспечения. Для решения этой задачи создаются обработчики событий: как только программа попадает в заданное состояние, происходит событие, посылается сообщение, а обработчик перехватывает это сообщение. Событие, по сути, сообщает об изменении состояния некоего объекта. Наиболее наглядно события представлены в пользовательском интерфейсе, где на каждое действие пользователя выдается какой-либо ответ. В объектно-ориентированном анализе для описания динамического поведения объектов принято использовать модель состояний. Событие - это переход объекта из одного состояния в другое. Взаимодействие объектов также осуществляется при помощи событий: изменение состояния одного объекта приводит к изменению состояния другого объекта, а событие оказывается средством связи между объектами.

Выделяются четыре аспекта события:

- Метка — уникальный идентификатор события.
- Значение — текстовое сообщение о сути произошедшего.
- Предназначение — модель событий, которая принимает значение.
- Данные — данные, которые переносятся от одного объекта к другому.

Событийно-ориентированное программирование, как правило, применяется в следующих случаях:

- Построение пользовательских интерфейсов (в том числе графических);
- Создание серверных приложений;
- Моделирование сложных систем;
- Параллельные вычисления;
- Автоматические системы управления, SCADA;
- Программирование игр, в которых осуществляется управление множеством объектов.

18 Введение в компьютерную графику: использование простых графических API

Компьютерная графика — область деятельности, в которой компьютеры используются в качестве инструмента для создания изображений.

Пока мы будем рассматривать двумерную графику. Двумерная графика бывает трех типов:

Растровая — изображение задается множеством различных точек растра, которые называются **пикселями**;

Векторная — изображение задается в качестве математической функции, описывающей геометрический примитив. Такими функциями могут быть многочлены, кривые Безье и так далее;

Фрактальная — изображение строится из самого себя посредством **фракталов** — объектов, отдельные элементы которых наследуют свойства родительских структур. Сфера применения такого типа графики ограничена по понятным причинам.

Каждый из этих типов имеет свои плюсы и минусы:

Растровая графика:

+ Простая;

+ Таким образом можно изобразить любую фигуру любой сложности.

- Занимает много места;

- При масштабировании становятся видны пиксели, что делает её непригодной для шрифтов.

Векторная графика:

+ Малый размер занимаемой памяти;

+ Отсутствие артефактов при отрисовке.

- Обычные (не геометрически правильные) изображения очень трудно передать с помощью примитивов.

Фрактальная графика:

+ Построение сложных структур с помощью небольшого числа операций и при минимуме занимаемой памяти

- Лишь некоторые структуры можно описать с помощью фракталов.

Передача цвета.

Передача цвета осуществляется с помощью **цветовых моделей** — математических моделей описания цветов в виде кортежей чисел, называемых цветовыми компонентами или цветовыми координатами. Наиболее общеупотребительная цветовая модель — RGB, но используется также модель CMYK.

API — интерфейс прикладного программирования. Интерфейс — это то, с помощью чего что-то взаимодействует с чем-то (идеальное определение).

Существует множество графических библиотек, которые позволяют взаимодействовать с графической подсистемой операционной системы на любом уровне. Однако работать с низкоуровневым взаимодействием очень неудобно, на то оно и низкоуровневое. Поэтому были придуманы различные высокоуровневые графические API, один из которых — Qt.

Qt — фреймворк, кроссплатформенный инструментарий для разработки ПО на языке C++. Есть также возможность разрабатывать и на других языках.

Огромным преимуществом Qt является то, что он абстагирован от операционной системы и имеет свои реализации некоторых систем, что позволяет компилировать написанный на Qt код под любую ОС. Чаще всего править исходные коды не придется.

Как и любой фреймворк (ПО, облегчающее разработку и объединение разных компонентов большого проекта), Qt имеет очень много «плюшек», и возможность рисования интерфейсов — лишь одна из них. Но, так как тема билета все-таки графические API, мы не будем останавливаться на тех сторонах Qt, которые к делу не относятся.

Qt предоставляет огромные возможности для создания GUI — графического интерфейса пользователя. К примеру, класс QStyle, который «подстраивает» интерфейс программы под интерфейс операционной системы, используя её таблицу стилей. Таким образом, программа, запущенная под Windows будет иметь оформление строго с стиле Windows, а в Ubuntu — стиль Ubuntu. При этом, если вы смените стандартную схему оформления на другую, окно также сменит тему оформления (возможно, придется перезапустить программу).

QDialogButtonBox предоставляет стандартный интерфейс диалогового окна.

Для взаимодействия с пользователем используется механизм сигналов и слотов. Слот — это функция, которая может быть вызвана в процессе выполнения программы. Сигнал — функция, вызывающая функции-слоты, которые ассоциированы с ним посредством QObject::connect. К примеру, в коде:

```
#include <QtGui>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    QTextEdit textEdit;
```

```
    QPushButton quitButton("Quit");

    /* 10 */ QObject::connect(&quitButton, SIGNAL(clicked()),
                                          qApp, SLOT(quit()));

    /* 12 */ QVBoxLayout layout;
    /* 13 */ layout.addWidget(&textEdit);
    /* 14 */ layout.addWidget(&quitButton);

    QWidget window;
    /* 17 */ window.setLayout(&layout);

    window.show();

    return app.exec();
}
```

Создается окно, выводящее некое сообщение и кнопка Quit, являющаяся частью этого окна. С помощью Qobject::connect связывается сигнал clicked(), который передает нажатие кнопки и слот quit(), который завершает приложение.

Даже и не знаю, что еще написать. Буду благодарен, если подскажете.