1. **Стили программирования. Примеры разработки программ.**

Технологии программирования - совокупность методов и средств, разработки программ и правило применения этих методов и средств

1) Процедурное программирование – это программирование, в котором отражен фон Неймановской архитектуры компьютера. Все программы, написанные на данном языке, являют собой определенную последовательность команд, которые устанавливают некий алгоритм для разрешения того или иного пакета задач. Самой важной командой является операция присвоения, что предназначена для установления и корректировки содержимого в памяти компьютера.

-Сходящий(+)

-Восходящий

2)Функциональное - здесь программа разрабатывается с использованием функционального(аппликативного) стиля, состоит из совокупности определений функций. Функции в свою очередь представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции которая в свою очередь вызывает функции, входящие в ее определение и тд в соответствии с иерархией определений и структурой условных предложений. Здесь функции часто либо, прямо либо опосредованно вызывают сами себя. Каждый вызов фунции возвращает некоторое значение в вызывающую его функцию, вычисление которой после этого продолжается. Этот процесс повторяется до тех пор, пока не вычислится значение нужной функции. Первым языком является Lisp, современный - Haskell.

//Код - main = putStrLn "Hello World!"

//Запуск runhaskell ./hello.hs

3) Логическое програмирование(реляционное) - программа должна уметь работать по логическим построениям, программа в таких языках состовляет собой совокупность правил и целей. Процесс выполнения программы трактуется как процесс установления общезначимости логичекой формулы по правилам, установленным семантикой того или иного языка(Prolog)

//Задание prolog 2\_1: Запустите компилятор Tprolog. Создайте новый файл и наберите код программы, выводящий ответ на вопрос «Любит ли Мэри яблоки?» (ответ true или false). Код представлен ниже: domains a=symbol predicates likes (a,a) clauses likes (mary,apples). Перейдите в окно Dialog (меню Run) и введите запрос: likes(mary,apples) В результате в окне должен появиться ответ true

4 принципа ООП Абстрагирование Наследование Инкапсуляция Полиморфизм(переопределяет метод класса родителя) Они полиморфизились. Я инкапсулировался. Саня абстрагировал всю ситуацию. Все это наследовано от предков. (с)Конфуций 228 год до н.э.

1. **Структура программы. Пространство имен.**

Пространство имен помогают избегать конфликтов имен(функций, переменных (Пример, две разные библиотеки функций содержат функции с одинаковыми названиями, то следует объявить уникально простанство имен для функций и классов отдельной библиотеки, тогда можно либо при вызове определенных функций использовать префикс просртанства имен, либо объявить, что все функции по умолчанию будут вызываться из определнного пространства имен. Эта строка using namespace std - что все определенные ниже имена в программе будут относиться к пространству имен с именем std)).

1. **Переменные и константы. Основные типы данных. Преобразование типов.**

Const, int, char, double, float, bool

1. **Потоки ввода и вывода. Манипуляторы ввода-вывода.**

Манипуляторы ввода-вывода управляют форматом вводимого и выводимого значений. Это функции, которые вставляются между вводимыми и выводимыми значениями и тем самым изменяют состояние потока - #include - Таблица со значениями манипуляторов: Манипулятор | Описание dec - ввод-вывод в десятичной форме endl - перевод строки ends - вывод нулевого символа flush - перевод в поток содержимого буфера hex - ввод-вывод в шестнадцатиричной форме oct - ввод-вывод в восьмеричной форме resetioflags(long f) - сбрасывает флаги, указанные в f setbase(int base) - Устанавливает базу счисле­ния равной параметру base setfill(int ch) - Устанавливает символ заполнения равным ch setiosflags(long f) - Устанавливает флаги, указанные в f setprecision(int p) - Устанавливает число цифр после запятой setw(int w) - Устанавливает ширину поля равной w ws - Пропускает начальный символ-разделитель

1. **Операторы цикла. Примеры использования.**

For, while, do while

1. **Условные операторы. Условная операция. Примеры использования.**

If else

1. **Операторы перехода. Операции отношения. Примеры использования.**

break;

continue;

return [expression];

goto identifier;

|  |  |
| --- | --- |
| **Операторы отношения** | |
| **Оператор** | **Действие** |
| > | Больше чем |
| >= | Больше чем или равно |
| < | Меньше чем |
| <= | Меньше чем или равно |
| == | Равно |
| != | Не равно |

1. **Арифметические операции. Логические операции. Примеры использования.**

+-\*/%

|  |  |
| --- | --- |
| **Логические операторы** | |
| && | И |
| || | ИЛИ |
| ! | НЕ |

**9. Структуры. Вложенные структуры. Массив структур. Примеры использования.**

Struct A{}; A f[2];

**10. Перечисления. Примеры использования.**

Enum, с 0 начинается, всем целые числа, на 1 больше предыдущего, числа могут повторяться

**11. Функции. Область видимости и классы памяти.**

|  |  |  |  |
| --- | --- | --- | --- |
| Динамический класс памяти | | Статический класс памяти | |
| Автоматический | Регистровый | Локальный | Глобальный |
| auto | register | static | Extern |

* auto — [автоматическая](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F&action=edit&redlink=1) ([локальная](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F)). Автоматические переменные создаются при входе в функцию и уничтожаются при выходе из неё. Они видны только внутри функции или блока, в которых определены. В [C++11](https://ru.wikipedia.org/wiki/C%2B%2B0x) значение слова auto изменили.
* static — [статическая переменная](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F&action=edit&redlink=1) (локальная). Первое означает, что определяемый объект располагается по фиксированному адресу. Тем самым обеспечивается существование объекта с момента его определения до конца выполнения программы. Второе значение означает локальность. Объявленный со спецификатором static локален в одном программном модуле (то есть, недоступен из других модулей многомодульной программы) или в классе extern — [внешняя](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%BD%D0%B5%D1%88%D0%BD%D1%8F%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F&action=edit&redlink=1) ([глобальная](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D0%BE%D0%B1%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F)) переменная. Внешние переменные доступны везде, где описаны, а не только там, где определены. Использование ключевого слова extern позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. Для таких переменных связывание с адресом происходит на этапе компоновки.
* register — [регистровая переменная](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F&action=edit&redlink=1) (локальная). Это слово является всего лишь «рекомендацией» компилятору помещать часто используемую переменную в [регистры процессора](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D1%8B_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0) для ускорения программы.

**12. Перегруженные функции. Пример использования.**

Перегруженные функции - функции с одним именем, но с разным кол-ом аргументов. Их используют Когда несколько функций выполняют одинаковые действия и имеют одинаковый тип, но над разными типами.

Пример: i

nt summ(int &a, int &b){ return a + b; }

int summ(int &a, int &b, int &c){ return a + b + c; }

int summ(float &a, float &b){ return floor(a + b); }

**13. Способы передачи аргументов в функцию. Примеры.**

Существует три способа передачи параметра в ф-ф:

1)передача по значению

При передаче аргументов по значению внешний объект, который передается в качестве аргумента в функцию, не может быть изменен в этой функции

void square(int a, int b)

2)передача оп указателю

Если передавать параметры через указатель (так называемый С-стиль), код функции выглядит так:

mul5(int \*x){\*x = \*x \* 5;}

3)передача по ссылке

При передаче параметров по ссылке передается ссылка на объект, через которую мы можем манипулировать самим объектов, а не просто его значением

void square(int &a, int &b)

**14. Рекурсивные функции. Виды рекурсии. Примеры.**

Рекурсия на спуске/на подъеме (факториал)

int fact(int n){ if (n == 0 || n == 1) return 1; else return n \* fact(n - 1); }

**15. Одномерный и многомерный массивы. Инициализация массивов. Передача массивов в функцию.**

int ary[3][5][2];

трёхмерный массив, содержащий 3 матрицы, каждая из которых состоит из 5 строк по 2 значения типа int в каждой строке.

Если функция принимает в качестве параметра массив, то фактически в эту функцию передается указатель на первый элемент массива. То есть как и в случае с указателями нам доступен адрес, по которому мы можем менять значения. Поэтому следующие объявления функции будут по сути равноценны:

|  |  |
| --- | --- |
| 1  2 | void print(int numbers[]);  void print(int \*numbers); |

**16. Строки на основе массива char. Массивы строк. Примеры использования.**

**17. Строки на основе класса string. Методы класса string. Примеры использования.**

+, ==,=, size, empty getline, at()/[]

**18. Указатели и массивы. Примеры использования.**

Имя массива по сути является адресом его первого элемента

адрес второго элемента будет представлять выражение a+1, а его значение - \*(a+1).

a = &b;         // так тоже сделать нельзя

**19. Указатели и функции. Примеры использования.**

double(\*pfunc)(double);    // переменная указатель на функцию

pfunc = area;

cout << "для круга радиуса " << r << " площадь = "

             << (\*pfunc)(r) << endl;

**20. Указатели и строки. Примеры использования.**

char \*s

s--;

//перейти к предыдущему символу строки

//(на sizeof(char) байт, необязательно один)

2. Указателю можно присваивать адрес объекта унарной операцией "&":

int \*px; int x,y;

px=&x;

//теперь px показывает на ячейку памяти со

// значением x

px=&y; //а теперь – на ячейку со значением y

3. Значение переменной, на которую показывает указатель, берется унарной операцией "\*" ("взять значение"):

x=\*px; //косвенно выполнили присваивание x=y

(\*px)++; //косвенно увеличили значение y на 1

**21. Управление памятью (new, delete). Примеры использования.**

Выделение памяти осуществляется с помощью оператора [new](http://ru.wikipedia.org/wiki/New_(C++)) и имеет вид: тип\_данных \*имя\_указателя = new тип\_данных;, например int \*a = new int;. После удачного выполнения такой операции, в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа int.

Инициализация значения, находящегося по адресу указателя выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением: тип данных \*имя\_указателя = new тип\_данных(значение). В нашем примере это int \*b = new int(5).

Для того, чтобы получить адрес в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом &. перед ним

Для того, чтобы получить значение, которое находится по адресу, на который ссылается указатель, используется префикс \*. Данная операция называется разыменованием указателя.

При использовании оператора delete для указателя, знак \* не используется.

**22. Основные принципы ООП. Абстрагирование.**

Абстрагирование- это выделение таких существенных характеристик объекта, которые отличают его от всех других видов объектов и таким образом чётко определяются особенности данного объекта с точки зрения дальнейшего его рассмотрения. Абстрагирование позволяет отделить самые существенные особенности поведения от несущественных. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Абстракция сущности (Объект представляет собой полезную модель некой сущности в предметной области).

Абстракция поведения (Объект состоит из обобщенного множества операций)

Абстракция виртуальной машины (Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня)

Произвольная абстракция (Объект включает в себя набор операций, не имеющих друг с другом ничего общего)

**23. Основные принципы ООП. Наследование.**

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность. При объявлении нового класса новый класс называется потомком или дочерним классом старого класса , называемого предком или родительским классом.

**24. Основные принципы ООП. Инкапсуляция.**

Инкапсуляция — это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик — пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов.

**25. Основные принципы ООП. Полиморфизм.**

Полиморфизм – свойство классов решать схожие по смыслу проблемы разными способами. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением сигнатуры метода. Это обеспечивает сохранение неизменным интерфейса класса-предка и позволяет осуществить связывание имени метода в коде с разными классами — из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем.

**26. Определение класса. Поля и методы класса. Примеры создания классов.**

Class, private, public, protected

**27. Определение методов в классе и вне класса. Примеры.**

Чтобы вынести функции класса за пределы класса, нужно использовать операцию разрешения контекса, в народе — двойное двоеточие: ::.

Для выноса функций из класса, но оставления принадлежности функций к классу, внутри класса оставляются прототипы выносимых за пределы класса функций, а вне класса функциям уточняют принадлежность объекта к классу с помощью операции разрешения контекста.

type class\_name::method\_name() :: - операция разрешения

**28. Конструкторы и деструкторы. Примеры.**

Необходимость инициализации объекта является общим требованием, поэтому в языке С++ есть возможность делать это автоматически при объявлении объекта. Это автотматическая инициализация реализуется с помощью использования функции, называемой коструктором. Конструктор - это специальная функция, являющаяся членом класса и имеющая тоже имя, что и класс. Инициализация не проводится в теле конструктора. Инициилизирующее значение расположено в скобках после имени поля, если необходимо инициализировать сразу несколько полей, то значение разделяются запятыми и образуют списко инициализации

Нельзя вызывать конструктор в явном виде. Конструктор может быть перегруженным. Если в классе не объявлен ни один конструктор, то компилятор сам создает конструктор класса. Для освобождения памяти из-под объекта, т.е. при прекращении действия объекта выполняется деструктор. Деструктор имеет такое же имя как и класс, только перед ним ставится значок "~". Деструктор не имеет параметров и не возвращает значений.

**29. Конструктор копирования по умолчанию. Пример.**

Так же есть способ инициализации объекта, использующий значение полей уже существующего объекта. Такой конструктор представляется компилятором для каждого создаваемого класса и называется копирующим коструктором по умолчанию. Этот конструктор имеет единственный аргумент, являющийся объектом того же класса, что и конструктор. Конструктор копирования по умолчанию создается всякий раз, когда создается новый объект и в качестве его значений выбирается существующий объект того же типа. Тогда создали такой Product P еще два объекта можно инициилизировать следующим образом: Product P1(P); Product P3 = P; В обоих случаях вызывается конструктор копирования по умолчанию, действия которого заключаются в копировании значнеий полей объекта P в P1 и P3

**30. Статические данные класса. Пример.**

Если поле данных класса описано с ключевым словом static, то значение этого поля будет одинаковым для всех объектов этого класса. Статические данные класса применяются тогда, когда необходимо чтобы все объекты включали одинаковые значения. Статические данные видны только внутри класса и время жизни этих данных совпадает со временем выполнения программы. Объявление статического поля находится внутри класса, а определение поля располагается вне класса и представляет собой определение глобальной переменной. Это делается потому что, если бы определение находилось внутри класса, то это бы нарушило принцип, в соответствии с которым определение класса не должно быть связано с выделением памяти. Поместив статическое поле вне класса мы обеспечили однакркатное выделение памяти под это поле до того как программа будет запущена на выполнение и статическое поле станет доступно всему классу. static int weight; int Person::weight = 10;

**31. Перегрузка унарных операций. Примеры.**

тип operator @ (список\_параметров-операндов)  
{  
// … тело функции …  
}

где @ — знак перегружаемой операции (-, +, \*  и т. д.),  
тип — тип возвращаемого значения.

Тип возвращаемого значения должен быть отличным от void, если необходимо использовать перегруженную операцию внутри другого выражения.

 complex c = a + b;          // сокращенная запись

 complex d = operator+(a,b); // явный вызов

Если унарная операция перегружается как функция-член, то она не должна иметь аргументов, так как в этом случае ей передается неявный аргумент-указатель this на текущий объект.

Если унарная операция перегружается дружественной функцией, то она должна иметь один аргумент – объект, для которого она выполняется.

**32. Перегрузка бинарных операций. Примеры.**

Если бинарная операция перегружается с использованием функции-члена, то в качестве своего первого аргумента она получает неявно переданную переменную класса (указатель this на объект), а в качестве второго — аргумент из списка параметров. То есть, фактически, бинарная операция, перегружаемая функцией-членом, имеет один аргумент (правый операнд), а левый передается неявно через указатель this.

complex complex :: operator +(complex &c) {

  complex temp;

  temp.real=this->real+c.real;

  temp.imag=this->imag+c.imag;

Если бинарная операция перегружается дружественной функцией, то в списке параметров она должна иметь оба аргумента:

complex operator + (const complex &c1, const complex &c2) {

  complex temp;

  temp.real=c1.real+c2.real;

  temp.imag=c1.imag+c2.imag;

**33. Перегрузка операций индексации массива. Пример.**

char   operator[](int);

char String::operator[](int index)  {

  return (str[index]);}

для работы с массивом класса

**34. Преобразование типов от основного к пользовательскому. Пример.**

Для перехода от основного к определенному пользователем типу используется конструктор с одним аргументом(в качестве аргумента будет использоваться базовый тип)

Его называют конструктором преобразования.

operator тип\_в\_который\_преобразуем()

{...;return переменная нового типа;}

Операция мб вызвана явно

переменная = static\_cast<тип\_в\_который\_преобразуем>(объект);

переменная = объект; // Неявно

**35. Преобразование типов от пользовательского к основному. Пример.**

Для преобразования от определенного пользователем типа к основному создается операция преобразования.

**36. Преобразование типов от пользовательского к пользовательскому. Пример.**

Для преобразования между объектами различных, определенных пользователем классов применяются те же два метода преобразования, что и для преобразований между основными типами и объектами определенных пользователем классов. То есть можно использовать конструктор с одним аргументом или операцию преобразования. Выбор зависит от того, нужно ли записать функцию преобразования в классе для исходного объекта или для объекта назначения

**37. Базовый и производный классы. Примеры использования спецификаторов доступа.**

Наследование - это процесс создания новых классов называемых наследниками или производными из уже существующих базовых классов. Производный класс - пролучает все возможности базового класса, но может быть усовершенствован. Основная форма наследования class name\_child\_class:режим\_доступа name\_base\_class Существует три режима доступа: public, private, protected. Послдений позволяет членам быть доступными методам своего и производного класса, при этом ограничивает доступ из функций, не принадлежащих этим классам. Если режим доступа отсутствует, то у производного класса подразумевается private. Использование спецификаторов в различных ситуациях:

|К элементу в |Режим |К элементу в |

|базовом классе |доступа |производном классе |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

| private | |недоступен |

| protected |public |protected |

| public | |public |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

| private | |недоступен |

| protected |protected |protected |

| public | |public |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

| private | |недоступен |

| protected |private |private |

| public | |private |

Для объектов производного класса могут быть использованы методы базового класса.Это называется правами доступа. Если в производном классе не определен конструктор,

то будет использоваться подходящий конструктор базового класса. Также, есть возможность использовать доступный метод взамен отсутствующего.Наследование не работает

в обратном направлении и базовому классу не доступны производные методы.

**38. Конструкторы производного класса. Примеры.**

Для инициализации объекта производного класса нельзя воспользовать конструктором базового класса. Компилятор будет использовать только конструктор базового класса без аргументов. Для производного класса нужно писать новые конструкторы. В примере конструктор A(int A1, char C1) вызывает конструктор родителя С(int A1, char C1) записанный через ":".

**39. Перегрузка методов базового класса. Примеры.**

Для производного класса можно определять методы, имеющие такие же имена, что и у методов базового класса. В этом случае имеет место перегрузка функции. В случае, если метод с одним и тем же названием есть и в базовом, и в производном классе, то будет выполнен метод производного класса. Методы производного класса получают доступ к методам базового класса используя операцию разрешения(::).

**40. Пример иерархии классов.**

**41. Множественное наследование. Неопределенности при множественном наследовании.**

Класс может быть производным от нескольких базовых классов. Форма описания следующая: class имя производного класса: список базовых классов{...}.

Список базовых классов содержит перечисленные через запятую классы с соответствующими режимами доступа каждому базовому классу. Пока конструкторы базовых классов не

имеют аргументов, то производный класс может не иметь функцию конструктора. Если конструктор базового класса имеет 1 или неск аргументов, то каждый производный класс

обязан иметь конструктор.Чтобы передать аргументы в базовый класс нужно определить их после объявления конструктора базового класса.

Форма записи: имя производного класса(список аргументов: базовыйКласс1(список аргументов)...базовыйКлассN(...)).

Здесь базовыйКласс1(список аргументов)...базовыйКлассN(...)

Имена конструкторов базового класса, которые наследуются в производном классе. : отделяет имя конструктора производного класса от списка аргументов базвого класса.

Список аргументов, ассоциированный с базовым классом, может состоять из констант и переменных

Неопределенность при множественном наследовании

Допустим, что в обоих классах A и B с-ют методы с одинаковыми именами, а в производном классе метода с таким именем нет.Здесь проблема решается путем использования

оператора разрешения, определяющего класс, в котором находится метод.Процесс направления в версии метода конкретного класса называется устранением неоднозначности.

class A{...};

class B{...};

class C: public A, public B{...};

Мы создаем производный класс от двух базовых классов, которые в свою очередь являются производными одного класса.Это создает дерево наследования в форме ромба.

class A{...};

class B{...};

class C: public A, public B{...};

class D: public B, public C{...};

Трудности начинаются, когда объект класса D пытается воспользоваться методом класса А. Пусть объекту класса D необходимо воспользоваться методом Show() класса А.

Классы B и С содержат копии метода Show(), унаследованные от класса А. Тогда компилятор не может решить какой из методов Show() использовать.

**42. Функции-шаблоны. Пример использования.**

Функция шаблона определяет общий набор операций, который будет применен к различным типам данных. Шаблоны используются для создания каркаса функции, оставляя компилятору реализацию подробностей. template type name\_function(args){} Пользовательский тип является "Держателем места" для имени типа данных, которое используется функциями. Он(тип) будет автоматически заменен компилятором на фактический тип данных во время создания конкретной версии функции. Можно определить несколько типов шаблонов данных, используя список с запятыми в качестве разделителя.

**43. Классы-шаблоны. Пример использования.**

полезны тогда, когда класс содержит логику допускающую значительные обобщения используя классы шаблона, можно создавать классы, поддерживающие очереди, связные списки

и тд для произвольных типов данных. Компилятор автоматически создаёт корректный код, основываясь на типе данных, указанном перед компиляцией

общая форма записи класса-шаблона

template <class тип>

class имя\_класса{}

при необходимости можно определить неск пользовательских типов, указывая их через запятую. При создании класса-шаблона можно создать конкретный экземпляр этого класса

используя след форму записи:имя\_класса <тип> имя\_объекта

template <class MyT>

class stack

{

MyT stck[100];

int n;

public:

stack();

~stack();

void push(MyT i);

MyT pop();

};

template <class MyT>

stack <MyT>:: stack(){n = 0}

template <class MyT>

stack <MyT>:: push(Myt i)

{

if (n == 100)

{

cout << "NO" << endl;

return n;

}

else stck[n] = i;

n++;

}

template <class MyT>

stack <MyT>:: pop()

{

if(n == 0)

{

cout<<"NO"<<endl;

return 0;

}

n--;

return stck[n];}

**44. Обработка исключений. Использование нескольких инструкций catch. Пример.**

Используя обработку исключений, программа может автоматически вызывать функцию-обработчик ошибок тогда, когда такая ошибка возникает.Обработка исключений использует

3 ключевых слова: try, catch, throw(Передать); Те инструкции, в которых ожидается появление исключительных ситуаций, содержится в блоке try. Если в блоке try возникает

исключение, т.е. ошибка, то генерируется исключение, которое перехватывается используя catch.

Синтаксис:

try{...}

catch(тип1 аргумент1){...}

catch(тип2 аргумент2){...}

Одному блоку try может отвечать неск. инструкций блока catch. Какая именно инструкция catch исполняется, зависит от типа исключения.Перехваченным может быть любой

тип данных, включая созданные программистом классы.Если никакого исключения не сгенерировано(т.е. никакой ошибки не возникло в блоке try), то инструкции catch

выполняться не будут.Исключение мб сгенерировано из функции вызванной изнутри блока try

throw должна выполняться либо внутри блока try, либо в функции, вызванной из блока try.

Общая форма записи:

throw исключение;

Пример:

int num1, num2;

cout <<"num1 = ";cin >> num1;

cout <<"num2 = ";cin >> num2;

cout <<"num1 + num2 = " << num1 + num2 << endl;

cout <<"num1 / num2 =" ;

set\_terminate(f);

try{

if(num2 == 0){throw 100;} если num2 = 0, то в num2 перейдет 100

cout << num1/num2 << endl;

}

catch (int i){cout << "Нельзя делить на 0.Ошибка " << i << endl;}

cout << "num1 - num2 = " << num1 - num2 << endl;

Если генерируется исключение, для которого отсутствует подходящая инструкция catch, то произойдет аварийное завершение программы. При генерации необработанного исключения

вызывается функция terminate().По умолчанию, функция terminate() вызывает функцию abort(), завершающую выполение программы.

Однако можно задать свою собственную обработку используя функцию set\_terminate()

////////2.Использование нескольких конструкций catch

Если используется несколько конструкций catch, то они должны перехватывать разные типы исключений. Инструкции catch проверяются на соответствие типа в порядке их

расположения в программе. Каждая инструкция catch отвечает только за свой тип данных

**45. Обработка исключений. Перехват всех исключений. Пример.**

Если требуется перехватить все исключения,а не какой-то конкретный тип, то исп след форма записи:

catch(...(обязательно ... что значит перехватывем всё)){...}

int test;cin >>test;

try{

if(test == 0){throw test}

if(test == 1){throw 'a'}

if(test == 2){throw 120.5}

}

catch(...){cout << "Перехвачено исключение" << endl;}

**46. Обработка исключений. Задание ограничений на исключения. Пример.**

Когда фукция вызывается из блока try можно ограничить тип исключений, который эта функция может сгенерировать.Кроме того, можно вообще запретить ей генерировать

исключения. Для этого используется следующий синтаксис:

тип\_функции имя\_функции(аргументы)

throw(список типов)

Здесь ▲ только типы данных, содержащиеся в списке типов, могут быть сгенерированы данной функцией. Если будут сгенерированы исключения какого-либо типа, то произойдет

аварийное завершение программы.Если необходимо, чтобы функция не могла сгенерировать никакого исключения, то следует оставить список пустым

Попытка генерации исключения будет иметь своим результатом вызов функции

unexpected().

Эта функция в свою очередь вызывает функцию terminate. Можно задать свой собственный порядок обработки непредвиденных исключений с помощью функции set\_expected()

используется как set\_terminate.

void F(int test) throw(int, char, double);{

if(test == 0){throw test}

if(test == 1){throw 'a'}

if(test == 2){throw 120.5}

}

int main{

int x; cin >> x;

try{

F(x);

}

catch(int i){cout << "INT" << endl;}

catch(char c){cout << "CHAR" << endl;}

catch(double d){cout << "DOUBLE" << endl;}

}

**47. Обработка исключений. Повторная генерация исключений.**

Если возникает необходимость снова сгенерировать исключение из try, можно сделать это путем вызова без указания исключения throw. В результате текущее исключение будет передано во внешнюю последовательность try / catch обработки исключений.

void f(){

try{throw "Hello";}

catch(char \*){

cout << "Исключение2" << endl; throw; } }

int main(){

try{f();}

catch(char \*){

cout << "Исключение1" << endl; } }

**48. Потоковый ввод/вывод файлов. Примеры.**

При открытии файла с ним связывается ***поток ввода-вывода***. Выводимая информация записывается в поток, вводимая информация считывается из потока.  
  
Для работы с файлами необходимо подключить заголовочный файл <fstream>. В нем определены несколько классов и подключены заголовочные файлы

* <ifstream> — файловый ввод ;
* <ofstream> — файловый вывод.

Файловый ввод-вывод аналогичен стандартному вводу-выводу, единственное отличие – это то, что ввод-вывод выполнятся не на экран, а в файл.  
  
Если ввод-вывод на стандартные устройства выполняется с помощью объектов cin и cout, то для организации файлового ввода-вывода достаточно создать собственные объекты, которые можно использовать аналогично [этим операторам](https://prog-cpp.ru/cpp-std/).  
  
При работе с файлом можно выделить следующие этапы:

* создать объект класса fstream (возможно, ofstream или ifstream);
* связать объект класса fstream с файлом, который будет использоваться для операций ввода-вывода;
* осуществить операции ввода-вывода в файл;
* закрыть файл.
* Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе ios предусмотрены константы, которые определяют режим открытия файлов.

| **Константа** | **Описание** |
| --- | --- |
| ios::in | открыть файл для чтения |
| ios::out | открыть файл для записи |
| ios::ate | при открытии переместить указатель в конец файла |
| ios::app | открыть файл для записи в конец файла |
| ios::trunc | удалить содержимое файла, если он существует |
| ios::binary | открытие файла в двоичном режиме |

* Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове метода open().
* ofstream fout("file.txt", ios::app);  
  fout.open("file.txt", ios::app);
* Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции **ИЛИ** |, например:  
    
  **ios::out | ios::in** - открытие файла для записи и чтения.

**49. Файловый ввод/вывод с помощью put и get. Примеры.**

Считывание целой строки до перевода каретки производится так же как и в **iostream** методом **getline()**. Причем рекомендуется использовать его переопределеную версию в виде функции, если считывается строка типа string:

|  |  |
| --- | --- |
| 1  2  3  4 | //Считка строки из текста  string s;  getline(file,s);  cout << s << endl; |

Если же читать нужно в массив символов char[], то либо **get()** либо **getline()** именно как методы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int n = 10;  //Создаем буффер для чтения  char\* buffer = new char[n+1];          buffer[n]=0;  //Читаем n символов  file.get(buffer,n);  //Или так, но до первого пробела  file.getline(buffer,n,' ');  //выводим считанное  cout << buffer;  //Освобождаем буффер  delete [] buffer; |

Принцип в общем тот же, что и в аналогах из iostream: Указывается в параметрах буфер (переменная, куда будет производиться чтение), или точнее указатель на блок памяти (если переменная объявлена статически: **char buffer[255]** к примеру, то пишется в параметры &buffer), указывается максимальное количество считываемого (в примере это n), дабы не произошло переполнение и выход за пределы буфера и по необходимости символ-разделитель, до которого будет считка (в примере это пробел).

С помощью функции **put()**можно *записать байт*, а с помощью функции **get() –***считать* *байт*.

**50. Файловый ввод/вывод с помощью read и write. Примеры.**

|  |
| --- |
| //Считка из файла N байт  int n=10;  //Создаем буффер  char\* buffer=new char[n+1]; buffer[n]=0;  //Читаем в него байты  file.read(buffer,n);  //выводим их на экран  cout<<buffer;  delete [] buffer; |

Собственно тут тот же результат — считается указанное количество символов. Исключение только в том, что нельзя указать разделитель. read() применяется для неформатированного ввода. Призван в первую очередь читать бинарные файлы. Поскольку текстовый файл — частный случай бинарного, этот метод вполне применим и к текстовому файлу.

Для считывания и записи группы *байтов* (*блоков двоичных данных*) используют функции **read()** и **write()**

istream& **read(**char **\* buf**, type **num);**

ostream& **write(**const char \* **buf**, type **num);**

Функция **read()**считывает**num**байтов данных из связанного с файлом потока и помещает их в буфер, адресуемый параметром **buf.** Функция **write()**записывает **num**байтов данных в связанный с файлом поток из буфера, адресуемого параметром **buf.**Тип **type** должен быть определён как некоторая разновидность целочисленного типа.

При считывании конца файла функция**read()**возвращает значение **0.**

Приведение типа к **(char\*)** в функциях **read()** и **write()**необходимо, если буфер ввода–вывода не определён как символьный массив, так как в **С++**указатель на один тип не преобразуется автоматически в указатель на другой тип.

Если при считывании данных конец файла достигнут до того, как было считано **num** символов, выполнение функции **read()** просто прекращается.