# **Абстрактные типы данных**

**Абстракция** - разделение и независимость интерфейса и реализации.

**Интерфейс** - средство взаимодействия с объектом.

**Реализация** - внутреннее свойство объекта.

Абстракция с помощью модульности обязывает скрывать детали реализации, предотвращая ее умышленное или случайное изменение. Интерфейс и реализация должны быть максимально независимы друг от друга, т.е. изменение кода функций класса не должно изменять интерфейс. Различают процедурную абстракцию и абстракцию данных. Процедурная абстракция требует раздельного рассмотрения назначения процедуры (например, функции С/С++) и ее реализации. Абстракция данных требует раздельного рассмотрения операций над данными и реализации этих операций. Достаточно знать, какие операции выполняет модуль (функция С/С++), но не требуется знать, какие данные он при этом использует (они скрыты) и как в действительности выполняются эти операции

# **Базовые принципы объектно-ориентированного программирования**

ООП основывается на трех основных концепциях: **инкапсуляции**, **полиморфизме** и **наследовании**

**Инкапсуляция** - механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования. Инкапсуляция позволяет создавать объект, являющийся логическим целым, включающим данные и код для работы с этими данными. Объект обеспечивает защиту против модификации частных (private) составляющих его членов. Закрытые данные или методы доступны только для других частей этого объекта и недоступны вне его.

**Полиморфизм** обеспечивает возможность реагировать различным образом на одно и то же сообщение (вызов функции-члена). Поддержка полиморфизма в ООП осуществщ+яется через виртуальные функции и механизм перегрузки и переопределения. Ключевым в понимании полиморфизма является то, что он позволяет манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

**Наследование** представляет собой механизм, благодаря которому новый (производный) класс может создаваться, наследуя свойства от уже существующего (базового) класса. Новый класс, используя наследование, нуждается только в определении специфических только для этого класса компонент. Может быть одиночным и множественным.

**Передача сообщений**: программы представляются в виде набора объектов и передачи сообщений между ними.

class ***Base***

{private:

int variable;

public:

virtual void PrintInfo()

{ cout << "Base." << endl; }

int GetVariable()

{ return variable; }

void SetVariable(int variable)

{ this->variable = variable; }};

class ***Derived*** : public Base

{public:

virtual void PrintInfo() override

{ cout << "Derived." << endl; }};

void ***main***()

{ Base obj1; Derived obj2; Base\* p; p = &obj1; p->PrintInfo(); p = &obj2; p->PrintInfo(); cout << obj1.GetVariable();}

# **Основные достоинства языка С++**

Язык С++ основывается на языке С, сохраняя большую часть возможностей языка С и расширяя их новыми, ориентированными на реализацию идей ООП. Язык С++ является мобильным и легко переносимым языком. Получаемый программный код обладает высоким быстродействием и компактными размерами.

# **Особенности языка С++**

Необходимо четко представлять, что достоинство языка С++ состоит не в добавлении в С новых типов, операций и т.д., а в возможности поддержки объектно-ориентированного подхода к разработке программ.

# **Ключевые слова**

Язык С++ расширяет множество ключевых слов, принятых в языке С, следующими ключевыми словами: class, new, inline, try, private, delete, operator, catch, public, this, template, throw, protected, friend, virtual

# **Константы и переменные**

В С++ односимвольные константы (данные, не изменяющие своего значения) имеют тип char, в то же время в С++ поддерживается возможность работы с двухсимвольными константами типа int: ’aB’ , ’\n\t’ . При этом первый символ располагается в младшем байте, а второй − в старшем.

const int x = 22; x = 78; const float pi = 3.14;

# **Операции**

В языке С++ введены следующие новые операции:

**::** − операция разрешения контекста;

**.\*** и **->\*** − операции обращения через указатель к компоненте класса;

**new** и **delete** − операции динамического выделения и освобождения памяти.

# **Типы данных**

В С++ поддерживаются все типы данных, предопределенные в С. Кроме того, введено несколько новых типов данных: классы, ссылки, bool, auto.

**Ссылки** расширяют и упрощают используемую в С передачу аргументов в функцию: по значению и по адресу.

**bool**: логический тип. Может принимать одну из двух значений true (истина) и false (ложь).

Иногда бывает трудно определить тип выражения. И согласно последним стандартам можно предоставить компилятору самому выводить тип объекта. И для этого применяется спецификатор **auto**. При этом если мы определяем переменную со спецификатором auto, эта переменная должна быть обязательно инициализирована каким-либо значением: auto number = 5;

# **Передача аргументов в функцию по умолчанию**

В С++ поддерживается возможность задания некоторого числа аргумен-тов по умолчанию. Это означает, что в заголовке функции некоторым параметрам при их описании присваиваются значения. При вызове данной функции число фактических параметров может быть меньше числа формальных параметров. В этом случае принимается умалчиваемое значение соответствующего параметра. Например:

#include <iostream>

using mamespace std;

int sm(int i1, int i2, int i3=0, int i4=0)

{ cout<<i1<<’ ’<<i2<<’ ’<<i3<<’ ’<<i4<<’ ’;

return i1+i2+i3+i4; }

int main()

{ cout <<"сумма = "<< sm(1,2) << endl;

cout <<"сумма = "<< sm(1,2,3) << endl;

cout << "сумма = "<< sm(1,2,3,4) << endl;

return 0; }

Результатом работы программы будет:

1 2 0 0 сумма = 3

1 2 3 0 сумма = 6

1 2 3 4 сумма = 10

Описание параметров по умолчанию должно находиться в конце списка формальных параметров (в заголовке функции). Задание параметров по умолчанию может быть выполнено только в прототипе функции или при его отсутствии в заголовке функции.

# **Простейший ввод и вывод**

В С++ ввод и вывод данных производится потоками байт. Поток (последовательность байт) − это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода-вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот.

Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.h:

cin − поток стандартного ввода;

cout − поток стандартного вывода;

cerr − поток стандартной ошибки;

clog − буферизируемый поток стандартных ошибок.

# **Объект cin**

Для ввода информации с клавиатуры используется объект cin. Формат записи cin имеет следующий вид:

cin [>>имя\_переменной];

Объект cin имеет некоторые недостатки. Необходимо, чтобы данные вводились в соответствии с форматом переменных, что не всегда может быть гарантировано. Например:

int a;

cin>>a;

# **Объект cout**

Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид:

сout << data [ << data];

data − это переменные, константы, выражения или комбинации всех трех типов.

Простейший пример применения cout − это вывод, например, символьной строки:

cout << ”объектно-ориентированное программирование ”;

cout << ”программирование на С++”.

Надо помнить, что cout не выполняет автоматический переход на новую строку после вывода информации. Для перевода курсора на новую строку надо вставлять символ ’\n’ или манипулятор endl. cout << ”объектно-ориентированное программирование \n”;

cout << ”программирование на С++”<<endl;

Для управления выводом информации используются манипуляторы.

# **Манипуляторы ввода/вывода**

Для форматирования выводимой информации используются манипуляторы. Описания для стандартных манипуляторов включены в файл iomanip.h Манипуляторы dec, hex и oct используются для вывода числовой информации в десятичном, шестнадцатеричном или восьмеричном представлении.

Применение их можно видеть на примере следующей программы:

#include <iostream> using namespace std;

int main()

{ int a=0x11, b=4, // целые числа: шестнадцатеричное и десятичное c=051, d=8, // восьмеричное и десятичное i,j;

i=a+b;

j=c+d;

cout << i <<' ' <<hex << i <<' '<<oct << i <<' ' <<dec << i <<endl;

cout <<hex << j <<' ' << j <<' '<<dec << j <<' ' << oct << j <<endl;

return 0;}

В результате выполнения программы на экран будет выведена следующая информация:

21 15 25 21 31 31 49 61

Манипуляторы изменяют значение некоторых переменных в объекте cout. Эти переменные называются флагами состояния. Когда объект cout посылает данные на экран, он проверяет эти флаги.

Рассмотрим манипуляторы, позволяющие выполнять форматирование выводимой на экран информации:

#include <iostream> #include <iomanip> using namespace std;

int main() { int a=0x11;

double d=12.362;

cout << setw(4) << a << endl;

cout << setw(10) << setfill('\*') << a << endl;

cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl;

return 0; }

Результат работы программы:

17 \*\*\*\*17 12.4

В приведенной программе использованы манипуляторы setw(), setfill(' ') и setprecision(). Синтаксис их показывает, что это функции. На самом деле это компоненты-функции, позволяющие изменять флаги состояния объекта cout. Для их использования необходим заголовочный файл iomanip.h. Функция setw(4) обеспечивает вывод значения переменной a в четы-ре позиции. Функция setfil(’символ’) заполняет пустые позиции символом. Функция setprecision(n) обеспечивает вывод числа с плавающей запятой с точностью до n знаков после запятой (при необходимости производится округление дробной части). Если при этом не установлен формат вывода с плавающей запятой, то точность указывает общее количество значащих цифр. Заданная по умолчанию точность − шесть цифр. Таким образом, функции имеют следующий формат:

setw(количество\_позиций\_для\_вывода\_числа) setfil(символ\_для\_заполнения\_пустых\_позиций)

setprecision (точность\_при\_выводе\_дробного\_числа)

Наряду с перечисленными выше манипуляторами в С++ используются также манипуляторы setiosflags() и resetiosflags() для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на переменные состояния. Функция setiosflags() устанавливает указанные в ней флаги, а resetiosflags() очищает их.

Пример программы, в которой использованы манипуляторы:

#include <iostream> #include <iomanip> using namespace std;

int main() { char s[]="БГУИР факультет КСиС"; cout << setw(30) << setiosflags(ios::right) << s << endl; cout << setw(30) << setiosflags(ios::left) << s << endl; }

Результат работы программы:

БГУИР факультет КСиС БГУИР факультет КСиС

Наряду с манипуляторами setiosflags() и resetiosflags(), для того чтобы установить или сбросить некоторый флаг, могут быть использованы функции класса ios setf() или unsetf(). Например:

#include <iostream> using namespace std;

#include <string.h> int main() { char \*s="Я изучаю С++";

cout.setf(ios::uppercase | ios::showbase | ios::hex);

cout << 88 << endl;

cout.unsetf(ios::uppercase);

cout << 88 << endl;

cout.unsetf(ios::uppercase | ios::showbase | ios::hex);

cout.setf(ios::dec);

int len = 10 + strlen(s);

cout.width(len); cout << s << endl; cout << 88 << " hello C++ " << 345.67 << endl;

return 0; }

Результат работы программы:

0X58 0x58 Я изучаю С++ 88 hello C++ 345.67

# **Операторы для динамического выделения и освобождения памяти (new и delete)**

Различают два типа памяти: статическую и динамическую. В статической памяти размещаются локальные и глобальные данные при их описании в функциях. Для временного хранения данных в памяти используется динамическая память, или heap. Размер этой памяти ограничен, и запрос на динамическое выделение памяти может быть выполнен далеко не всегда.

Для работы с динамической памятью в языке С использовались функции calloc, malloc, realloc, free и др. В С++ для операций выделения и освобождения памяти можно также использовать встроенные операторы new и delete.

Оператор new имеет один операнд. Оператор имеет две формы записи:

[::] new [(список\_аргументов)] имя\_типа [(инициализирующее\_значение)]

[::] new [(список\_аргументов)] (имя\_типа) [(инициализирующее\_значение)]

В простейшем виде оператор new можно записать:

new имя\_типа или new (имя\_типа)

Оператор new возвращает указатель на объект типа «имя\_типа», для которого выполняется выделение памяти. Например:

char \*str; // str – указатель на объект типа char str=new char; // выделение памяти под объект типа char или str=new (char);

В качестве аргументов можно использовать как стандартные типы данных, так и определенные пользователем. В этом случае именем типа будет имя структуры или класса. Если память не может быть выделена, оператор new возвращает значение NULL.

Оператор new позволяет выделять память под массивы. Он возвращает указатель на первый элемент массива в квадратных скобках. Например:

int \*n; // n – указатель на целое

n=new int[20]; // выделение памяти для массива

При выделении памяти под многомерные массивы все размерности кроме крайней левой должны быть константами. Первая размерность может быть задана переменной, значение которой к моменту использования new известно пользователю, например:

int \*n; // n – указатель на целое

n=new int[20]; // выделение памяти для массива

k=3;

int \*p[]=new int[k][5]; // ошибка cannot convert from 'int (\*)[5]' to 'int \*[]'

int (\*p)[5]=new int[k][5]; // верно

При выделении памяти под объект его значение будет неопределенным.

Однако объекту можно присвоить начальное значение.

int \*a = new int (10234);

Этот параметр нельзя использовать для инициализации массивов. Однако на место инициализирующего значения можно поместить через запятую список значений, передаваемых конструктору при выделении памяти под массив (массив новых объектов, заданных пользователем). Память под массив объектов может быть выделена только в том случае, если у соответствующего класса имеется конструктор, заданный по умолчанию.

class matr

{ int a; float b;

public:

matr(){}; // конструктор по умолчанию

matr(int i,float j): a(i),b(j) {}

~matr(){}; };

int main()

{ matr mt(3,.5);

matr \*p1=new matr[2]; // верно р1 − указатель на 2 объекта

matr \*p2=new matr[2] (2,3.4); // неверно, невозможна инициализация

matr \*p3=new matr (2,3.4); // верно р3 – инициализированный объект }

Следует отметить, что в примере конструктор по умолчанию требуется при использовании оператора new matr[2], т.е. создании массива объектов.

Оператор new вызывает функцию operator new().

Аргумент имя\_типа используется для автоматического вычисления размера памяти sizeof(имя\_типа), т.е. инструкция типа new имя\_типа приводит к вызову функции:

operator new(sizeof(имя\_типа));

Доопределение оператора new позволяет расширить возможности выделения памяти для объектов (их компонент) данного класса.

Создание объекта с помощью операции new вызывает также выполнение конструктора для этого объекта. Если в new не указан список инициализации либо он пуст, то выполняется конструктор по умолчанию (default), который будет рассмотрен ниже. Если имеется непустой список инициализации, то выполняется тот конструктор, для которого этот список соответствует списку аргументов. При создании массива выполняется стандартный конструктор для каждого элемента.

Отметим преимущества использования оператора new перед использованием malloc():

- оператор new автоматически вычисляет размер необходимой памяти.

Не требуется использование оператора sizeof(). При этом он предотвращает выделение неверного объема памяти;

- оператор new автоматически возвращает указатель требуемого типа (не требуется использование оператора преобразования типа);

- имеется возможность инициализации объекта;

- можно выполнить перегрузку оператора new (delete) глобально или по отношению к тому классу, в котором он используется.

Для разрушения объекта, созданного с помощью оператора new, необходимо использовать в программе оператор delete.

Оператор delete имеет две формы записи:

[::] delete переменная\_указатель // для указателя на один элемент

[::] delete [] переменная\_указатель // для указателя на массив

Единственный операнд в операторе delete должен быть указателем, возвращаемым оператором new. Если оператор delete применить к указателю, полученному не посредством оператора new, то результат будет непредсказуем.

Использование оператора delete вместо delete[] по отношению к указате-лю на массив может привести к логическим ошибкам. Таким образом, освобождать память, выделенную для массива, необходимо оператором delete [], а для отдельного элемента − оператором delete. #include <iostream> using mamespace std;

class A { int i; // компонента-данное класса А public:

A(){} // конструктор класса А

~A(){} // деструктор класса А };

int main()

{ A \*a,\*b; // описание указателей на объект класса А

float \*c,\*d; // описание указателей на элементы типа float

a=new A; // выделение памяти для одного объекта класса А

b=new A[3]; // выделение памяти для массива объектов класса А

c=new float; // выделение памяти для одного элемента типа float

d=new float[4]; // выделение памяти для массива элементов типа float

delete a; // освобождение памяти, занимаемой одним объектом

delete [] b; // освобождение памяти, занимаемой массивом объектов

delete c; // освобождение памяти одного элемента типа float

delete [] d; // освобождение памяти массива элементов типа float }

При удалении объекта оператором delete вначале вызывается деструктор этого объекта, а потом освобождается память. При удалении массива объектов с помощью операции delete[] деструктор вызывается для каждого элемента массива.

# **Объекты**

Объект С++ − абстрактное описание некоторой сущности.

Свойства объекта: состояние и поведение, которые однозначно идентифицируют объект.

Состояние объекта объединяет все его поля данных (статическая компонента) и текущие значения каждого из этих полей (динамическая компонента).

Поведение объекта определяет, как объект изменяет свои состояния и взаимодействует с другими объектами. Идентификация объекта позволяет выделить объект из числа других объектов.

Процедурный подход к программированию предполагает разработку взаимодействующих подпрограмм, реализующих некоторые алгоритмы. Объектно-ориентированный подход представляет программы в виде взаимодействующих объектов. Взаимодействие объектов осуществляется посредством сообщений. Под передачей сообщения объекту понимается вызов некоторой функции (компонента этого объекта).

Говоря об объекте, можно выделить две его характеристики: интерфейс и реализацию.

Интерфейс показывает, как объект общается с внешней средой. Он может быть ассоциирован с окном, через которое можно заглянуть внутрь объекта и получить доступ к функциям и данным объекта.

Все данные делятся на локальные и глобальные. Локальные данные недоступны (через окно). Доступ к ним и их модификация возможна только из компонент-функций этого объекта. Глобальные данные видны и модифицируемы через окно (извне). Для активизации объекта ему посылается сообщение. Сообщение проходит через окно и активизирует некоторую глобальную функцию. Тип сообщения определяется именем функции и значениями передаваемых аргументов. Локальные функции (за некоторым исключением) доступны только внутри класса.

Говоря о реализации объекта, мы подразумеваем особенности реализации функций соответствующего класса (особенности алгоритмов и кода функций).

Объект включает в себя все данные, необходимые, чтобы описать сущность, и функции или методы, которые манипулируют этими данными.

# **Понятие класса**

Класс – это множество объектов, имеющих общую структуру и поведение.

Основная идея: разделение интерфейса и реализации

По синтаксису класс аналогичен структуре. Класс определяет новый тип данных, объединяющих данные и код (функции обработки данных), и используется для описания объекта.

Реализация сокрытия информации происходит с помощью компонент: **public**, **private** (по умолчанию), **protected**.

Все переменные, объявленные с использованием ключевого слова public, являются доступными для всех функций в программе. Частные (private) данные доступны только в функциях, описанных в данном классе. Этим обеспечивается принцип инкапсуляции (пакетирования).

В общем случае доступ к объекту из остальной части программы осуществляется с помощью функций со спецификатором доступа public.

Знак **::** называется областью действия оператора. Он используется для информирования компилятора о том, что описываемая функция (в примере это summ) принадлежит классу, имя которого расположено слева от знака ::.

class kls

{ int sm; // по умолчанию для класса предполагается

int m[5]; // атрибут private

public:

void inpt(int); // прототип функции ввода данных

int summ(); // прототип функции summ };

void kls::inpt(int i) // описание функции inpt

{ cin >> m[i]; }

int kls::summ() // описание функции summ

{ sm=0; // инициализация компоненты sm класса

for(int i=0; i<5; i++) sm+=m[i];

return sm; }

В приведенном примере описан класс, для которого задан пустой список объектов. В функции main() объявлены два объекта описанного класса. При описании класса в него включаются прототипы функций для обработки данных класса. Текст самой функции может быть записан как внутри описания класса

(функция inpt()), так и вне его (функция summ()).

# **Конструктор с параметрами, конструктор по умолчанию**

В языке С++ имеется возможность одновременно с описанием (созданием) объекта выполнять и его инициализацию. Эти действия выполняются специальной функцией, принадлежащей этому классу. Эта функция носит специальное название: конструктор. Название этой функции всегда должно совпадать с именем класса, которому она принадлежит. **Конструктор** представляет собой обычную функцию, имя которой совпадает с именем класса, в котором он объявлен и используется. Он никогда не должен возвращать никаких значений. Количество и имена фактических параметров в описании функции конструктора зависят от числа полей, которые будут инициализированы при объявлении объекта (экземпляра) данного класса. Например: class A{ int a,b; A(int a1,int a2){ a=a1; b=b1; } }

Наряду с перечисленными выше формами записи конструктора существует конструктор, либо не имеющий параметров, либо все аргументы которого заданы по умолчанию – конструктор по умолчанию.

Каждый класс может иметь только один конструктор по умолчанию. Более того, если при объявлении класса в нем отсутствует явно описанный конструктор, то компилятором автоматически генерируется конструктор по умолчанию. Конструктор по умолчанию используется при создании объекта без инициализации его, а также незаменим при создании массива объектов. Если при этом конструкторы с параметрами в классе есть, а конструктора по умолчанию нет, то компилятор зафиксирует синтаксическую ошибку.

Отметим основные свойства и правила использования конструкторов:

- конструктор – функция, имя которой совпадает с именем класса, в кото-

ром он объявлен;

конструктор предназначен для создания объекта (массива объектов) и инициализации его компонент-данных;

- конструктор вызывается, если в описании используется связанный с ним тип:

class cls{ . . . };

int main()

{ cls aa(2,.3); // вызывает cls :: cls(int,double)

extern cls bb; // объявление, но не описание, конструктор не вызывается }

- конструктор по умолчанию не требует никаких параметров;

- если класс имеет члены, тип которых требует конструкторов, то он может иметь их определенными после списка параметров для собственного конструктора. После двоеточия конструктор имеет список обращений к конструкторам типов, перечисленным через запятую;

- если конструктор объявлен в private-секции, то он не может быть явно

вызван (из main функции) для создания объекта класса.

{ int m;

char st[20];

public:

stroka(char \*st); // конструктор

~stroka(); // деструктор

void out(char);

int poisk(char); };

stroka::stroka(char \*s)

{ cout << "\n работает конструктор";

strcpy(st,s); }

stroka::~stroka(void)

{cout << "\n работает деструктор";}

void stroka::out(char c)

{ cout << "\n символ " << c << " найден в строке "<< st<<m<<” раз”; }

int stroka::poisk(char c)

{ m=0;

for(int i=0;st[i]!='\0';i++)

if (st[i]==c) m++;

return m;

Все функции-компоненты класса stroka объявлены со спецификатором

public и, следовательно, являются глобальными и могут быть вызваны из функции main() . Вызов функции осуществляется с использованием префикса str.

str.poisk(c);

str.out(c);

# **Деструктор**

Противоположным по отношению к конструктору является деструктор − функция, приводящая к разрушению объекта соответствующего класса и возвращающая системе область памяти, выделенную конструктором. Деструктор имеет имя, аналогичное имени конструктора, но перед ним ставится знак ~: ~kls(void){} или ~kls(){} // функция-деструктор.

Далее выделим основные правила использования деструкторов:

- имя деструктора совпадает с именем класса, в котором он объявлен с

префиксом ~;

- деструктор не возвращает значения (даже типа void);

- деструктор не наследуется в производных классах;

- деструктор не имеет параметров (аргументов);

- в классе может быть только один деструктор;

- деструктор может быть виртуальным (виртуальной функцией);

- невозможно получить указатель на деструктор (его адрес);

- если деструктор отсутствует в описании класса, то он автоматически ге-

нерируется компилятором (с атрибутом public);

- библиотечная функция exit вызывает деструкторы только глобальных

объектов;

- библиотечная функция abort не вызывает никакие деструкторы.

# **Конструктор копирования**

Необходимость использования конструктора копирования вызвана тем, что объекты наряду со статическими могут содержать и динамические данные. В то же время, например, при передаче объекта в качестве параметра функции в ней создается локальная (в пределах функции) копия этого объекта. При этом указатели обоих объектов будут содержать один и тот же адрес области памяти. При выводе локального объекта из поля видимости функции для его разрушения вызывается деструктор. В функцию деструктора входит также освобождение динамической памяти, адрес которой содержит указатель. При окончании работы программы (при вызове деструкторов) производится повторная попытка освободить уже освобожденную ранее память. Это приводит к ошибке. Для устранения этого в класс необходимо добавить конструктор копирования, который в качестве единственного параметра получает ссылку на объект класса. Общий вид конструктора копирования имеет следующий вид: имя\_класса (const имя\_класса & );

Пример: String::String(const String& other)

{size = other.size;

ss = new char[size + 1]; strcpy\_s(ss,size+1,other.ss);}

Следует также отметить, что конструктор копирования вызывается в случае, если локальный объект в функции должен быть возвращен, используя оператор return. В этом случае происходит его копирование во временный объект и вызов деструктора. Временный объект возвращается в точку вызова функции.

class cls

{ char \*str;

int dl;

// . . . другие данные класса

public:

cls (); // конструктор по умолчанию

cls(cls &); // копирующий конструктор

~cls(); // деструктор

// . . . другие методы класса };

cls::cls ()

{ dl=10;

str=new char[dl]; }

cls::cls(cls & obj1) // копирующий конструктор из obj1 в obj

{ dl=obj1.dl; // копирование длины строки

str=new char[dl]; // выделение памяти “под” строку длиной dl

strcpy(str,obj1.str); // копирование строки }

cls::~cls()

{ delete [] str;

cout<<"деструктор"<<endl; }

void fun(cls obj1)

{ // код функции

cout<<" выполняется функция "<<endl; }

void main(void)

{ cls obj;

fun(obj);

Если для класса конструктор копирования явно не описан, то компилятор

сгенерирует его. При этом значения компоненты-данного одного объекта будут

скопированы в компоненту-данное другого объекта

# **Конструктор explicit**

В С++ компилятор для конструктора с одним аргументом может автоматически выполнять неявные преобразования. В результате этого тип, получаемый конструктором, преобразуется в объект класса, для которого определен данный конструктор.

#include <iostream>

using namespace std;

class array // класс-массив целых чисел

{ int size; // размерность массива

int \*ms; // указатель на массив

public: array(int = 1); ~array();

friend void print(const array&); };

array::array(int kl) : size(kl)

{ cout<<"работает конструктор"<<endl;

ms=new int[size]; // выделение памяти для массива

for(int i=0; i<size; i++) ms[i]=0; // инициализация }

array::~array()

{ cout<<"работает деструктор"<<endl;

delete [] ms; }

void print(const array& obj)

{ cout<<"выводится массив размерностью"<<obj.size<<endl;

for(int i=0; i<obj.size;i++)

cout<<obj.ms[i];

cout<<endl; }

void main()

{ array obj(10);

print(obj); // вывод содержимого объекта

obj print(5); // преобразование 5 в array и вывод }

В результате выполнения программы получим:

работает конструктор выводится массив размерностью 10

0 0 0 0 0 0 0 0 0 0

работает конструктор

выводится массив размерностью 5

0 0 0 0 0

работает деструктор

работает деструктор

В данном примере в инструкции:

array obj(10);

определяется объект obj и для его создания (и инициализации) вызывается конструктор array(int). Далее в инструкции:

print(obj); // вывод содержимого объекта obj

выводится содержимое объекта obj, используя friend-функцию print(). При выполнении инструкции:

print(5); // преобразование 5 в array и вывод

компилятором не находится функция print(int) и выполняется проверка на наличие в классе array конструктора, способного выполнить преобразование в объект класса array. Так как в классе array имеется конструктор array(int), а точнее, просто конструктор с одним параметром, то такое преобразование возможно (создается временный объект, содержащий массив из пяти чисел).

В некоторых случаях такие преобразования являются нежелательными или, возможно, приводящими к ошибке. В С++ имеется ключевое слово explicit для подавления неявных преобразований. Конструктор, объявленный как explicit:

explicit array(int = 1);

не может быть использован для неявного преобразования. В этом случае компилятором (в частности Microosft C++) будет выдано сообщение об ошибке:

Compiling...

error C2664: 'print' : cannot convert parameter 1 from 'const int' to 'const class array &' Reason: cannot convert from 'const int' to 'const class array'

No constructor could take the source type, or constructor overload resolution was ambiguous

Если необходимо при использовании explicit-конструктора все же создать массив и передать его в функцию print, то надо использовать инструкцию

print(array(5));

Правильно сконструировав классы, можно достичь, чтобы создание объектов было разрешено, а нежелательные неявные преобразования типов запрещены. Рассмотрим пример, в котором целочисленный размер массива может быть передан в качестве параметра конструктору и недопустимы преобразования целых чисел во временный объект.

class array // класс-массив целых чисел

{ public:

class array\_size // класс-размер массива

{ public:

array\_size(int \_kl):kl(\_kl){}

int size() const { return kl;}

private:

int kl; };

array(int n,int m) { this->n=n;

ms=new int[this->n];

for(int i=0;i<this->n;i++) ms[i]=m; }

array(array\_size \_size)

{ n=\_size.size();

ms=new int[n]; }

~array(){ delete [] ms;}

private:

int n;

int \*ms; };

main()

{ array a(1); . . . };

Компилятор для объекта a генерирует вызов конструктора array(int). Но такого конструктора не существует. Компиляторы могут преобразовать аргумент типа int во временный объект array\_size, поскольку в классе array\_size имеется конструктор с одним параметром. Это обеспечивает успешное создание объекта.

# **21. Указатель this**

В памяти для каждого располагаемого объекта создается скрытый указатель, адресующий начало выделенной под объект области памяти. Получить значение этого указателя в компонентах-функциях можно посредством ключевого слова this. Для любой функции, принадлежащей классу my\_class, указатель this неявно объявлен так: my\_class \*const this;

m\_cl \* m\_cl::inpt() // функция ввода

{ for(int i=0;i<3;i++) cin >> a[i];

return this; // возврат скрытого указателя this} // (адреса начала объекта)

Отметим основные **правила** использования this-указателей:

- каждому объявляемому объекту соответствует свой скрытый this указатель;

- this-указатель может быть использован только для нестатической функции;

- this указывает на начало своего объекта в памяти;

- this не надо дополнительно объявлять;

- this передается как скрытый аргумент во все нестатические (не имеющие спецификатора static) компоненты-функции своего объекта;

- указатель this − локальная переменная и недоступна за пределами объекта;

- обращаться к скрытому указателю можно this или \*this

Varyа, [05.01.2024 21:20]

class m\_cl

{ int a[3];

public:

m\_cl srt(); // функция упорядочивания информации в массиве

m\_cl \*inpt(); // функция ввода чисел в массив

void out(); // вывод информации о результате сортировки };

m\_cl m\_cl::srt() // функция сортировки

{ for(int i=0;i<2;i++)

for(int j=i;j<3;j++)

if (a[i]>a[j]) {a[i]=a[i]+a[j]; a[j]=a[i]-a[j]; a[i]=a[i]-a[j];}

return \*this; // возврат содержимого объекта, на который } // указывает указатель this

m\_cl \* m\_cl::inpt() // функция ввода

{ for(int i=0;i<3;i++)

cin >> a[i];

return this; // возврат скрытого указателя this } // (адреса начала объекта)

void m\_cl::out()

{ cout << endl;

for(int i=0;i<3;i++)

cout << a[i] << ' '; }

main()

{ m\_cl o1,o2; // описание двух объектов класса m\_cl

o1.inpt()->srt().out(); // вызов компонент-функций первого объекта

o2.inpt()->srt().out(); // вызов компонент-функций второго объекта

return 1; }

Вызов компонент-функций для каждого из созданных объектов осущест-

вляется

o1.inpt()->srt().out

# **Встроенные функции (спецификатор inline)**

В ряде случаев в качестве компонент класса используются достаточно простые функции. Вызов функции означает переход к памяти, в которой расположен выполняемый код функции. Команда перехода занимает память и требует времени на ее выполнение, что иногда существенно превышает затраты памяти на хранение кода самой функции. Для таких функций целесообразно поместить код функции вместо выполнения перехода к функции. В этом случае при выполнении функции (обращении к ней) выполняется сразу ее код. Функции с такими свойствами являются встроенными. Т.е. если описание компоненты функции включается в класс, то такая функция называется встроенной. Например:

{ class stroka

char str[100];

public:

……..

int size()

{ for (int i=0; \*(str+i); i++);

return i; }};

В описанном примере функция size() – встроенная. Если функция, объявленная в классе, а описанная за его пределами, должна быть встроенной, то она описывается со спецификатором inline:

class stroka

{ char str[100];

public:

……..

int size(); };

inline int stroka ::size()

{ for(int i=0; str[i]; i++);

return i; }

Спецификация inline может быть проигнорирована компилятором, поскольку иногда введение встроенных функций оказывается невозможным или нежелательным.

# **Организация внешнего доступа к локальным компонентам класса (спецификатор friend)**

Основное правило ООП – данные объекта защищены от воздействий извне и доступ к ним можно получить только с помощью методов объекта. Но бывают такие случаи, когда нам необходимо организовать доступ к данным объекта, не используя его функции. Конечно, можно добавить новую public-функцию к классу для получения прямого доступа к внутренним переменным. Однако в большинстве случаев интерфейс объекта реализует определенные операции, и новая функция может оказаться излишней. В то же время иногда возникает необходимость организации прямого доступа к внутренним (локальным) данным двух разных объектов из одной функции. При этом в С++ одна функция не может быть компонентой двух различных классов. Для реализации этого в С++ введен спецификатор friend. Если некоторая функция определена как friend-функция для некоторого класса, то она:

- не является компонентой-функцией этого класса;

- имеет доступ ко всем компонентам этого класса (private, public и protected).

Ниже рассматривается пример, когда внешняя функция получает доступ к внутренним данным класса.

#include <iostream> using namespace std;

class kls { int i,j;

public: kls(int i,int J) : i(I),j(J) {} // конструктор int max() {return i>j? i : j;} // функция-компонента класса kls friend double fun(int, kls&); // friend-объявление внешней функции fun };

double fun(int i, kls &x) // внешняя функция { return (double)i/x.i;

} main() { kls obj(2,3);

cout << obj.max() << endl;

сout << fun(3,obj) << endl;

return 1;}

Функции со спецификатором friend, не являясь компонентами класса, не имеют и, следовательно, не могут использовать this-указатель. Следует также отметить ошибочность следующей заголовочной записи функции double kls :: fun(int i,int j), так как fun не является компонентой-функцией класса kls.

В общем случае friend-функция является глобальной независимо от секции, в которой она объявлена (public, protected, private), при условии, что она не объявлена ни в одном другом классе без спецификатора friend. Функция friend, объявленная в классе, может рассматриваться как часть интерфейса класса с внешней средой.

Вызов компоненты-функции класса осуществляется с использованием операции доступа к компоненте (.) или (->). Вызов же friend-функции производится по ее имени (так как friend-функции не являются компонентами класса). Следовательно, как это будет показано далее, в friend-функции не передается this-указатель и доступ к компонентам класса выполняется либо явно (.), либо косвенно (->).

Компонента-функция одного класса может быть объявлена со спецификатором friend для другого класса:

class X{ ……….

void fun (….); };

class Y{ ……….

friend void X:: fun (….); };

В приведенном фрагменте функция fun() имеет доступ к локальным ком-понентам класса Y. Запись вида friend void X:: fun (….) говорит о том, что функ-ция fun принадлежит классу Х, а спецификатор friend разрешает доступ к ло-кальным компонентам класса Y (так как она объявлена со спецификатором в классе Y).

Ниже приведен пример программы расчета суммы двух налогов на зарплату.

#include <iostream>

#include <iomanip.h>

using namespace std;

#include <string.h> class nalogi; // неполное объявление класса nalogi

class work { char s[20]; // фамилия работника

int zp; // зарплата

public:

float raschet(nalogi); // компонента-функция класса work

void inpt()

{ cout << "вводите фамилию и зарплату" << endl;

cin >> s >>zp;

} work(){}

~work(){}; };

class nalogi {

float pd, // подоходный налог

st; // налог на соцстрахование

friend float work::raschet(nalogi); // friend-функция класса nalogi

public:

nalogi(float f1,float f2) : pd(f1),st(f2){};

~nalogi(void){}; };

float work::raschet(nalogi nl)

{ cout << s << setw(6) << zp <<endl; // доступ к данным класса work

cout << nl.pd << setw(8) << nl.st <<endl; // доступ к данным класса nalogi

return zp\*nl.pd/100+zp\*nl.st/100; }

int main()

{ nalogi nlg((float)12,(float)2.3); // описание и инициализация объекта

work wr[2]; // описание массива объектов

for(int i=0;i<2;i++) wr[i].inpt(); // инициализация массива объектов cout<<setiosflags(ios::fixed)<<setprecision(3)<<endl;

cout << wr[0].raschet(nlg) << endl; // расчет налога для объекта wr[0]

cout << wr[1].raschet(nlg) << endl; // расчет налога для объекта wr[1]

return 0; }

Следует отметить необходимость выполнения неполного (предварительного) объявления класса nalogi, так как в прототипе функции raschet класса work используется объект класса nalogi, объявляемого далее. В то же время полное объявление класса nalogi не может быть выполнено ранее (до объявления класса work), так как в нем содержится friend-функция, описание которой должно быть выполнено до объявления friend-функции. В противном случае компилятор выдаст ошибку. Если функция raschet в классе work также будет использоваться со спецификатором friend, то приведенная выше программа будет выглядеть следующим образом:

#include "iostream.h"

#include "string.h"

#include "iomanip.h"

class nalogi; // предварительное объявление класса nalogi

class work

{ char s[20]; // фамилия работника

int zp; // зарплата

public:

friend float raschet(work,nalogi); // friend-функция класса work

void inpt()

{ cout << "вводите фамилию и зарплату" << endl;

cin >> s >>zp;

} work(){} // конструктор по умолчанию

~work(){} // деструктор по умолчанию };

class nalogi

{ float pd, // подоходный налог

st; // налог на соцстрахование

friend float raschet(work,nalogi); // friend-функция класса nalogi

public:

nalogi(float f1,float f2) : pd(f1),st(f2){};

~nalogi(void){}; };

float raschet(work wr,nalogi nl)

{ cout << wr.s << setw(6) << wr.zp <<endl;// доступ к данным класса work

cout << nl.pd << setw(8) << nl.st <<endl;// доступ к данным класса nalogi

return wr.zp\*nl.pd/100+wr.zp\*nl.st/100; }

int main()

{ nalogi nlg((float)12,(float)2.3); // описание и инициализация объекта

work wr[2];

for(int i=0;i<2;i++) wr[i].inpt(); // инициализация массива объектов cout<<setiosflags(ios::fixed)<<setprecision(3)<<endl;

cout << raschet(wr[0],nlg) << endl; // расчет налога для объекта wr[0]

cout << raschet(wr[1],nlg) << endl; // расчет налога для объекта wr[1]

return 0; }

Отметим основные свойства и правила использования спецификатора friend:

- friend-функции не являются компонентами класса, но имеют доступ ко всем его компонентам независимо от их атрибута доступа;

- friend-функции не имеют указателя this;

- friend-функции не наследуются в производных классах;

- отношение friend не является ни симметричным (т.е. если класс А есть friend классу В, то это не означает, что В также является friend классу А), ни транзитивным (т.е. если A friend B и B friend C, то не следует, что A friend C);

- друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

# **24. Дружественные классы**

Все функции одного класса можно объявить со спецификатором friend по отношению к другому классу следующим образом:

class X{

………..

friend class Y; };

class Y{ };

В этом случае все компоненты-функции класса Y имеют спецификатор friend для класса Х (имеют доступ к компонентам класса Х).

- friend-функции не наследуются в производных классах;

- отношение friend не является ни симметричным (т.е. если класс А friend классу В, то это не означает, что В также является friend классу А), ни транзитивным (т.е. если A friend B и B friend C, то не следует, что A friend C);

- друзьями класса можно определить перегруженные функции.

Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

#include <iostream> using namespace std;

class A

{ int i; // компонента-данное класса А

public:

friend class B; // объявление класса В другом класса А

A():i(1){} // конструктор

void f1\_A(B &); // объявление метода, оперирующего данными обоих классов };

class B

{ int j; // компонента-данное класса В

public:

friend A; // объявление класса А другом класса В

B():j(2){} // конструктор

void f1\_B(A &a){ cout<<a.i+j<<endl;} // описание метода, оперирующего данными обоих классов };

void A :: f1\_A(B &b)

{ cout<<i<<' '<<b.j<<endl;}

void main()

{ A aa;

B bb;

aa.f1\_A(bb);

bb.f1\_B(aa);}

Результат выполнения программы:

1 2

3

В объявлении класса А содержится инструкция friend class B, являющаяся и предварительным объявлением класса В, и объявлением класса В дру-жественным классу А. Отметим также необходимость описания функции f1\_A после явного объявления класса В (в противном случае не может быть создан объект b еще не объявленного типа).

Отметим основные свойства и правила использования спецификатора friend:

- friend-функции не являются компонентами класса, но имеют доступ ко всем его компонентам независимо от их атрибута доступа;

- friend-функции не имеют указателя this;

- friend-функции не наследуются в производных классах;

- отношение friend не является ни симметричным (т.е. если класс А есть friend классу В, то это не означает, что В также является friend классу А), ни транзитивным (т.е. если A friend B и B friend C, то не следует, что A friend C);

- друзьями класса можно определить перегруженные функции. Каждая перегруженная функция, используемая как friend для некоторого класса, должна быть явно объявлена в классе со спецификатором friend.

# **25. Вложенные классы**

Один класс может быть объявлен в другом классе, в этом случае внутренний класс называется вложенным:

class ext\_class

{ class int\_cls

{ • • • };

public:

• • • };

Класс int\_class является вложенным по отношению к классу ext\_class (внешний).

Доступ к компонентам вложенного класса, имеющим атрибут private, возможен только из функций вложенного класса и из функций внешнего класса, объявленных со спецификатором friend во вложенном классе.

#include <iostream>

using namespace std;

class cls1 // внешний класс

{ class cls2 // вложенный класс

{ int b; // все компоненты private для cls1 и cls2

cls2(int bb) : b(bb){} // конструктор класса cls2 };

public: // public секция для cls1

int a;

class cls3 // вложенный класс

{ int c; // private для cls1 и cls3

public: // public-секция для класса cls3

cls3(int cc): c(cc) {} // конструктор класса cls3 };

cls1(int aa): a(aa) {} // конструктор класса cls };

void main()

{ cls1 aa(1980); // верно

cls1::cls2 bb(456); // ошибка cls2 cannot access private

cls1::cls3 cc(789); // верно

cout << aa.a << endl; // верно

cout << cc.c << endl; // ошибка 'c' : cannot access private member declared in class 'cls1::cls3' }

В приведенном тексте программы инструкция cls1::cls2 bb(456) является ошибочной, так как объявление класса cls2 и его компонента находятся в сек-ции private. Для устранения ошибки при описании объекта bb необходимо из-менить атрибуты доступа для класса cls2 следующим образом:

public:

class cls2

{ int b; // private-компонента

public:

cls2(int bb) : b(bb){} };

Пример доступа к private-компонентам вложенного класса из функций внешнего класса, объявленных со спецификатором friend, приводится ниже.

#include <iostream>

using namespace std;

class cls1 // внешний класс

{ int a;

public:

cls1(int aa): a(aa) {} class cls2; // неполное объявление класса

void fun(cls1::cls2); // функция получает объект класса cls2

class cls2 // вложенный класс

{ int c;

public:

cls2(int cc) : c(cc) {}

friend void cls1::fun(cls1::cls2); // функция, дружественная

int pp(){return c;} // классу cls1 }; };

void cls1::fun(cls1::cls2 dd) {cout << dd.c << endl << a;}

int main()

{ cls1 aa(123);

cls1:: cls2 cc(789);

aa.fun(cc);

return 0;}

Внешний класс cls1 содержит public-функцию fun (cls1::cls2 dd), где dd есть объект, соответствующий классу cls2, вложенному в класс cls1. В свою очередь в классе cls2 имеется friend-функция friend void cls1::fun(cls1::cls2 dd), обеспечивающая доступ функции fun класса cls1 к локальной компоненте с класса cls2.

#include <iostream>

using namespace std;

#include <string.h>

class ext\_cls // внешний класс

{ int gd; // год рождения

double zp; // з/плата

class int\_cls1 // первый вложенный класс

{ char \*s; // фамилия

public:

int\_cls1(char \*ss) // конструктор 1-го вложенного класса

{ s=new char[20];

strcpy(s,ss); }

~int\_cls1() // деструктор 1-го вложенного класса

{ delete [] s;}

void disp\_int1() {cout << s<<endl;} };

public:

class int\_cls2 // второй вложенный класс

{ char \*s; // фамилия

public:

int\_cls2(char \*ss) //конструктор 2-го вложенного класса

{ s=new char[20];

strcpy(s,ss); }

~int\_cls2() // деструктор 2-го вложенного класса

{ delete [] s;}

void disp\_int2() {cout << s << endl;} };

// public функции класса ext\_cls

ext\_cls(int god,double zpl):gd(god),zp(zpl){}// конструктор внешнего класса

int\_cls1 \*addr(int\_cls1 &obj){return &obj;} // возврат адреса объекта obj

void disp\_ext()

{ int\_cls1 ob("Иванов"); // создание объекта вложенного класса

addr(ob)->disp\_int1(); // вывод на экран содержимого объекта ob

cout <<gd<<’ ’<< zp<<endl; }};

int main()

{ ext\_cls ext(1980,230.5);

// ext\_cls::int\_cls1 in1("Петров"); // неверно, т.к. int\_cls1 имеет атрибут private

ext\_cls::int\_cls2 in2("Сидоров");

in2.disp\_int2();

ext.disp\_ext();

return 0;

В результате выполнения программы получим:

Сидоров

Иванов

1980 230.5

В строке in2.disp\_int2() (main функции) для объекта вложенного класса вызывается компонента-функция ext\_cls::int\_cls2::disp\_int2() для вывода содер-жимого объекта in2 на экран. В следующей строке ext.disp\_ext() вызывается функция ext\_cls::disp\_ext(), в которой создается объект вложенного класса int\_cls1. Затем, используя косвенную адресацию, вызывается функция ext\_cls::int\_cls1::disp\_int1(). Далее выводятся данные, содержащиеся в объекте внешнего класса.

# **26. Static-члены (данные) класса**

Компоненты-данные могут быть объявлены с модификатором класса памяти static. Класс, содержащий static компоненты-данные, объявляется как глобальный (локальные классы не могут иметь статических членов). Static-компонента совместно используется всеми объектами этого класса и хранится в одном месте. Статическая компонента глобального класса должна быть явно определена в контексте файла. Использование статических компонент-данных класса продемонстрируем на примере программы, выполняющей поиск введенного символа в строке.

#include "string.h"

#include "iostream.h"

enum boolean {fls,tru};

class cls

{ char \*s;

public:

static int k; // объявление static-члена в объявлении класса

static boolean ind;

void inpt(char \*,char);

void print(char); };

int cls::k=0; // явное определение static-члена в контексте файла

boolean cls::ind;

void cls::inpt(char \*ss,char c)

{ int kl; // длина строки

cin >> kl;

ss=new char[kl]; // выделение блока памяти под строку

cout << "введите строку\n";

cin >> ss;

for (int i=0; \*(ss+i);i++)

if(\*(ss+i)==c) k++; // подсчет числа встреч буквы в строке

if (k) ind=tru; // ind==tru − признак того, что буква есть в строке

delete [] ss; // освобождение указателя на строку}

void cls::print(char c)

{ cout << "\n число встреч символа "<< c <<"в строках = " << k; }

void main()

{ cls c1,c2;

char c;

char \*s;

cls::ind=fls;

cout << "введите символ для поиска в строках";

cin >> c;

c1.inpt(s,c);

c2.inpt(s,c);

if(cls::ind) c1.print(c);

else cout << "\n символ не найден"; }

Объявление статических компонент-данных задает их имена и тип, но не инициализирует значениями. Присваивание им некоторых значений выполняется в программе (вне объекта).

В функции main() использована возможная форма обращения к static-компоненте cls::ind (имя класса :: идентификатор), которая обеспечивается тем, что идентификатор имеет видимость public. Это дальнейшее использование оператора разрешения контекста “::”. Отметим основные правила использования статических компонент:

- статические компоненты будут одними для всех объектов данного класса, т.е. ими используется одна область памяти;

- статические компоненты не являются частью объектов класса;

- объявление статических компонент-данных в классе не является их описанием. Они должны быть явно описаны в контексте файла;

- локальный класс не может иметь статических компонент;

- к статической компоненте st класса cls можно обращаться cls::st, независимо от объектов этого класса, а также при помощи операторов . и -> при использовании объектов этого класса;

- статическая компонента существует даже при отсутствии объектов этого класса;

- статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

# **27. Компоненты-функции static и const**

В С++ компоненты-функции могут использоваться с модификатором static и const. Обычная компонента-функция, вызываемая object.function(a,b); имеет явный список параметров a и b и неявный список параметров, состоящий из компонент данных переменной object. Неявные параметры можно представить, как список параметров, доступных через указатель this. Статическая (static) компонента-функция не может обращаться к любой из компонент посредством указателя this. Компонента-функция const не может изменять неявные параметры.

#include <iostream>

using namespace std;

class cls

{ int kl; // количество изделий

double zp; // зарплата на производство одного изделия

double nl1,nl2; // два налога на з/пл

double sr; // количество сырья на производство одного изделия

static double cs; // цена сырья на одно изделие

public:

cls(){} // конструктор по умолчанию

~cls(){} // деструктор

void inpt(int);

static void vvod\_cn(double);

double seb() const; };

double cls::cs; // явное определение static-члена в контексте файла

void cls::inpt(int k)

{ kl=k;

cout << "Введите з/пл и 2 налога";

cin >> nl1 >> nl2 >> zp; }

void cls::vvod\_cn(double c)

{ cs=c; // можно обращаться в функции только к static-компонентам;}

double cls::seb() const

{return kl\*(zp+zp\*nl1+zp\*nl2+sr\*cs); //в функции нельзя изменить ни один неявный параметр(kl zp nl1 nl2 sr) }

void main()

{ cls c1,c2;

c1.inpt(100); // инициализация первого объекта

c2.inpt(200); // инициализация второго объекта

cls::vvod\_cn(500.);

cout << "\nc1" << c1.seb() << "\nc2" << c2.seb() << endl; }

Ключевое слово static не должно быть включено в описание объекта статической компоненты класса. Так, в описании функции vvod\_cn отсутствует ключевое слово static. В противном случае возможно противоречие между static-компонентами класса и внешними static-функциями и переменными. Следующий далее пример демонстрирует доступ к static данным класса из различных функций.

#include <iostream>

using namespace std;

class cls

{ static int i;

int j;

public:

static int k; // объявление static-члена в объявлении класса

void f1();

static void f2(); };

int cls::k=0; // явное определение static-члена в контексте файла

int cls::i=0;

void cls::f1() // из функции класса возможен доступ к private и public static данным

{ cout << ++k<<' '<<++i<< endl;}

void cls::f2() // из static функции класса возможен доступ к private и public static данным

{ cout <<++k<<' '<<++i<<endl;}

void f3() // из внешней функции возможен доступ только к public static данным

{cout<<++cls::k<<endl;}

void main()

{ cls obj;

cout << cls::k<<endl; // возможен доступ только к public static данным obj.f1();

cls::f2();

f3();}

Результат работы программы:

0

1 1

2 2

3

Функции класса, объявленные со спецификатором const, могут быть вы-званы для объекта со спецификатором const, а функции без спецификатора const − не могут.

const cls c1;

cls c2;

c1.inpt(100); // неверный вызов

c2.inpt(100); // правильный вызов функции

c1.seb(); // правильный вызов функции

Для функций со спецификатором const указатель this имеет следующий тип:

const имя\_класса \* const this;

Следовательно, нельзя изменить значение компоненты объекта через указатель this без явной записи. Рассмотрим это на примере функции seb.

double cls::seb() const

{ ((cls \*)this)->zp--; // возможная модификация неявного параметра zp посредством явной записи this-указателя

return kl\*(zp+zp\*nl1+zp\*nl2+sr\*cs);}

Если функция, объявленная в классе, описывается отдельно (вне класса), то спецификатор const должен присутствовать как в объявлении, так и в описании этой функции.

Основные свойства и правила использования static- и const-функций:

- статические компоненты-функции не имеют указателя this, поэтому об-ращаться к нестатическим компонентам класса можно только с использованием . или ->;

- не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов, чтобы при этом одна была статической, а другая нет;

- статические компоненты-функции не могут быть виртуальными.

# **28. Proxi-классы**

Реализация скрытия данных и интерфейса некоторого класса может быть выполнена посредством использования proxi-класса. Proxi-класс позволяет кли-ентам исходного класса использовать этот класс, не имея доступа к деталям его реализации. Реализация proxi-класса предполагает следующую общую структуру:

- реализация исходного класса, компоненты которого требуется скрыть;

- реализация proxi-класса для доступа к компонентам исходного класса;

- функция, в которой вызываются компоненты proxi-класса

// заголовочный файл cls.h для класса cls

class cls

{ int val;

public:

cls(int);

set(int);

int get() const; };

// файл реализации для класса cls

#include "cls.h"

cls :: cls(int v) {val=v;}

cls :: set(int v) {val=v;}

int cls :: get() const {return val;}

// заголовочный файл prox.h для proxi-класса prox

class cls; // предварительное объявление класса cls

class prox

{ cls \*pr; // для этого и требуется предварительное объявление

public:

prox(int);

set(int);

int get() const;

~prox(); };

// файл реализации для proxi-класса prox

#include "prox.h"

#include "cls.h"

prox :: prox(int vv) {pr=new cls(vv);}

prox :: set(int vv){pr->set(vv);}

int prox :: get() const {return pr->get();}

prox :: ~prox(){delete pr;}

// программа скрытия данных класса cls посредством proxi-класса prox

#include <iostream>

using namespace std;

#include "prox.h"

int main()

{ prox obj(1);

cout<<” Значение val класса cls = ”<<obj.get()<<endl;

obj.set(2);

cout<<” Значение val класса cls = ”<<obj.get()<<endl; }

В результате выполнения программы получим:

Значение val класса cls = 1

Значение val класса cls = 2

Исходный класс cls содержит один private-компонент val и методы, которые требуется скрыть от клиента, взаимодействующего с main()-функцией. В то же время клиент должен иметь возможность взаимодействовать с классом cls. Для этого используется класс prox, реализация которого содержит public-интерфейс, аналогичный интерфейсу класса cls, и единственный private-компонент – указатель на объект класса cls. Класс prox является proxi-классом для класса cls. Так как в определении класса prox используется только указатель на объект класса cls, то включение заголовочного файла класса cls посредством инструкции #include необязательно. Достаточно объявить класс как тип данных путем предварительного объявления.

Файл реализации cls.cpp включает заголовочный файл cls.h с объявлением класса cls. Файл реализации prox.h класса prox является единственным файлом, включающим файл реализации cls.cpp, а следовательно, и cls.h. Файл prox.cpp является доступным клиенту только как скомпилированный объектный код и, следовательно, клиент не может видеть взаимодействия между proxi-классом и классом cls.

В main()-функцию включается только файл реализации prox.cpp, при этом отсутствует указание на существование класса cls. Следовательно, private-данные класса cls скрыты от клиента.

# **29. Ссылки**

В С++ известны три способа передачи данных в функцию: по значению, посредством указателя и используя ссылки. При передаче параметров в функцию они помещаются в стековую память. В отличие от стандартных типов данных (char, int, float и др.) объекты обычно требуют много больше памяти, при этом стековая память может существенно

увеличиться. Для уменьшения объема передаваемой через стек информации в С++ используются указатели. В языке С++ наряду с использованием механизма указателей имеется возможность использовать неявные указатели (ссылки). Ссылка, по существу, является не чем иным, как вторым именем некоторого объекта.

Формат объявления ссылки имеет вид

тип & имя\_ссылки = инициализатор.

Ссылку нельзя объявить без ее инициализации. То есть ссылаться всегда можно на некоторый существующий объект. Можно выделить следующие различия ссылок и указателей. Во-первых, невозможность существования нулевых ссылок подразумевает, что корректность их не требуется проверять. А при использовании указателя требуется проверять его на ненулевое значение. Во-вторых, указатели могут указывать на различные объекты, а ссылка всегда на один объект, заданный при ее инициализации. Ниже приведен пример использования ссылки.

#include <iostream>

using namespace std;

#include <string.h>

class A

{ char s[80];

int i;

public :

A(char \*S,int I):i(I) { strcpy(s,S);}

~A(){}

void see() {cout<<s<<" "<<i<<endl;} };

int main()

{ A a("aaaaa",3),aa("bbbb",7);

A &b=a; // ссылка на объект класса А инициализирована значением а

cout<<"компоненты объекта :"; a.see();

cout<<"компоненты ссылки :"; b.see();

cout <<"адрес a="<<&a << " адрес &b= "<< &b << endl;

b=aa; // присвоение значений объекта aa ссылке b (и объекту a)

cout<<"компоненты объекта :"; a.see();

cout<<"компоненты ссылки :"; b.see();

int i=4,j=2;

int &ii=i; // ссылка на переменную i типа int

cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;

ii++; // увеличение значения переменной i

cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;

ii=j; // инициализация переменной i значением j

cout << "значение i= "<<i<<" значение ii= "<<ii<<endl;

}

В результате выполнения программы получим:

компоненты объекта : aaaaa 3

компоненты ссылки : aaaaa 3

адрес a= 0x\_\_ адрес &b= 0х\_\_

компоненты объекта : bbbbb 7

компоненты ссылки : bbbbb 7

значение i= 4 значение ii= 4

значение i= 5 значение ii= 5

значение i= 2 значение ii= 2

Из примера следует, что переменная и ссылка на нее имеют один и тот же

адрес в памяти. Изменение значения по ссылке приводит к изменению значения

переменной и наоборот.

Ссылка может также указывать на константу, в этом случае создается

временный объект, инициализируемый значением константы.

const int &j=4; // j инициализируется const-значением 4

j++; // ошибка l-value specifies const object

int k=j; // переменная k инициализируется значением

// временного объекта

Если тип инициализатора не совпадает с типом ссылки, то могут возник-

нуть проблемы с преобразованием данных одного типа к другому, например:

double f=2.5;

int &n=(int &)f;

cout<<”f=”<<f<<” n=”<<n; // результат f= 2.5 n=2

Адрес переменной f и ссылки n совпадают, но значения различаются, так

как структуры данных плавающего и целочисленного типов различны.

Можно создать ссылку на ссылку, например:

int k=1;

int &n=k; // n – ссылка на k (равно k)

n++; // значение k равно 2

int &nn=n; // nn – ссылка на ссылку n (переменную k)

nn++; // значение k равно 3

Значения адреса переменной k, ссылок n и nn совпадают, следовательно,

для ссылки nn не создается временный объект.

На применение переменных ссылочного типа накладываются некоторые

ограничения:

- ссылки не являются указателями;

- можно взять ссылку от переменной ссылочного типа;

- можно создать указатель на ссылку;

- нельзя создать массив ссылок;

- ссылки на битовые поля не допускаются.

# **30. Параметры ссылки**

Если требуется предоставить возможность функции изменять значения передаваемых в нее параметров, то в языке С они должны быть объявлены либо глобально, либо работа с ними в функции осуществляется через передаваемые в нее указатели на эти переменные. В С++ аргументы в функцию можно передавать также и через ссылку. Для этого при объявлении функции перед параметром ставится знак &.

#include <iostream>

using namespace std;

void fun1(int,int);

void fun2(int &,int &);

int main()

{ int i=1,j=2; // i и j – локальные параметры

cout << "\n адрес переменных в main() i = "<<&i<<" j = "<<&j;

cout << "\n i = "<<i<<" j = "<<j;

fun1(i,j);

cout << "\n значение i = "<<i<<" j = "<<j;

fun2(i,j);

cout << "\n значение i = "<<i<<" j = "<<j;

}

void fun1(int i,int j)

{ cout << "\n адрес переменных в fun1() i = "<<&i<<" j = "<<&j;

int a; // при вызове fun1 i и j из main() копируются

a=i; i=j; j=a; // в стек в переменные i и j при возврате в main()

} // они просто теряются

void fun2(int &i,int &j)

{ cout << "\n адрес переменных в fun2() i = "<<&i<<" j = "<<&j;

int a; // здесь используются ссылки на переменные i и j из

a=i; i=j; j=a; // main() (вторые их имена) и таким образом действия

// в функции производятся с теми же переменными i и j

}

В функции fun2 в инструкции

a=i;

не используется операция \*. При объявлении параметра-ссылки компилятор С++ определяет его как неявный указатель (ссылку) и обрабатывает соответствующим образом. При вызове функции fun2 ей автоматически передаются адреса переменных i и j. Таким образом, в функцию передаются не значения переменных, а их адреса, благодаря чему функция может модифицировать значения этих переменных. Будьте внимательны, при вызове функции fun2 знак & перед переменными i и j говорит о том, что в функцию будут переданы адреса этих переменных, а не о том, что используются ссылки на них.

# **31. Пространства имен**

При совпадении имен разных элементов в одной области действия часто возникает конфликт имен. Наиболее часто это возникает при использовании различных пакетов библиотек, содержащих, например, одноименные классы. Пространства имен используются для разделения глобального пространства имен, что позволяет уменьшить количество конфликтов.

# **32. Определение пространства имен**

Синтаксис пространства имен некоторым образом напоминает синтаксис структур и классов. После ключевого слова namespace следует необязательное имя пространства имен, затем описывается пространство имен, заключенное в фигурные скобки.

namespace NAME

{ int a;

doudle b;

char \*fun(char \*,int);

class CLS

{ . . .

public:

. . . }}

Далее, если обращение к элементам пространства имен производится вне контекста, его имя должно быть полностью квалифицировано, используя ::

NAME::b=2;

NAME:: fun(str,NAME:: a);

Внутри пространства имен можно поместить группу объявлений классов, типов и функций. Реализация функций пространства имен должна находиться вне самого пространства имен. Это позволит не только отделить реализацию функций от их объявления, но и избежать загромождения пространства имен. По существу, namespace определяет область видимости. Использование безымянного пространства имен (отсутствует имя пространства имен) позволяет определить уникальность объявленных в нем идентификаторов с областью видимости в пределах файла. Контексты пространства имен могут быть вложены.

namespace NAME1

{ int a;

namespace NAME2

{ int a;

int fun1(){return NAME1:: a}; // возвращается значение первого a

int fun2(){return a}; // возвращается значение второго a } }

NAME1::NAME2::fun1(); // вызов функции

Если в каком-то месте программы интенсивно используется некоторый контекст и все имена уникальны по отношению к нему, то можно сократить полные имена, объявив контекст текущим с помощью оператора using. Если элементы пространства имен будут интенсивно использоваться, то можно использовать ключевое слово using для упрощения доступа к ним. Ключевое слово using используется и как директива, и для объявления. Синтаксис слова using определяет, является ли оно директивой или объявлением.

# **33. Ключевое слово using как директива**

Инструкция using namespace имя позволяет предоставить все имена, объявленные в пространстве имен, для доступа в текущей области действия. Эта инструкция называется директивой using. Это позволит обращаться к этим именам без указания их полного имени, включающего название пространства имен.

#include <iostream>

using namespace std;

namespace NAME

{ int n1=1;

int n2=2; }

// int n1; приводит к неоднозначности в main для переменной n1

int main()

{ NAME::n1=3;

// n1=3; // error 'n1' : undeclared identifier

// n2=4; // error 'n2' : undeclared identifier

using namespace NAME; // далее n1 и n2 доступны

n2=4;

cout << n1 <<" "<< n2 << endl; // результат 3 4

{ n1=5;

n2=6;

cout << n1 <<" "<< n2 << endl; // результат 5 6 }

return 0; }

В результате выполнения программы получим:

3 4

5 6

Область действия директивы using распространяется на блок, в котором она использована, и на все вложенные блоки. Если одно из имен относится к глобальной области, а другое объявлено внутри пространства имен, то возникает неоднозначность. Это проявится только при использовании этого имени, а не при объявлении.

#include <iostream>

using namespace std;

В данном фрагменте стандартный заголовочный файл библиотеки ввода/вывода iostream не имеет расширения. Все содержимое этого файла определяется как часть namespace std.

Для достижения переносимости рекомендуется использовать директиву using, хотя и существуют компиляторы, не поддерживающие данную возможность. Основная проблема, которую призвана решить такая конструкция это независимость от ограничения на длину имени файла в различных операционных системах. Более того, компиляторы Microsoft последних версий вообще не поддерживают вариант с подключением файлов стандартной библиотеки с расширением .h, т.е. конструкция #include <iostream.h> в Visual C++ 7.1 не компилируется.

# **34. Ключевое слово using как объявление**

using как объявление синонимов типов позволяет создавать альтернативные имена для существующих типов данных.

#include <iostream>

using MyInt = int; // Создание синонима типа MyInt для типа int

int main() {

MyInt number = 42; // Использование синонима типа MyInt

std::cout << number << std::endl;

return 0;}

Когда вы объявляете имя из пространства имен с помощью using, вы делаете это для удобства использования в текущей области действия, без необходимости указывать полное квалифицированное имя каждый раз.

#include <iostream>

namespace MyNamespace {

int myNumber = 42;}

int main() {

{using MyNamespace::myNumber; // Объявление myNumber из пространства имен MyNamespace для текущего блока

std::cout << myNumber << std::endl; // Использование myNumber без указания полного имени пространства имен}

return 0;}

# **35. Псевдоним пространства имен**

Псевдоним пространства имен существует для того, чтобы назначить другое имя именованному пространству имен. Это особенно полезно, когда вы работаете с пространствами имен, имеющими длинные иерархические имена, и хотите сделать код более читабельным и компактным.

#include <iostream>

namespace VeryLongNamespaceName {

namespace SubNamespace {

void printMessage() {

std::cout << "Hello from VeryLongNamespaceName::SubNamespace!" << std::endl;}}}

int main() {

namespace VLN = VeryLongNamespaceName::SubNamespace; // Создание псевдонима пространства имен

VLN::printMessage(); // Использование псевдонима пространства имен

return 0;}

# **36. Наследование**

Наследование заключается в том, что один класс наследует некоторые свойства другого. Этот принцип предполагает использование базового класса, описывающего наиболее общие свойства ряда объектов. Производные классы включают в себя все черты базового класса, а также добавляют новые, характерные только для объектов данного класса. Спецификация описания производного класса имеет следующий синтаксис:

*class имя\_производного\_класса: [атрибут] имя\_базового\_класса {тело\_произв\_класса} [список объектов];*

#include <iostream>

// Базовый класс

class Animal {

public:

void eat() {

std::cout << "The animal is eating." << std::endl; }};

// Производный класс

class Dog : public Animal { // Ключевое слово "public" указывает на наследование от базового класса Animal

public:

void bark() {

std::cout << "The dog is barking." << std::endl;}};

int main() {

Dog dog;

dog.eat(); // Метод eat() унаследован от базового класса Animal

dog.bark(); // Метод bark() определен в производном классе Dog

return 0;}

[**37. Наследование (производные классы)**](file:///C:\Users\user\Downloads\Telegram%20Desktop\Вопросы%20к%20экзамену.docx#_Toc185879304)

В C++ производный класс, также известный как подкласс или наследник, создается на основе уже существующего базового класса. Производный класс наследует свойства и методы базового класса и может добавлять свои собственные свойства и методы.

В C++ существуют три атрибута доступа: public, protected и private. если базовый класс имеет атрибут public, то компоненты public и protected базового класса наследуются с атрибутами public и protected в производном классе. Компоненты private остаются private -компонентами базового класса; если базовый класс имеет атрибут protected, то компоненты public и protected базового класса наследуются с атрибутом protected в производном классе. Компоненты private остаются private -компонентами базового класса; если базовый класс имеет атрибут private, то компоненты public и protected базового класса наследуются с атрибутами private в производном классе. Компоненты private остаются private -компонентами базового класса.

Отмеченные типы наследования называются: внешним, защищенным и внутренним наследованием.

#include <iostream>

// Базовый класс

class Base {

public:

void publicMethod() {

std::cout << "This is a public method of the Base class." << std::endl;}

protected:

void protectedMethod() {

std::cout << "This is a protected method of the Base class." << std::endl;}

private:

void privateMethod() {

std::cout << "This is a private method of the Base class." << std::endl;}};

// Производный класс

class Derived : public Base {

public:

void accessBaseMembers() {

publicMethod(); // Доступ к публичному методу базового класса

protectedMethod(); // Доступ к защищенному методу базового класса

// privateMethod(); // Недоступно - приватный метод базового класса}};

int main() {

Derived derivedObj;

derivedObj.accessBaseMembers(); // Вызов метода производного класса, который обращается к членам базового класса

return 0;}

# **38. Конструкторы и деструкторы при наследовании**

При наследовании сначала вызывается конструктор базового класса, затем конструктор производного класса. В противоположность этому деструктор производственного класса вызывается перед вызовом деструктора базового класса.

*КонструкторПроизводногоКласса(СписокФормальныхАргументов)*

*: КонструкторБазовогоКласса (СписокФактическихАргументов)*

*{ // тело конструктора производного класса }*

*class A*

*{ protected:*

*int a;*

*char s[N];*

*public:*

*A(int a, const char\*s): a(a){strcpy\_s(this->s, s);} ;*

*~A();}*

*class B: public A*

*{ protected:*

*int b;*

*public:*

*B(int b, int a, const char\*s): A(a,s), b(b){};*

*~B():}*

# **39. Виртуальные функции**

Виртуальная ф-я - это ф-я, объявленная с ключевым словом virtual в базовом классе и переопределенная в 1 или нескольких производных от этого класса. При вызове объекта базового или производных классов динамически (во время выполнения программы) определяется, какую из функций требуется вызвать, основываясь на типе объекта.

class A {

. . .

public: virtual void fun() {} };

class B : public A {

. . . public: void fun() override {}};

class C : public B {

. . . public: . . . // в объявлении класса С отсутствует описание функции fun() };

main() {

A a,\*p=&a; B b; C c;

p->fun(); // вызов версии виртуальной функции fun для класса А

p=&b; p->fun(); // вызов версии виртуальной функции fun для класса B

p=&c; p->fun(); // вызов версии виртуальной функции fun для класса С (из А) }

Если в производном классе виртуальная функция не переопределяется, то используется ее версия из базового класса.

**override** используется в классе-потомке, чтобы указать что функция должна переопределять виртуальную функцию, объявленную в базовом классе. Это позволяет избавиться от ошибок, когда из-за опечатки вместо переопределения существующей виртуальной функции была создана новая (с другим именем или сигнатурой).

# **40. Абстрактные классы**

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, обеспечивающих динамическую типизацию. Часто в базовом классе эти виртуальные функции фиктивны и имеют пустое тело. Эти функции существуют как некоторая абстракция, конкретное значение им придается в производных классах. Такие функции называются ***чисто виртуальными функциями***, то есть такими, тело которых, как правило, не определено (но тело должно быть определено во всех производных классах, иначе производный класс тоже станет абстрактным ). Общая форма записи абстрактной функции имеет вид:

virtual прототип функции = 0;

Если класс имеет хотя бы одну чисто виртуальную функцию, то он называется ***абстрактным***. Для абстрактного класса нельзя создать объекты и он используется только как базовый класс для других классов.

class Figure{

public:

virtual double getSquare() =0;};

class Rectangle : public Figure{

private:

double width; double height;

public:

Rectangle(double w, double h) : width(w), height(h) { } double getSquare() override

{ return width \* height; }};

class Circle : public Figure{

private:

double radius;

public:

Circle(double r) : radius(r) { } double getSquare() override{

return radius \* radius \* 3.14; }

# **41. Виртуальные деструкторы**

Виртуальные деструкторы необходимы при использовании указателей на базовый класс при выделении памяти под динамически создаваемые объекты производных классов. Это обусловлено тем, что если объект уничтожается явно, например, используя операцию delete, то вызывается деструктор только класса, совпадающего с типом указателя на уничтожаемый объект. При этом не учитывается тип объекта, на который указывает данный указатель.

В случае объявления деструктора базового класса виртуальным, все деструкторы производных классов становятся также виртуальными. При этом если будет выполняться явное уничтожение объекта производного класса для указателя на базовый класс, то вначале вызывается деструктор производного класса, а затем вверх по иерархии до деструктора базового класса.

Если в классе имеются виртуальные функции, то желательно объявлять деструктор этого класса также виртуальным, даже если этого не требуется. Это может предотвратить возможные ошибки.

class integral{

int num;

public:

integral(int n):num(n) {} virtual ~integral()

{cout << "Inside integraln";}};

class rational:

public integral{

int den;

public:

rational(int n,int d): integral(n), den(d) {}

~rational() override

{cout << "Inside rationaln";}};

int main()

{integral \*ref = new rational; delete ref;

return 0;}

# **42. Динамическое и статическое связывание**

В C++ существует два типа связывания: статическое (раннее) связывание и динамическое (позднее) связывание. Статическая привязка происходит во время компиляции, а динамическая привязка — во время выполнения. Следовательно, их также называют ранним и поздним связыванием соответственно.

1. При ***статической*** привязке определение функции и вызов функции связываются во время компиляции, тогда как при ***динамической*** привязке вызовы функций не разрешаются до времени выполнения. Таким образом, они не связаны до времени выполнения.

2. ***Статическая*** привязка происходит, когда вся информация, необходимая для вызова функции, доступна во время компиляции. ***Динамическое*** связывание происходит, когда компилятор не может определить всю информацию, необходимую для вызова функции, во время компиляции.

3. ***Статическая*** привязка может быть достигнута с помощью обычных вызовов функций, перегрузки функций и перегрузки операторов, тогда как ***динамическая*** привязка может быть достигнута с использованием виртуальных функций.

4. Поскольку вся информация, необходимая для вызова функции, доступна до выполнения, ***статическая*** привязка приводит к более быстрому выполнению программы. В отличие от статического связывания вызов функции не разрешается до времени выполнения для последующего связывания, что приводит к несколько более медленному выполнению кода.

5. Основным преимуществом ***динамического*** связывания является его гибкость, поскольку одна функция может обрабатывать различные типы объектов во время выполнения. Это значительно уменьшает размер кодовой базы, а также делает исходный код более читабельным.

Пример статической привязки в C++:

class ComputeSum{

public:

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

int sum(int x, int y, int z) { return x + y + z;}};

int main()

{ComputeSum obj;

cout << "Sum is " << obj.sum(10, 20) << endl; cout << "Sum is " << obj.sum(10, 20, 30) <<endl;

return 0;}

Пример динамического связывания в C++:

У нас есть базовый класс B и производный класс D. Базовый класс B имеет виртуальную функцию f(), который переопределяется функцией в производном классе D, т.е. D::f() отменяет B::f().

Pешение о том, какая функция класса будет вызываться, зависит от динамического типа объекта, на который указывает basePtr. Эта информация может быть доступна только во время выполнения, и, следовательно, f() подлежит динамической привязке.

class B{

public:

virtual void f() { // Виртуальная функция

cout << "The base class function is called.\n"; }};

class D: public B

{public:

void f() {

cout << "The derived class function is called.\n";}};

int main(){

B base;

D derived;

B \*basePtr = &base; basePtr->f();

basePtr = &derived; basePtr->f();

return 0;}

# **43. Множественное наследование**

В языке С++ имеется возможность образовывать производный класс от нескольких базовых классов. Общая форма множественного наследования имеет вид:

*class имя\_произв\_класса : имя\_базового\_кл 1,…,имя\_базового\_кл N { содержимое класса }*

Иерархическая структура, в которой производный класс наследует от несколько базовых классов, называется ***множественным наследованием***. В этом случае производный класс, имея собственные компоненты, имеет доступ к protected- и public-компонентам базовых классов.

Конфликтные ситуации при применении множественного наследования:

1. Конфликт имен методов или атрибутов нескольких базовых классов

class A {public: void fun(){} };

class B { public: void fun(){} };

class C : public A, public B { };

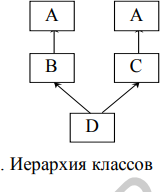
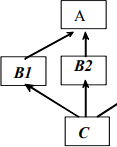
main() {C \*c=new C; c->A:: fun(); return 0; } // приходится использовать принадлежность к классу (::)

2. Многократное включение некоторого базового класса

Для решения конфликта используется виртуальное наследование.

# **44. Виртуальное наследование**

При использовании множественного наследования может возникнуть проблема из-за многократного включения некоторого базового класса. Иерархия будет выглядеть так(1) :

1 2

Для решения этой проблемы используется ***виртуальное наследование***. Если базовый класс будет являться виртуальным, то будет построен *единственный* объект этого класса. И иерархия будет выглядеть так (2)

class A { public: virtual int foo() { return 1; }};

class B : public virtual A {};

class C : public virtual A {};

class D : public B, public C {};

int main () {D d; cout << d.foo(); return 0;}

Если убрать ключевое слово virtual, то метод foo() не может быть определён однозначно и в результате не будет доступен, как и объект класса D — код не скомпилируется.

# **45. Перегрузка функций**

Одним из подходов реализации принципа полиморфизма в языке С++ является использование перегрузки функций. В С++ две и более функций могут иметь одно и то же имя. Компилятор С++ оперирует не исходными именами функций, а их внутренним представлением, которое учитывает количество и тип принимаемых аргументов. В то же время тип возвращаемого функцией значения не учитывается. Поэтому для компилятора функции с различным списком аргументов – это разные функции, а с одинаковым списком аргументов, но с разными типами возвращаемого значения – одинаковые. Для корректной работы программ последнего следует избегать. Функции, имеющие одинаковые имена, но разные списки аргументов, называются перегруженными.

class cls

{ int n; double f;

public:

cls(int N,float F) : n(N),f(F) {}

**int sum(int k)** // целочисленнный аргумент

{n+=k; return n; }

**double sum(double k)**{ // дробный аргумент

{f+=k; return f; }

void see()

{cout <<n<<' '<<f<<endl;}

};

void main() { cls obj(1,2.3);

obj.see(); // вывод содержимого объекта

cout <<obj.sum(1)<<endl; // вызов sum с целочисл. аргументом

cout <<obj.sum(1.6)<<endl; // вызов sum с дробным аргументом}

# **46. Перегрузка операторов**

Большинство операторов (операций) в С++ может быть перегружено (переопределено), в результате чего расширяется диапазон применения этих операций. Для перегрузки (переопределения) оператора разрабатываются функции, являющиеся либо членами, либо friend-функциями того класса, для которого они используются. Остановимся на перегрузке пока только с использованием методов класса. Для того, чтобы перегрузить оператор, требуется определить действие этого оператора внутри класса.

Общая форма записи функции-оператора, являющейся компонентой класса, имеет вид:

*Возвращ\_тип имя\_класса :: operator #(список аргументов) { действия, выполняемые применительно к классу }*

class Counter{

public:

Counter(int sec){ seconds = sec; }

void display() { std::cout << seconds << " seconds" << std::endl; }

int seconds;};

Counter operator + (Counter c1, Counter c2){

return Counter(c1.seconds + c2.seconds);}

bool operator == (Counter c1, Counter c2){

return c1.seconds == c2.seconds;}

bool operator > (Counter c1, Counter c2){

return c1.seconds > c2.seconds;}

int main(){

Counter c1(20);

Counter c2(10);

Counter c3 = c1 + c2;

c3.display(); // 30 seconds

bool b1 = c1 == c2; // false

bool b2 = c1 > c2; // true

return 0;}

# **47. Перегрузка бинарного оператора**

Бина́рная, или двуме́стная, опера́ция — математическая операция, принимающая два аргумента и возвращающая один результат. Функция operator для перегрузки (доопределения) бинарных операторов может быть описана двумя способами:

-как компонента-функция класса с одним аргументом;

-как глобальная функция (функция, описанная вне класса) с двумя аргументами.

При перегрузке бинарного оператора # выражение a#b может быть представлено при первом способе как a.operator#(b) или как operator #(a,b) при втором способе перегрузки.

В примере использован первый способ перегрузки.

class dek\_koord {

int x,y; // декартовы координаты точки

public: dek\_koord(){};

dek\_koord(int X,int Y): x(X),y(Y) {} ;

dek\_koord operator\*(const dek\_koord );

dek\_koord operator=(const dek\_koord);

dek\_koord operator>(const dek\_koord);

};

dek\_koord &dek\_koord:: operator\*(const dek\_koord &a) { x\*=a.x;

y\*=a.y; return \*this; }

dek\_koord dek\_koord::operator =(const dek\_koord a) { x=a.x; // перегрузка операции =

y=a.y;

return \*this; }

dek\_koord dek\_koord::operator >(const dek\_koord a) { if (x if (y0 && y-a.y>0) return 1;

if (x-a.x<0 && y-a.y<0) return -1; else return 2; }

Следует отметить, что если в описании класса присутствуют два метода (функции) перегрузки:

class dec\_koord {

…

dec\_koord operator\*(const dec\_koord k);

dec\_koord& operator\*(const dec\_koord& k); }

то возникает ошибка. Аналогично ошибкой будет, если одна функция является членом, а вторая – глобальной. Если возвращаемое значение функции operator является ссылкой, то в этом случае возвращаемое значение не может быть автоматической или статической локальной переменной. При перегрузке бинарного оператора с использованием функции-члена ей передается в качестве параметра только 1 аргумент, второй функция получает неявно через указатель \*this.

# **48. Перегрузка унарного оператора**

При перегрузке унарной операции функция operator не имеет параметров. Как и в предыдущем случае, модифицируемый объект передается в функцию operator неявным образом, используя указатель this. Унарный оператор, как и бинарный, может быть перегружен двумя способами:

-как компонента-функция без аргументов;

-как глобальная функция с одним аргументом.

Как известно, унарный оператор может быть префиксным и постфиксным. Для любого префиксного унарного оператора выражение #a может быть представлено при первом способе как a.operator#(), а при втором как #operator(a).

При перегрузке унарного оператора, используемого в постфиксной форме, выражение вида a# может быть представлено при первом способе как a.operator#(int) или как operator#(a,int) при втором способе. При этом аргумент типа int не существует и используется для отличия префиксной и постфиксной форм при перегрузке.

class dek\_koord {

int x,y; // декартовы координаты точки

public:

dek\_koord(){};

dek\_koord(int X,int Y): x(X),y(Y) {};

void operator++(); void operator++(int);

dek\_koord operator=(dek\_koord); void see(); };

void dek\_koord::operator++(){ x++;}

void dek\_koord::operator++(int){ y++; }

dek\_koord dek\_koord::operator =(dek\_koord a) {

x=a.x;

y=a.y;

return \*this; }

# **49. Дружественная функция operator**

Функция operator может быть не только членом класса, но и friend-функцией этого класса. Как было отмечено ранее, friend-функции, не являясь компонентами класса, не имеют неявного указателя this.

Следовательно, при перегрузке унарных операторов в функцию operator передается один, а бинарных – два аргумента. Необходимо отметить, что операторы: =, (), [] и -> не могут быть перегружены с помощью friend-функции operator.

class dek\_koord {

int x,y; // декартовы координаты точки

public:

dek\_koord(){};

dek\_koord(int X,int Y): x(X),y(Y) {} ;

friend dek\_koord operator\*(int,dek\_koord);

friend dek\_koord operator\*(dek\_koord,int); dek\_koord operator=(dek\_koord);

void see(); };

dek\_koord operator\*(int k,dek\_koord dk) {

dk.x\*=k;

dk.y\*=k;

return dk; }

dek\_koord operator\*(dek\_koord dk,int k) {

dk.x\*=k;

dk.y\*=k;

return dk; }

dek\_koord dek\_koord::operator=(dek\_koord dk) {

x=dk.x;

y=dk.y;

return \*this; }

# **50. Особенности перегрузки операции =**

Доопределение оператора = позволяет решить проблему присваивания, но не решает задачи инициализации, так как при инициализации должен вызываться соответствующий конструктор. В рассматриваемом случае это решается использованием конструктора копирования. Общий вид конструктора копирования имеет следующий вид:

*имя\_класса (const имя\_класса &);*

Конструктор выполняет все необходимые действия при вызове функции и копировании содержимого объекта в стек (из стека). Вызов конструктора копирования осуществляется при обращении к функции и передаче в нее в качестве параметра объекта (объектов), а также возврата значения (объекта) из функции.

class string {

char \*str;

int size;

public:

string(){}; // конструктор по умолчанию

string(int n,char \*s){ // конструктор с параметрами

str=new char[size=n>=(strlen(s)+1)?n:strlen(s)+1]; strcpy(str,s); }

string(const string &); // конструктор копирования

~string(){}; // деструктор

friend string operator+(string, const string); string &operator=(const string &); };

string:: string(const string &a){ // описание конструктора копирования

str=new char[a.size+1]; // выделяем память под this->str (+1 под ’\0’)

strcpy(str,a.str); // копирование строки

size=strlen(str); }

// перегрузка операции + string ss;

string operator+(string s1, const string s2){

ss.str=new char[ss.size = strlen(s1.str) + strlen(s2.str) + 1];

for(int i=0; ss.str[i]=s1.str[i]; i++); // перезапись символа ’\0’

ss.str[i]=' '; // удаление ’\0’

for(int j=0; ss.str[i+1]=s2.str[j]; i++,j++); // дозапись второй строки

return ss; }

string &string::operator =(const string &st){ // перегрузка операции =

if(this!=&st) // проверка, не копирование ли объекта в себя {

delete str; // освобождаем память старой строки

str=new char[size=st.size];// выделяем память под новую строку

strcpy(str,st.str); }

return \*this;

};

# **51. Перегрузка оператора []**

Оператор = необходимо перегружать с помощью компоненты-функции, использование friend-функции запрещено. Общая форма функции operator[]() имеет вид:

*Тип\_возвр\_значения имя\_класса::operator [](int i){тело функции}*

Параметр функции необязательно должен иметь тип int, но он использован, так как operator[] в основном применяется для индексации. Рассмотрим пример программы, с использованием перегрузки операции []:

class massiv {

float f[3];

public:

massiv(float i,float j,float k){

f[0]=i ; f[1]=j; f[2]=k; }

float operator[](int i) {

return f[i]; } // перегрузка оператора []

};

Данная программа при небольшой модификации может позволить использовать оператор [] как справа, так и слева от оператора присваивания. Для этого необходимо, чтобы функция оператор[]() возвращала не объект, а ссылку на него.

class massiv {

float f[3];

public:

massiv(float i,float j,float k){

f[0]=i ; f[1]=j; f[2]=k; }

float& operator[](int i) {

if(i<0 || i>2)

return f[i]; } // перегрузка оператора []

};

# **52. Перегрузка оператора ()**

Оператор () необходимо перегружать только с помощью компоненты-функции, использование friend-функции запрещено. Общая форма функции operator()() имеет вид: *тип\_возвр\_значения имя\_класса::operator()(список\_аргументов){тело функции}*

class matr { int \*\*m,a,b; public:

matr(int,int); ~matr();

int operator()(int,int); operator()(int);};

matr::matr(int i,int j): a(i),b(j){ i=0; m=new int \*[a]; for(int k=0; k<a; k++) { \*(m+k)=new int[b]; for(int n=0; n<b; n++) \*(\*(m+k)+n)=i++;}}

matr::~matr(){ for(int k=0; k<a; k++) delete [] m[k]; delete [] m;}

int matr::operator()(int i,int j) { if (i<0 || i>=a || j<0 || j>=b) { cerr<<"выход за пределы матрицы ";

return 0; } return m[i][j];}

int matr::operator()(int i) { if (i<0 || i>=a\*b) { cerr<<"выход за пределы массива "; return \*\*m;}

return m[i/b][i%b];}

# **53. Преобразование типов**

Приведение типов — это явление, при котором тип данных преобразуется из одного типа в другой, чтобы упростить вычисление исходного кода. На первом этапе выполняется попытка использовать стандартные преобразования типов (неявное преобразование). Если это невозможно, то компилятор использует преобразования, определенные пользователем (явные преобразования).

Неявные преобразования. Этот тип приведения типов используется в основном в программе, в которой в выражении присутствуют два или более типов данных. Итак, мы используем приведение типов, чтобы наши данные не были потеряны. Типы данных всех переменных обновляются до типа данных той переменной, которая содержит самый большой тип данных среди них.

Однако в случае неявных преобразований есть вероятность потери данных, теряются разные рабочие признаки. Переполнение данных также может произойти при преобразовании длинного числа в число с плавающей запятой.

Основные преобразования:

1. Если один из операндов имеет тип long double, то другой преобразуется к типу long double.

2. Если один из операндов имеет тип double, то другой преобразуется к типу double.

3. Если один из операндов имеет тип ﬂoat, то другой преобразуется к типу ﬂoat.

4. Иначе, если один из операндов имеет тип unsigned long, то другой преобразуется к типу unsigned long.

5. Иначе, если один из операндов имеет тип long, то другой преобразуется к типу long.

6. Иначе, если один из операндов имеет тип unsigned, то другой преобразуется к типу unsigned.

7. Иначе оба операнда должны иметь тип int.

Тип результата тот же, что и тип участвующих в выражении операндов.

int number1 = 10;

double number2 = 4;

double result = number1 + number2; // 14.000000

# **54. Явные преобразования типов**

Преобразование путем присвоения: это достигается путем явного определения требуемого типа данных в круглых скобках. Синтаксис представлен как (тип) выражение. «Тип» относится к типу данных, в который преобразуется результирующее значение.

int a = (int) b;

char \*s=(char \*) addr;

Недостатком их является полное отсутствие контроля, что приводит к ошибкам и путанице. Этого можно избежать в С++. Для преобразования с минимальным контролем можно использовать операцию static\_cast.

*static\_cast<тип> (выражение)*

Она позволяет выполнять преобразования, не проверяя типы выражений во время выполнения, а основываясь на сведениях, полученных при компиляции. Операция static\_cast позволяет выполнять преобразования не только указателя на базовый класс к указателю на производный, но и наоборот. В то же время преобразование int к int\* приведет к ошибке компиляции.

Для преобразований, не связанных между собой типов используется

*reinterpret\_cast. int i; void \*addr= reinterpret\_cast< void \*> i;*

Если же нужно выполнить преобразование неизменяемого типа к изменяемому, то можно использовать const\_cast: *const char \*s*

*char ss=const\_cast s*

//Снятие константности

void foo(const int\* in1, const int& in2){ int \*p; //Сняли константность и записали 33

p = const\_cast<int\*>(in1); \*p = 33; //Сняли константность и записали 55 const\_cast<int&>(in2) = 55;}

А также используется dynamic\_cast. Оператор приведения dynamic\_cast применяется для полиморфного приведения типов на этапе выполнения программы (класс считается полиморфным, если в нем есть хотя бы одна виртуальная функция). Если указатель, подлежащий приведению, ссылается на объект результирующего класса или объект класса производный от результирующего то приведение считается успешным.

istream &operator>>(std::istream &in, Car &machine) { in >> dynamic\_cast<Passenger&>(machine);}

# **55. Преобразования типов, определенных в программе**

Конструктор с одним аргументом можно явно не вызывать.

class my\_class { double x,y; public: my\_class(double X){x=X; y=x/3;} double summa(); };

double my\_class::summa() { return x+y; } void main()

{ double d; my\_class my\_obj1(15), my\_obj2=my\_class(15),

my\_obj3=15 d = my\_obj1; // ошибка не определен оператор, который принимает правый операнд из cout << my\_obj1.summa() << endl; cout << my\_obj2.summa() << endl; cout << my\_obj3.summa() << endl;}

В рассматриваемом примере все три создаваемых объекта будут инициализированы числом 15 (первые два явным, третий неявным вызовом конструктора). Следовательно, значение базовой переменной (определенной в языке) может быть присвоено переменной типа, определенного пользователем. Для выполнения обратных преобразований, то есть от переменных, имеющих тип, определенный пользователем к базовому типу, можно задать преобразования с помощью соответствующей функции operator(), например: class my\_class{ double x,y; public: operator double() {return x;} my\_class(double X){x=X; y=x/3;} double summa();};

Теперь в выражении d=my\_obj1 не будет ошибки, так как мы задали прямое преобразование типа. При выполнении этой инструкции активизируется функция operator, преобразующая значение объекта к типу double и возвращающая значение компоненты объекта. Наряду с прямым преобразованием в С++ имеется подразумеваемое преобразование типа:

class my\_cl1{ double x; // public: operator double(){return x;} my\_cl1(double X) : x(X) {}};

class my\_cl2{ double x; public: operator double(){return x;} my\_cl2(my\_cl1 XX): x(XX) {}};

void fun(my\_cl2 YY){ cout << YY <<endl; } void main()

{ fun(12); fun(my\_cl2(12)); // подразумеваемое преобразование типа}

Разрешается выполнять только одноуровневые подразумеваемые преобразования. В приведенном выше примере инструкция fun(12) соответствует двухуровневому неявному преобразованию, где первый уровень – my\_cl1(12) и второй – my\_cl2(my\_cl1(12))

Оператор typeid позволяет определить тип объекта во время выполнения.

Результатом typeid является const type\_info&. Значение является ссылкой на type\_info объект, представляющий идентификатор типа или тип выражения, в зависимости от того, какая форма typeid используется. Дополнительные сведения см. в разделе type\_info класса

int a; float\* b;

const type\_info & t = typeid(a);

std::cout << t.name(); // выводит “int”

std::cout << typeid(b).name(); // выводит “float \*”

# **56. Параметризированные классы**

Параметризированный класс – некоторый шаблон, на основе которого можно строить другие классы. Этот класс можно рассматривать как некоторое описание множества классов, отличающихся только типами их данных. В С++ используется ключевое слово template для обеспечения параметрического полиморфизма. Параметрический полиморфизм позволяет использовать один и тот же код относительно различных типов (параметров тела кода). Это наиболее полезно при определении контейнерных классов.

Спецификация шаблона класса имеет вид:

*template <список параметров> class объявление класса*

Список параметров класса-шаблона представляет собой идентификатор типа, подставляемого в объявление данного класса при его генерации.

Рассмотрим пример шаблона класса работы с динамическим массивом и выполнением контроля за значениями индекса при обращении к его элементам.

template <class T>

class vector{ T \*ms; int size; public: vector() : size(0),ms(NULL) {}

~vector(){delete [] ms;}

void inkrem(const T &t) // увеличение размера массива на 1 элемент

{ T \*tmp = ms; ms=new T[size+1]; if(tmp) memcpy(ms,tmp,sizeof(T)\*size);

ms[size++]=t; if(tmp) delete [] tmp; }

void decrem(void) // уменьшение размера массива на 1 элемент

{ T \*tmp = ms; if(size>1) ms=new T[--size];

if(tmp) { memcpy(ms,tmp,sizeof(T)\*size);

delete [] tmp; }

T &operator[](int ind) { // if(ind<0 || (ind>=size)) throw IndexOutOfRange; }};

void main(){ vector <int> VectInt; vector <double> VectDouble; VectInt.inkrem(3); VectDouble. inkrem(.26);

int b=VectInt[4]; // будет возбуждена исключительная ситуация VectInt[0]=1; VectDouble[1]=2.41;}

# **57. Передача в шаблон класса дополнительных параметров**

При создании экземпляра класса из шаблона в него могут быть переданы не только типы, но и переменные и константные выражения:

template <class T1,int i=0,class T2>

class cls {

T1 a; T2 b;

public:

cls(T1 A,T2 B) : a(A),b(B){}

~cls(){}

T1 sm(){ //описание шаблона ф-ции суммирования компонент объекта

return (T1)(a+b+i);}};

void main()

{

cls <int,1,int> obj1(3,2); // в шаблоне const i инициализируется 1

cls <int,0,int> obj2(3,2,1); // error 'cls<int,0>::cls<int,0>':no overloaded

// function takes 3 parameter s

cls <int,int,int> obj13(3,2,1); // error 'cls' : invalid template argument for 'i',

cout<<obj1.sm()<<endl;}

Результатом работы программы будет выведенное на экран число 6.

В этой программе инструкция template <class T1,int i=0,class T2> говорит о том, что шаблон класса cls имеет три параметра, два из которых – имена типов (Т1 и Т2), а третий (int i=0) – целочисленная константа. Значение константы i может быть изменено при описании объекта cls <int,1,int> obj1(3,2).

# **58. Шаблоны функций**

В С++, так же как и для класса, для функции (глобальной, то есть не являющейся компонентой-функцией) может быть описан шаблон. Это позволит снять достаточно жесткие ограничения, накладываемые механизмом формальных и фактических параметров при вызове функции. Рассмотрим это на примере функции, вычисляющей сумму нескольких аргументов.

template <class T1,class T2>

T1 sm(T1 a,T2 b)

{return (T1)(a+b);}

template <class T1,class T2,class T3>

T1 sm(T1 a,T2 b,T3 c)

{return (T1)(a+b+c);}

Int main()

{cout<<"sm(int,int) = "<<sm(4,6)<<endl;

cout<<"sm(int,int,int) = "<<sm(4,6,1)<<endl;

cout<<"sm(int,double) = "<<sm(5,3)<<endl;

cout<<"sm(double,int,short)="<<sm(0.4,6,(short)1)<<endl;

return 0;}

Используемые типы Т1, Т2, Т3 заданы как параметры для функции с помощью выражения template <class T1,class T2,class T3>. Это выражение предполагает использование типов Т1, Т2 и Т3 в виде ее дополнительных параметров. Результат работы программы будет иметь вид: вызов функции суммирования sm(int,int) = 10 вызов функции суммирования sm(int,int,int) = 11 вызов функции суммирования sm(int,double) = 8 вызов функции суммирования sm(double,int,short)= 7.4

В случае попытки передачи в функцию sm() двух строк, то есть типов, для которых не определена данная операция, компилятор выдаст ошибку. Чтобы избежать этого, можно ограничить использование шаблона функции sm(), описав явным образом функцию sm() для некоторых конкретных типов данных. В нашем случае:

char \*sm(char \*a,char \*b) // явное описание функции объединения двух строк

{char \*tmp=a;

a=new char[strlen(a)+strlen(b)+1];

strcpy(a,tmp); strcat(a,b); return a;}

Следует отметить, что шаблон функции не является ее экземпляром.

Только при обращении к функции с аргументами конкретного типа происходит генерация конкретной функции.

# **59. Совместное использование шаблонов и наследования**

Шаблонные классы, как и обычные, могут использоваться повторно. Шаблоны и наследование представляют собой механизмы повторного использования кода и могут включать полиморфизм. Шаблоны и наследования связаны между собой следующим образом:

– шаблон класса может быть порожден от обычного класса;

– шаблонный класс может быть производным от шаблонного класса;

– обычный класс может быть производным от шаблона класса.

Ниже приведен пример простой программы, демонстрирующей наследование шаблонного класса oper от шаблонного класса vect.

template <class T>

class vect

{protected:

T \*ms;

int size;

public:

vect(int n) : size(n)

{ ms=new T[size];}

~vect(){delete [] ms;}

T &operator[](const int ind) // доопределение операции []

{ if((ind>0) && (ind<size)) return ms[ind];

else return ms[0]; }

};

template <class T>

class oper : public vect<T> // класс операций над вектором

{ public:

oper(int n): vect<T>(n) {}

~oper(){}

void print()

{

for(int i=0;i<size;i++)

cout<<ms[i]<<' '; cout<<endl; }

};

void main() {

oper <int> v\_i(4); oper <double> v\_d(4);

v\_i[0]=5; v\_i[1]=3; v\_i[2]=2; v\_i[3]=4;

v\_d[0]=1.3; v\_d[1]=5.1; v\_d[2]=.5; v\_d[3]=3.5;

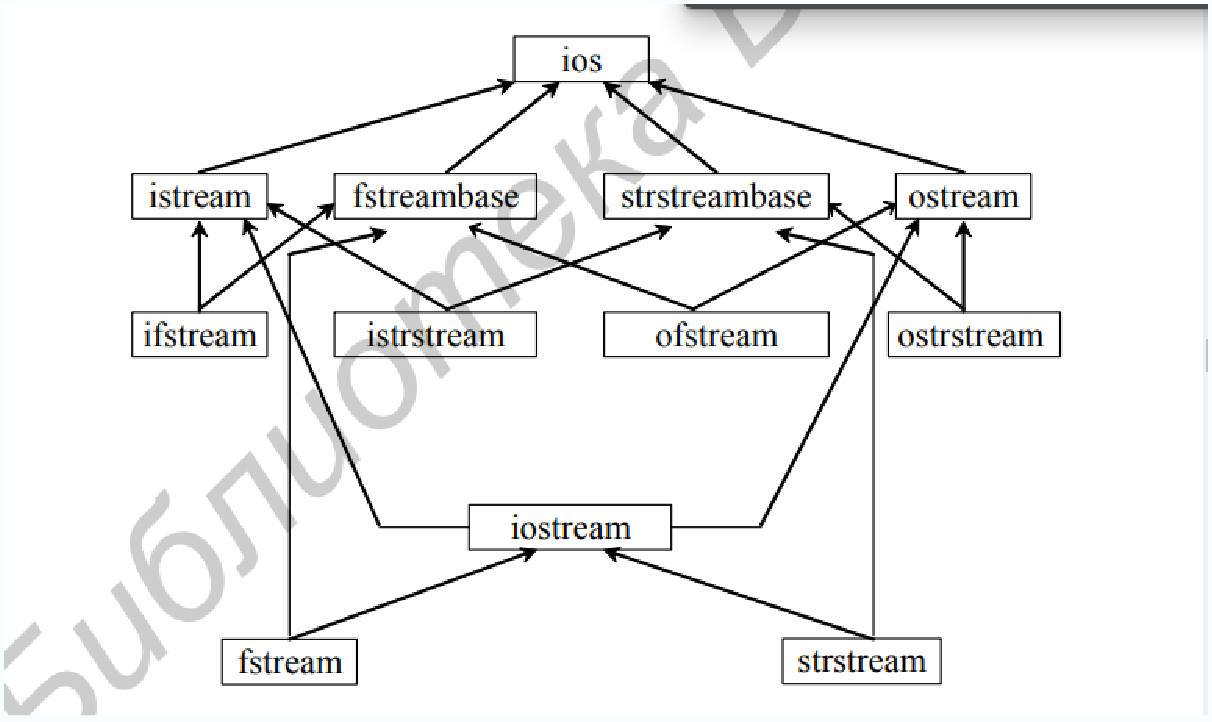
cout<<"int вектор = "; v\_i.print();

cout<<"double вектор = "; v\_d.print();}

Как следует из примера, реализация производного класса от класса шаблона в основном ничем не отличается от обычного наследования.

# **60. Организация ввода-вывода**

Системы ввода-вывода С и С++ основываются на понятии потока. Поток в С++ это абстрактное понятие, относящееся к переносу информации от источника к приемнику. В языке С++ реализованы 2 иерархии классов, обеспечивающих операции ввода-вывода, базовыми классами которых являются streambuf и ios. На рис приведена диаграмма классов, базовым для которых является ios.



# **62. Организация ввода / вывода, потоки. Перегрузка операторов << и >>.**

В С++ используется достаточно гибкий способ выполнения операций ввода-вывода классов с помощью перегрузки операторов << (вывода) и >> (ввода). Операторы, перегружающие эти операции, обычно называют инсертером и экстрактором. Для обеспечения работы с потоками ввода-вывода необходимо включить файл iostream.h, содержащий класс iostream. Этот класс является производным от ряда классов, таких как ostream, обеспечивающего вывод данных в поток, и istream – соответственно чтения из потока.

Общая форма функции перегрузки оператора ввода-вывода имеет вид:

*istream &operator>>(istream &поток,имя\_класса &объект) ostream &operator<<(ostream &поток,const имя\_класса объект)*

class cls

{ char c;

short i;

public :

cls(char C,short I ) : c(C), i(I){}

~cls(){}

friend ostream &operator<<(ostream &,const cls);

friend istream &operator>>(istream &,cls &);};

ostream &operator<<(ostream &out,const cls obj)

{ out << obj.c<<obj.i << endl;

return out;}

istream &operator>>(istream &in,cls &obj)

{ in >> obj.st>>obj.i;

return in;}

main()

{ cls s(’a’,10),ss(’ ’,0);

out<<"abc"<<endl;

cout<<s<<ss<<endl;

cin >> ss;

return 0;}

# **63. Функция get() с тремя параметрами**

Функция get() принимает три параметра: символьный массив, максимальное число символов и ограничитель ввода (по умолчанию ’\n’). Ввод прекращается, когда считано число символов на один меньшее максимального или считан символ-ограничитель. При этом в вводимую строку добавляется нуль-символ. Символ-ограничитель из входного потока не удаляется, это при повторном вызове функции get приведет к формированию пустой строки.

сhar s[30];

сin.get(s,20)) // аналогично cin.get(s,20,’\n’) cout<<s<<endl;

# **64. Функция get() без параметров и с одним параметром.**

Функция get без параметров. Вводит символ из соответствующего потока одиночный символ и возвращает его значение. Если из потока прочитан признак конца файла, то get возвращает EOF.

main()

{ char c;

Cout << "вводите текст" << endl; while((c=cin.get())!=EOF)

cout.put(c);

cout << endl<<cin.eof(); return 0; }

В программе считывается из потока cin очередной символ и выводится с помощью функции put. При считывании признака конца файла завершается цикл while. До и после цикла выводится значение cin.eof(), равное false (выводится 0) до начала цикла и true (выводится 1) после его окончания.

Без параметров форма get возвращает элемент, считываемый как целое число или конец файла. Остальные формы возвращают поток (\*this).

while(сin.get(с))

cout.put(c);

# **65. Функция getline().**

Функция getline() извлекает символы из входного потока и добавляет их к строковому объекту до тех пор, пока не встретится символ-разделитель. При этом ранее сохраненное значение в строковом объекте будет заменено входной строкой

istream& getline(istream& is, string& str, char delim)

str: это строковый объект, входные данные сохраняются в этом объекте после чтения из потока.

delim- символ-разделитель, по умолчанию \n). Если этот параметр не указан, getline() будет считывать всю строку до символа конца строки \n строки.

int main() {

string input;

getline(cin, input, ';');

cout << "input =”<< input << endl;

return 0;

}

# **66. Состояние потока / 67. Функции чтения состояния потока**

Потоки iostream или ostream имеют связанное с ними состояние. В классе ios, базовом для классов iostream и ostream, имеется несколько public функций, позволяющих проверять и устанавливать состояние потока:

inline int ios::bad() const { return state & badbit; } inline int ios::eof() const { return state & eofbit; }

inline int ios::fail() const { return state & (badbit | failbit); } inline int ios::good() const { return state == 0; }

В классе ios есть еще несколько функций, позволяющих прочитать (rdstate) и очистить (clear) состояние потока:

inline int ios::rdstate() const { return state; }

inline void ios::clear(int \_i=0){ lock(); state = \_i; unlock(); }

Так, если было установлено состояние ошибки, то попытка выполнить ввод/вывод будет игнорироваться до тех пор, пока не будет устранена причина ошибки и биты ошибки не будут сброшены функцией clear().

main()

{ int i;

int flags;

do

{cin >> i;

flags=cin.rdstate(); // чтение состояния потока

if(flags | ios::badbit)

{ cout << "error in stream"<< endl;

cin.clear(0); // сброс всех состояний потока}}

while(flags); // пока ошибка во входном потоке

return 0;}

В приведенном примере функция cin.clear(0) выполняет сброс всех установленных бит ошибки. Если требуется сбросить, например, только badbit, то

clear(ios::badbit).

. . .

ifstream in("file");

while(1)

{ state=in.rdstate();

if (state)

{if(state&ios::badbit)

cout<<"ошибка открытия файла"<<endl;

else if(state&ios::eofbit)

cout<<"в файле больше нет данных"<<endl;

break;}

else

in >> ss;

**Строковые** потоки

Особой разновидностью потоков являются строковые потоки, представленные классом strstream:

class strstream : public iostream

{ public:

strstream();

strstream(char \*s, streamsize n, ios\_base::openmode=ios\_base::in | ios\_base::out); strstreambuf \*rdbuf() const;

void freeze(bool frz=true); char \*str();

streamsize pcount() const;};

Важное свойство класса strstream состоит в том, что в нем автоматически выделяется требуемый объем памяти для хранения строковых данных. Все операции со строковыми потоками происходят в памяти в специально выделенном для них буфере strstreambyf. Строковые потоки позволяют облегчить формирование данных в памяти. Далее в примере показывается проверка состояния потока, чтобы убедиться в правильности выполняемых операций. Это позволяет, например, легко определить переполнение буфера.

void fun(const char \*s,int n)

{ char \*buf=new char[n]; // входной буфер

strstream st(buf,n,ios::in|ios::out); // связывание потока st и буфера byf

st << s << ends; // ввод информации в буфер

if(st.good())

cout << buf<<endl;

else

cerr<<"error"<<endl;}

main()

{ fun("123456789",5);

fun("123456789",15);}

template <typename T>

bool bFile<T>::openWriteFile()

{try {

ft.open(file\_name, ios::out | ios::binary);

if (!ft.good()) throw FileException(5, "ошибка работы с файлами", "Ошибка при открытии файла");

if (!ft.is\_open())

return false;

else return true;

catch (FileException obj){ obj.Print();

exit(1);}}

# **68. Организация работы с бинарными файлами. Запись объектов в файл и чтение объектов из файла.**

В языке С++ для организации работы с  бинарными файлами используются классы потоков fstream. Данный класс является производным iostream.

В С++ файл открывается путем стыковки его с соответствующим

потоком. Рассмотрим организацию связывания потока с некоторым файлом.

Для этого используются конструкторы классов ifstream и ofsream:

**ofstream(const char\* Name, int nMode= ios::out, int nPot= filebuf::openprot);**

**ifstream(const char\* Name, int nMode= ios::in, int nPot= filebuf::openprot);**

**Первый аргумент** определяет имя файла (единственный обязательный параметр).

**Второй аргумен**т задает режим для открытия файла и представляет

битовое ИЛИ величин:

ios::app при записи данные добавляются в конец файла, даже если

текущая позиция была перед этим изменена функцией ostream::seekp;

ios::ate указатель перемещается в конец файла. Данные записываются в

текущую позицию (произвольное место) файла;

ios::in поток создается для ввода: если файл уже существует, то он

сохраняется;

ios::out поток создается для вывода (по умолчанию для всех ofstream

объектов);

ios::trunc если файл уже существует, его содержимое уничтожается. Этот режим действует по умолчанию, если ios::out установлен, а ios::ate, ios::app или ios:in не установлены;

ios::nocreate не открывать несуществующий файл ;

ios::noreplace не открывать для вывода существующий файл, если не установлены ate / app;

ios::binary ввод-вывод будет выполняться в двоичном виде (по

умолчанию текстовой режим).

**Третий аргумент** – данное класса filebuf, используется для установки

атрибутов доступа к открываемому файлу.

Возможные значения nProt:

filebuf::sh\_compat совместно используют режим;

filebuf::sh\_none режим Exclusive: никакое совместное использование;

filebuf::sh\_read совместно использующее чтение;

filebuf::sh\_write совместно использующее запись.

Для комбинации атрибутов filebuf::sh\_read and filebuf::sh\_write используется операция логическое ИЛИ ( || ).

class File{ protected: std::ifstream ifStream; std::ofstream ofStream; std::fstream fStream; };

class BinFile : public File{public: template<typename Type> void OutputBinFile(Type& obj); template<typename Type> void InputBinFile(Type& obj);};

template<typename Type>

void BinFile::OutputBinFile(Type& obj){ fStream << obj;}

template<typename Type>

void BinFile::InputBinFile(Type& obj){ fStream >> obj;}

# **69. Организация работы с файлами последовательного доступа. Запись объектов в файл и чтение объектов из файла.**

Последовательные файлы — [файлы](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB), хранящие информацию в неструктурированном (для поиска и обращения) виде. Поиск в таких файлах осуществляется последовательным считыванием файла с начала и сравнением «всего» с искомым. Так же и обращение к определённому участку файла каждый раз требует «чтения с начала».

Примером последовательных файлов являются [текстовые файлы](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) (\*.txt)

Последовательные файлы выигрывают у [файлов с произвольным доступом](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB%D1%8B_%D1%81_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%BB%D1%8C%D0%BD%D1%8B%D0%BC_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC) по компактности, но проигрывают по скорости доступа. Файлы с последовательным доступом – это в основном текстовые файлы, которые можно открывать с помощью текстового редактора. Текстовый файл может содержать коды символов, признак перевода строки vbCrLf, признак табуляции vbTab и признак конца файла. Здесь записи – это строки переменной длины, отделенные друг от друга символом перевода строки. Такие файлы обычно создаются приложениями для обработки и хранения текстовой информации (но не числовой). Файлы с последовательным доступом читаются от начала к концу, поэтому невозможно одновременно и считывать из них данные, и записывать таковые. Обычно информация из текстового файла считывается вся в память и сохраняется вся в файле после окончания работы с ней. Чтобы изменить одну запись файла последовательного доступа, его нужно весь записать заново. Если же приложению требуется частый доступ к данным, хранящимся в некотором файле, следует использовать файлы с произвольным доступом.

Запись:#define N 15

string filetName;

char name[N], surname[N], patronymic[N];

cin>> fileName;

cin>> name;

cin>> surname;

cin>> patronymic;

ofstream file(fileName, ios::app);

if (file.is\_open()) {

file << name<<" ";

file << surname << " ";

file << patronymic << endl;

file.close();}

else {cout << "file eror" << std::endl;}

Чтение

#define N 15

string filetName;

char name[N], surname[N], patronymic[N];

cin>> fileName;

ifstream file(fileName);

if (file.is\_open())

{file >> name;

file >> surname;

file >> patronymic;

file.close();}

else {cout << "file eror" << std::endl;}

# **70. Организация работы с файлами произвольного доступа. Запись объектов в файл, чтение объектов из файла, перезапись.**

Организация хранения информации в файле прямого доступа предполагает доступ к ней не последовательно от начала файла по некоторому ключу, а непосредственно, например, по их порядковому номеру. Для этого требуется, чтобы все записи в файле были бы одинаковой длины. Наиболее удобными для организации произвольного доступа при вводе/выводе информации являются следующие компоненты-функции:

istream& istream::read(reinterpret\_cast<char \*>(&s), streamsize n);

ostream& ostream::write(reinterpret\_cast<const char \*>(&s), streamsize n);

при этом, так как функция write (read) ожидает первый аргумент типа const сhar\* (char \*), то для требуемого приведения типов используется оператор явного преобразования типов:

Ниже приведен текст программы организации работы с файлом произвольного доступа на примере удаления и добавления в файл информации о банковских реквизитах клиента (структура inf).

struct inf

{ char cl[10];

int pk;

double sm;

}cldata;

class File

{ char filename[80];

fstream \*fstr;

int maxpos;

public:

File(char \*filename);

~File();

int Open();

const char\* GetName();

int Read(inf &);

void Remote();

void Add(inf);

int Del(int pos);

friend ostream& operator << (ostream &out, File &obj)

{inf p;

out << "File " << obj.GetName() << endl;

obj.Remote();

while(obj.Read(p))

out<<"\nКлиент -> "<<p.cl<<' '<<p.pk<<' '<<p.sm;

return out;}};

File::File(char \*\_filename) // конструктор

{strncpy(filename,\_filename,80);

fstr = new fstream();}

File::~File(){ fstr->close(); }

int File::Open(){

fstr->open(filename, ios::in | ios::out | ios::binary);

if (!fstr->is\_open()) return -1;

return 0;}

int File::Read(inf &p)

{if(!fstr->eof() && fstr->read(reinterpret\_cast<char\*>(&p),sizeof(inf))) return 1;

fstr->clear();

return 0;}

void File::Remote(){

fstr->seekg(0,ios\_base::beg);

fstr->seekp(0,ios\_base::beg);

fstr->clear();}

const char\* File::GetName() {

return this->filename;

} void File::Add(T cldata){

fstr->seekp(0,ios\_base::end);

fstr->write(reinterpret\_cast<char\*>(&cldata),sizeof(inf)); fstr->flush();}

int File::Del(int pos) {

Remote();

fstr->seekp(0,ios::end);

maxpos = fstr->tellp();

maxpos/=sizeof(inf);

if(maxpos<pos) return -1;

fstr->seekg(pos\*sizeof(inf),ios::beg);

while(pos<maxpos)

{ fstr->read(reinterpret\_cast<char\*>(&cldata),sizeof(inf));

fstr->seekp(-2\*sizeof(inf), ios::cur);

fstr->write(reinterpret\_cast<char\*>(&cldata),sizeof(inf));

fstr->seekg(sizeof(inf),ios::cur);

pos++;}

strcpy(cldata.cl,"");

cldata.pk=0;

cldata.sm=0;

fstr->seekp(-sizeof(inf), ios::end);

fstr->write(reinterpret\_cast<char\*>(&cldata),sizeof(inf)); fstr->flush();}

int main(void)

{int n;

File myfile("file");

if(myfile.Open() == -1)

{ cout << "Can't open the file\n";

return -1;}

cin >> cldata.cl >> cldata.pk >> cldata.sm;

myfile.Add(cldata);

cout << myfile << endl;

cout << "Введите номер для удаления "; cin>>n;

if(myfile.Del(n) == -1)

cout << "Желаемый объект "<<n<<" вне файла\n";

cout << myfile << endl; }

# **71. Функции позиционирования в файле.**

Классы istream и ostream содержат по 2 перегруженные компоненты-функции для перемещения указателя в требуемую позицию в потоке (связанном с ним файле). Такими функциями являются seekg (переместить указатель для извлечения из потока) и seekp (переместить указатель для помещения в поток). Обмен информацией с файлом осуществляется посредством функций get и put. Кроме рассмотренных функций в этих классах имеются еще функции tellg и tellp, возвращающие текущее положение указателей get и put соответственно.

Ниже приведена программа, выполняющая ввод символов в файл с их одновременным упорядочиванием (при вводе) по алфавиту. Использование функций seekg, tellg позволяет позиционировать текущую позицию в файле, то есть осуществлять прямой доступ в файл.

using namespace std;

void error(char \*s1,char \*s2="")

{ cerr<<s1<<" "<<s2<<endl; exit(1);}

int main()

{ char c,cc;

int n;

fstream f; // выходной поток

streampos p,pp;

f.open("aaaa",ios::in|ios::out);

if(!f)

error("ошибка открытия файла","aaaa");

f.seekp(0); // установить текущий указатель в начало потока

while(1)

{ cin>>c;

if (c=='q' || f.bad())

break;

f.seekg(0,ios::beg);

while(1)

{ if(((cc=f.get())>=c) || (f.eof()))

{ if(f.eof())

{ f.clear(0);p=f.tellg(); }

else

{ p=f.tellg()-1;

f.seekg(-1,ios::end);

pp=f.tellg();

while(p<=pp)

{cc=f.get();

f.put(cc);

f.seekg(--pp); }}

f.seekg(p);

f.put(c);

break; }}}

f.close();

return 1; }

# **72. Абсолютный обработчик**

Обработчики исключительных ситуаций catch. Обработчики исключительных ситуаций являются важнейшей частью всего механизма обработки исключений, так как именно они определяют поведение программы после генерации и перехвата исключительной ситуации. Синтаксис блока catch имеет следующий вид:

catch(тип 1 <аргумент>) { тело обработчика} catch(тип 2 <аргумент>)) { тело обработчика}...catch(тип N <аргумент>)) {

тело обработчика}

Таким образом, так же как и в случае блока try, после ключевого слова

catch должен следовать составной оператор, заключенный в фигурные скобки. В аргументах обработчика можно указать только тип исключительной ситуации, необязательно объявлять имя объекта, если этого не требуется. У каждого блока try может быть множество обработчиков, каждый из которых должен иметь свой уникальный тип исключительной ситуации. Неправильной будет следующая запись:

class cls{ public:int i; }; try{. . . }catch(cls i1){

. . . } catch(int i2){ }

В этом случае cls – это отдельный тип исключительной ситуации. Существует также абсолютный обработчик, который совместим с любым типом исключительной ситуации. Для написания такого обработчика надо вместо аргументов написать многоточие (эллипсис). catch (…){блок обработки исключения}

Использование абсолютного обработчика исключительных ситуаций рассмотрим на примере программы, в которой происходит генерация исключительной ситуации типа char \*, но обработчик такого типа отсутствует. В этом случае управление передается абсолютному обработчику.

void int\_exception(int i)

{if(i>100) throw 1;}

void string\_exception()

{ throw "Error";}

void main()

{try{

int\_exception(99);

string\_exception();

}catch(int){

cout<<"Обработчик для типа Int";

getch();

} catch(...)

{cout << "Абсолютный обработчик ";

getch( );}}

Результат выполнения программы: Абсолютный обработчик

Так как абсолютный обработчик перехватывает исключительные ситуации всех типов, то он должен стоять в списке обработчиков последним. Нарушение этого правила вызовет ошибку при компиляции программы.

# **73. Основы обработки исключительных ситуаций**

Для того чтобы эффективно использовать механизм обработки исключительных ситуаций, необходимо грамотно построить списки обработчиков, а для этого, в свою очередь, нужно четко знать следующие правила, по которым осуществляется поиск соответствующего обработчика:

– исключительная ситуация обрабатывается первым найденным обработчиком, т. е. если есть несколько обработчиков, способных обработать данный тип исключительной ситуации, то она будет обработана первым стоящим в списке обработчиком;

– абсолютный обработчик может обработать любую исключительную ситуацию;

– исключительная ситуация может быть обработана обработчиком соответствующего типа либо обработчиком ссылки на этот тип;

– исключительная ситуация может быть обработана обработчиком базового для нее класса. Например, если класс В является производным от класса А, то обработчик класса А может обработать исключительную ситуацию класса В;

– исключительная ситуация может быть обработана обработчиком, принимающим указатель, если тип исключительной ситуации может быть приведен к типу обработчика, путем использования стандартных правил преобразования типов указателей.

Если при возникновении исключительной ситуации подходящего обработчика нет среди обработчиков данного уровня вложенности блоков try, то обработчик ищется на следующем охватывающем уровне. Если обработчик не найден вплоть до самого верхнего уровня, то программа аварийно завершается.

class A{};

class B{};

class C : public A, public B {};

void f(int i)

{ if(i)

throw C(); // возбуждение исключительной ситуации типа объект С

else

throw new C; // возбуждение исключительной ситуации

// типа указатель на объект класса С}

void main()

{ int i;

try{

cin>>i;

f(i);

}catch(A)

{cout<<"A handler"; }

catch(B&) {

cout<<"B& handler"; }

catch(C) {

cout<<"C handler"; }

catch(C\*) {

cout<<"C\* handler"; }

catch(A\*) {

cout<<"A\* handler"; }

catch(void\*) {

cout<<"void\* handler"; }}

В данном примере исключительная ситуация класса С может быть направлена любому из обработчиков A, B& или C, поэтому выбирается обработчик, стоящий первым в списке. Аналогично для исключительной ситуации, имеющей тип указателя на объект класса С, выбирается первый подходящий обработчик A\* или C\*. Эта ситуация также может быть обработана обработчиками void\*. Так как к типу void\* может быть приведен любой указатель, то обработчик этого типа будет перехватывать любые исключительные ситуации типа указателя.

# **74. Перенаправление исключительных ситуаций**

Иногда возникает положение, при котором необходимо обработать исключительную ситуацию сначала на более низком уровне вложенности блока try, а затем передать ее на более высокий уровень для продолжения обработки.

Для того чтобы сделать это, нужно использовать throw без аргументов. В этомслучае исключительная ситуация будет перенаправлена к следующему подходящему обработчику (подходящий обработчик не ищется ниже в текущем списке – сразу осуществляется поиск на более высоком уровне). Приводимый нижепример демонстрирует организацию такой передачи. Программа содержит вложенный блок try и соответствующий блок catch. Сначала происходит первичная обработка, затем исключительная ситуация перенаправляется на более высокий уровень для дальнейшей обработки.

void func(int i)

{try

{if(i) throw "Error";}

catch(char \*s)

{cout<<s<<"- выполняется первый обработчик"<<endl; throw;}}

void main()

{try

{ func(1);

}catch(char \*s)

{cout<<s<<"- выполняется второй обработчик"<<endl;}}

Результат выполнения программы:

Error – выполняется первый обработчик Error – выполняется второй обработчик

Если ключевое слово throw используется вне блока catch, то автоматически будет вызвана функция terminate(), которая по умолчанию завершает программу.

# **75. Исключительная ситуация, генерируемая оператором new**

Исключение вызываемое оператором new - bad\_alloc. Пример:

int main()

{int\* ms;

try

{while(1) ms=new int [100000000000];}

catch(bad\_alloc exept)

{cout<<”Oshibka: ”<< exept.what()<<endl;}

return 0;}

Для переопределения вызываемого исключения используют - set\_new\_handeler(указатель на функцию) (#include <new>)

Пример:

void my\_handeler()

{ cout<<”Oshibka”<<endl;

throw bad\_alloc();}

int main()

{set\_new\_handler(my\_handeler);

int\* ms;

try{

while(1) ms=new int[100000000000];}

catch(bad\_alloc& exept)

{ cout<<exept.what()<<endl;}

return 0;}

# **76. Генерация исключений в конструкторах**

# Так как конструктор не возвращает значения, то соответственно нельзя возвратить некий код ошибки и приходится искать альтернативу. В этом случае наилучшим решением является генерация и обработка исключительной ситуации. При генерации исключения внутри конструктора процесс создания объекта прекращается. Если к этом моменту были вызваны конструкторы базовых классов, то будет обеспечен и вызов соответствующих деструкторов. Рассмотрим на примере генерацию исключительной ситуации внутри конструктора. Пусть имеется класс В, производный от класса А и содержащий в качестве компоненты-данного объект класса lосаl. В конструкторе класса В генерируется исключительная ситуация.

#include <iostream>

using namespace std;

**class local**

{ public:

local() { cout<<”Constructor of local”<<endl; }

~local() { cout<<”Destructor of local”<<endl; }};

**class A**

{ public:

A () { cout<<”Constructor of A”<<endl; }

~A () { cout<<”Destructor of A”<<endl; }};

**class B : public A**

{ public:

B(int)

{ cout<<”Constructor of B”<<endl;

if(i) throw 1; }

~B () { cout<<”Destructor of B”<<endl; }};

int main()

{ try {

B ob(1);}

catch(int) {

cout<<”int exception handler”;}

return 0;}

Результат выполнения программы:

Constructor of A

Constructor of local

Constructor of B

Destructor of local

Destructor of A

int exception handler

В программе при создании объекта производного класса В сначала вызываются конструкторы базового класса А, затем класса lосаl, который является компонентом класса В. После этого вызывается конструктор класса В, в котором генерируется исключительная ситуация. Видно, что при этом для всех ранее созданных объектов вызваны деструкторы, а для объекта самого класса В деструктор не вызывается, так как конструирование этого объекта не было завершено.

Если в конструкторах выполнялось динамическое выделение памяти, то при генерации исключительной ситуации выделенная память автоматически освобождена не будет, об этом необходимо заботиться самостоятельно, иначе возникнет утечка памяти.

# **77. Задание собственной функции завершения**

Если программа не может найти подходящий обработчик для сгенерированной исключительной ситуации, то будет вызвана процедура завершения {terminate() (ее также называют обработчиком завершения), по умолчанию выполнение программы будет остановлено и на экран будет выведено сообщение «Abnormal program termination». Однако можно установить собственный обработчик завершения, используя функцию set\_terminate(), единственным аргументом которой является указатель на новую функцию завершения (*функция, принимающая и возвращающая void*), а возвращаемое значение - указатель на предыдущий обработчик. Ниже приведен пример установки собственного обработчика завершения и генерации исключительной ситуации, для которой не можетбыть найден обработчик.

#include <iostream>

using namespace std;

#include <exception>

#include <stdlib.h>

void my\_term()

{ cout<<"Собственная функция-обработчик";

exit(1);}

int main()

{ set\_terminate(my\_term);

try {

throw 1; // генерация исключительной ситуации типа int}

catch(char) { // обработчик для типа char

cout<<"char handler";}

return 0;}

Результат выполнения программы:

Собственная функция-обработчик

# **78. Спецификации исключительных ситуаций**

Иногда возникает необходимость заранее указать, какие исключения могут генерироваться в той или иной функции. Это можно сделать с помощью так называемой спецификации исключительных ситуаций. Это средство позволяет указать в объявлении функции типы исключительных ситуаций, которые могут в ней генерироваться. Синтаксически спецификация исключения является частью заголовочной записи функции и имеет вид:

**объявление функции throw(тип1, тип2....) {тело функции}**

где тип1, тип2,... — список типов, которые может иметь выражение throw внутри функции. Если список типов пуст, то компилятор полагает, что функцией не будет выполняться никакой throw.

void fun(char c) throw();

Использование спецификации исключительных ситуаций не означает, что в функции не может быть сгенерирована исключительная ситуация некоторого не указанного в спецификации типа. Просто в этом случае программа по умолчанию завершится, так как подобные действия приведут к вызову неожиданного обработчика. Таким образом, когда функция генерирует исключительную: ситуацию, не описанную в спецификации, выполняется неожиданный обработчик unexpected().

**79. Правила поиска обработчика исключительных ситуаций**

Для того чтобы эффективно использовать механизм обработки исключительных ситуаций, необходимо грамотно построить списки обработчиков, а для этого, в свою очередь, нужно четко знать следующие правила, по которым осуществляется поиск соответствующего обработчика:

– исключительная ситуация обрабатывается первым найденным обработчиком, т. е. если есть несколько обработчиков, способных обработать данный тип исключительной ситуации, то она будет обработана первым стоящим в списке обработчиком;

– абсолютный обработчик может обработать любую исключительную ситуацию;

– исключительная ситуация может быть обработана обработчиком соответствующего типа либо обработчиком ссылки на этот тип;

– исключительная ситуация может быть обработана обработчиком базового для нее класса. Например, если класс В является производным от класса А, то обработчик класса А может обработать исключительную ситуацию класса В;

– исключительная ситуация может быть обработана обработчиком, принимающим указатель, если тип исключительной ситуации может быть приведен к типу обработчика, путем использования стандартных правил преобразования типов указателей.

Если при возникновении исключительной ситуации подходящего обработчика нет среди обработчиков данного уровня вложенности блоков try, то обработчик ищется на следующем охватывающем уровне. Если обработчик не найден вплоть до самого верхнего уровня, то программа аварийно завершается.

class A{};  
class B{};  
class C : public A, public B {};   
void f(int i)  
{ if(i)   
 throw C(); // возбуждение исключительной ситуации типа объект С  
 else   
 throw new C; // возбуждение исключительной ситуации  
// типа указатель на объект класса С}  
void main()  
{int i;   
 try{   
 cin>>i;  
 f(i);   
 }catch(A)   
 {cout<<"A handler"; }   
 catch(B&) {  
 cout<<"B& handler"; }   
 catch(C) {  
 cout<<"C handler"; }   
 catch(C\*) {  
 cout<<"C\* handler"; }   
 catch(A\*) {  
 cout<<"A\* handler"; }  
 catch(void\*) {  
 cout<<"void\* handler"; }}

В данном примере исключительная ситуация класса С может быть направлена любому из обработчиков A, B& или C, поэтому выбирается обработчик, стоящий первым в списке. Аналогично для исключительной ситуации, имеющей тип указателя на объект класса С, выбирается первый подходящий обработчик A\* или C\*. Эта ситуация также может быть обработана обработчиками void\*. Так как к типу void\* может быть приведен любой указатель, то обработчик этого типа будет перехватывать любые исключительные ситуации типа указателя.

# **80. Механизм развертывания стека**

Одним из главных достоинств использования механизма обработки исключительных ситуаций является обеспечение развертывания стека. Развертывание стека – это процесс вызова деструкторов локальных объектов, когда исключительные ситуации выводят их из области видимости.

class add\_class{

private:int num;

public:

add\_class(int a)

{ num=a; cout<<"Constructor "<<num<<endl;}

~add\_class()

{ cout<<"Destructor of add\_class "<<num<<endl; } void show\_num()

{ cout<<" "<<num<<" ";}

void input\_num(int a)

{ num=a; } int output\_num(){return num;}};

add\_class add(add\_class a,add\_class b)

{ add\_class sum(0); // объектов a и b

int s=a.output\_num()+b.output\_num();

if(s>INT\_MAX) throw 1; // генерация исключения типа int sum.input\_num(s);

return sum;}

int main()

{add\_class a(INT\_MAX),b(1),s(0);

try{ // охранный блок s=add(a,b); cout<<"Result"; s.show\_num(); cout<<endl;}

catch(int){ // обработчик исключения типа int cout<<"Overflow error"<<endl;}

return 0;}

Результат выполнения программы:

Constructor 2147483647

Constructor 1

Constructor 0

Constructor 0

Destructor of add\_class -2147483648

Destructor of add\_class 2147483647

Destructor of add\_class 1

Destructor of add\_class -2147483648

Result -2147483648

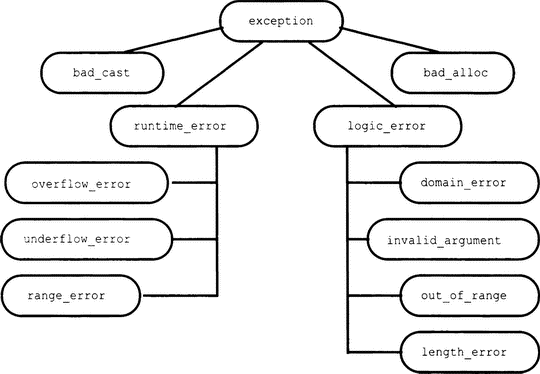
Destructor of add\_class -2147483648

Destructor of add\_class 1

Destructor of add\_class 2147483647

# **81. Иерархия исключений стандартной библиотеки**

Вершиной иерархии является класс exception (определенный в заголовочном файле <exception>). В этом классе содержится функция what(), переопределенная в каждом производном классе для выдачи сообщения об ошибке. Непосредственными производными классами от класса exception являются классы runtime\_error и logic\_error, имеющие по несколько производных классов. Производными от exception также являются исключения: bad\_alloc, генерируемое оператором new, bad\_cast, генерируемое dynamic\_cast, и bad\_typeid, генерируемое оператором typeid. Класс logic\_error и производные от него классы (invalid\_argument, length\_error, out\_of\_range) указывают на логические ошибки (передача неправильного аргумента функции, выход за пределы массива или строки). Класс runtime\_error и производные от него (overflow\_error и underflow\_error) указывают на математические ошибки переполнения сверху и снизу.



# **82. Общее понятие о контейнере**

**Контейнер** – это хранилище объектов (как встроенных, так и определённых пользователем типов). Как правило, контейнеры реализуются в виде шаблонных классов. Стандартная библиотека включает в себя реализации таких контейнеров, как контейнеры последовательностей (вектор (vector), список (list), очередь (deque)), ассоциативные контейнеры(ассоциативный массив (map), множество (set), multiset, multimap), адаптеры контейнеров(stack, queue, priority\_queue).

#include <vector>

int main() { vector <string> vec\_string;

vec\_string.push\_back("апельсин");

vec\_string.push\_back("груша");

vec\_string.push\_back("яблока");

vec\_string.push\_back("вишня");

for (int i = 0; i < vec\_string.size(); i++) {

    cout << vec\_string[i] << " ";} cout << endl;

vec\_string.pop\_back();

for (int i = 0; i < vec\_string.size(); i++) {    cout << vec\_string[i] << " ";  }

return 0;}

# **83. Общее понятие об итераторе**

Итератор – это абстракция указателя, т.е. объект, который может ссылаться на другие объекты, содержащиеся в контейнере. Основные функции итератора − обеспечение доступа к объекту, на который он ссылается (разыменование), и переход от одного элемента контейнера к другому (итерация, отсюда и название итератора). Итераторы обычно создаются как друзья классов, с которыми они работают, что позволяет выполнить прямой доступ к частным данным этих классов. Пример:

int main()

{ vector<int> vect;

vect.push\_back(2);

vect.push\_back(23);

vect.push\_back(12);

copy(vect.begin(), vect.end(), ostream\_iterator<int>(cout,” “);

return 0;}

# **84. Категории итераторов**

Итератор произвольного доступа -> итератор двунаправленного доступа -> итератор однонаправленного доступа->итератор ввода(->итератор вывода). Итераторы ввода (input iterator) стоят в самом низу иерархии итераторов. Это наиболее простые из всех итераторов STL, и доступны они только для чтения.

#include <algorithm> #include <iostream> #include <vector>

int main() { vector<int> v(4);

istream\_iterator<int> ii(cin); //итератор ввода for (int j, i = 0; i < 4; i++){ v[i] = \*ii++;}

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, "\n"));return 0;}

Однонаправленный итератор (forward iterator), который может перемещаться по цепочке объектов в одном направлении, за что и получил свое название. Для такого перемещения в итераторе определена операция инкремента (++). Кроме этого, в однонаправленном итераторе есть операторы сравнения (== и !=), присвоения (=) и разыменовывания (\*).

template void replace (ForwardIterator first, ForwardIterator last, const T& old\_value, const T& new\_value) { while (first != last) { if (\*first == old\_value) \*first = new\_value; firs++t;}}

# **85. Основные итераторы. Итераторы ввода/вывода**

Итераторы ввода (input iterator) стоят в самом низу иерархии итераторов. Это наиболее простые из всех итераторов STL, и доступны они только для чтения. Если итератор ввода предназначен для чтения данных, то итератор вывода (output iterator) служит для ссылки на область памяти, куда выводятся данные.

Для данного итератора определены операторы присвоения (=), разыменовывания (\*) и инкремента (++).

#include <algorithm>

#include <iostream>

#include <vector>

int main()

{ vector<int> v(4);

istream\_iterator<int> ii(cin); //итератор ввода

for (int i = 0; i < 4; i++)

{ v[i] = \*ii++;}

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, "\n"));

return 0;}

# **86. Вспомогательные итераторы**

Вспомогательные итераторы названы так потому, что они выполняют

вспомогательные операции по отношению к основным.

Реверсивные итераторы служит для обхода контейнера с конца (rend()) до начала (rbegin())

void main() { const int init[] = {1, 2, 3, 4, 5};

vector v(5); copy(init, init + 5, v.begin());

copy(v.begin(), v.end(), ostream\_iterator(cout, " "));

copy(v.rbegin(), v.rend(), ostream\_iterator(cout, " "));

//обратный обход}

Результаты работы программы:

1 2 3 4 5

5 4 3 2 1

Итераторы потока:

int main()

{ istream\_iterator<int> is(cin);

ostream\_iterator<int> os(cout, " – введенное значение \n");

int input;

while (input != 999)

{ input = \*is;

\*os++ = input;

is++;}

return 0;}

Константный итератор. Он не допускает изменения данных, на которые он ссылается. Можно считать константный итератор указателем на константу.

list::const\_iterator c\_itr;

# **87. Операции с итераторами**

Первая void advance (InputIterator& itr, Distance& n) - удобная форма инкрементирования итератора itr на определенное число n (itr+n)

Вторая операция измеряет расстояние между итераторами first и second, возвращая полученное число через ссылку n:

void distance(InputIterator& first, InputIterator& second, Distance& n); (n=second-first)

int main()

{ vector<int> vr(4);

int ms[] = { 2,3,4,5 };

copy(ms,ms+4,vr.begin());

auto itbeg = vr.begin();

advance(itbeg, 2);

copy(itbeg, vr.end(), ostream\_iterator<int>(cout, " "));

return 0;}

Результат

4 5

# **89. Контейнеры последовательностей**

Три контейнера последовательностей – vector, list и deque(двунаправленная очередь). Наиболее полезные методы: front и back − для возврата ссылки на первый и последний элемент в контейнере, push\_back и pop\_back – для вставки и удаления последнего элемента контейнера. Методы vector: erase - удаление произвольного элемента, swap - обмен элементами, begin и end - итератор на начало и конец, clear - очистка контейнера, перезагрузка [].

#include <vector>

int main(){

std::vector<int> v,vv;

v.push\_back(2);

v.push\_back(5);

v.push\_back(7);

vv.push\_back(1);

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.erase(v.begin()+1); //удаляем второй элемент copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.swap(vv);

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

v.clear();

return 0;}

Результат работы 2 5 7

2 7

1

Контейнерный класс list реализуется как двусвязный список. Шаблон класса list предоставляет еще восемь функций-членов:

splice, push\_front, pop\_front, remove, unique, merge, reverse и sort.

#include <list>

int main(){

std::list<int> v,vv;

v.push\_back(2);

v.push\_back(5);

v.push\_back(7);

vv.push\_back(2);

vv.push\_back(3);

v.merge(vv); //объединение листов v.sort();

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.remove(2); //удаление всех 2

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

return 0;}

Результат работы 2 2 3 5 7

3 5 7

Класс deque объединяет многие возможности классов vector и list.

# **90. Контейнер последовательностей vector**

**vector** обеспечивает непрерывную область памяти для размещения данных, следовательно возможен доступ к любому элементу посредством индексации. Если при добавлении выделенной памяти недостаточно, vector выделяет память большего размера, копирует информацию в выделенную область памяти и освобождает старую. vector поддерживает итераторы произвольного доступа.

Методы vector: erase - удаление произвольного элемента, swap - обмен элементами, begin и end - итератор на начало и конец, clear - очистка контейнера, перезагрузка []. front и back − для возврата ссылки на первый и последний элемент в контейнере, push\_back и pop\_back – для вставки и удаления последнего элемента контейнера.

#include <vector>

int main() {

std::vector<int> v,vv;

v.push\_back(2);

v.push\_back(5);

v.push\_back(7);

vv.push\_back(1);

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.erase(v.begin()+1); //удаляем второй элемент copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.swap(vv);

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

v.clear();

return 0;}

Результат работы 2 5 7

2 7

1

# **91. Контейнер последовательностей list**

Контейнерный класс list реализуется как двусвязный список, что позволяет поддерживать двунаправленные итераторы. Эффективно реализует операции вставки и удаления в любое место контейнера. Шаблон класса list предоставляет еще восемь функций-членов: splice, push\_front, pop\_front, remove, unique, merge, reverse и sort.(begin,end)

#include <list>

int main(){

std::list<int> v,vv;

v.push\_back(2);

v.push\_back(5);

v.push\_back(7);

vv.push\_back(2);

vv.push\_back(3);

v.merge(vv); //объединение листов v.sort();

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

v.remove(2); //удаление всех 2

copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

return 0;}

Результат работы 2 2 3 5 7

3 5 7

# **92. Контейнер последовательностей deque**

Класс deque представляет собой двунаправленную очередь. Эффективный индексный доступ. Реализован для эффективных операций вставки в начало и конец. erase - удаление произвольного элемента, swap - обмен элементами, begin и end - итератор на начало и конец, clear - очистка контейнера. front и back − для возврата ссылки на первый и последний элемент в контейнере, push\_back и pop\_back – для вставки и удаления последнего элемента контейнера, push\_front, pop\_front, unique, reverse (begin,end)

#include <deque>

int main()

{ std::deque<int> v;

v.push\_back(2);

v.push\_back(5);

v.push\_back(7);

v.insert(v.begin()+2, 1); //вставка после 2-го элемента 1 copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

copy(v.rbegin(), v.rend(), ostream\_iterator<int>(cout, " "));

cout << endl;

return 0;}

Результат работы 2 5 1 7

7 1 5 2

# **93. Ассоциативный контейнер multiset**

Ассоциативные контейнеры предназначены для обеспечения прямого доступа посредством использования ключей. В STL имеется четыре ассоциативных контейнерных класса: multiset, set, multimap и map.

Классы multiset и set манипулируют множествами значений, одновременно являющихся ключами. При этом multiset допускает одинаковые ключи, а set нет. Классы multimap и map манипулируют множествами значений, ассоциируемых с ключами. При этом multimap допускает хранение одинаковых ключей с ассоциированными значениями, а map нет.

#include <map>

int main()

{ multimap<int, int> mp;

mp.emplace(1, 1);

mp.emplace(1, 4);

mp.emplace(3, 2);

mp.emplace(2, 4);

cout << mp.find(2)->second << endl;

cout << mp.count(1) << endl;

for\_each(mp.begin(), mp.end(), [](pair<int, int> pr)

{cout << pr.second << "\t"; });

return 0;}

Результат работы

4

2

1 4 4 2

# **94. Ассоциативный контейнер set**

Контейнерный класс set используется для обеспечения быстрого сохранения и доступа к уникальным ключам. При попытке поместить в контейнер set дубликат ключа это действие игнорируется без идентификации ошибки. Данный контейнер поддерживает двунаправленные итераторы.

Класс set контейнера стандартной библиотеки C++ используется для хранения и извлечения данных из коллекции. Значения элементов в элементе set являются уникальными и служат ключевыми значениями, по которым данные автоматически упорядочены. Значение элемента в элементе set не может быть изменено напрямую. Вместо этого старые значения необходимо удалить и вставить элементы с новыми значениями.

set<int,std::less<int>>

По-человечески пишется как set<int> S;

1) Добавление. S.insert(1); S.emplace(3);

2) Удаление S.erase(1);

3) Поиск (Возвращает итератор, указывающий на первое расположение элемента в multiset, имеющего ключ, равный указанному ключу.) S.find(3);

4) Начало (Возвращает итератор, указывающий на первый элемент в multiset.) S.begin()

5) Конец (Возвращает итератор, указывающий на место после завершающего элемента в multiset.) S.end()

6) Возвращает итератор, указывающий на первый элемент в обратном multiset. S.rbegin()

7) Возвращает итератор, указывающий на местоположение, расположенное после завершающего элемента в обратном multiset.

S.rend()

8) Число вхождений ключа в multiset (во всех ассоциативных контейнерах) S.count(3)

9) Возвращает итератор, указывающий на первый элемент в контейнере multiset с ключом, который больше или равен указанному ключу. (во всех ассоциативных контейнерах) S.lower\_bound(3)

10) Возвращает итератор, указывающий на первый элемент в контейнере multiset с ключом, который больше указанного ключа. (во всех ассоциативных контейнерах) S.upper\_bound(3)

11) Проверяет, пуст ли multiset S.empty()

# **95. Ассоциативный контейнер multimap**

Ассоциативный контейнер multimap эффективен для быстрого сохранения и нахождения ключей и ассоциированных с ними значений. Многие методы, используемые в контейнерах set и multiset, применимы к контейнерам map и multimap.

Элементами multimap и map являются объекты pair — пары ключей и соответствующих им значений. Порядок сортировки ключей в контейнере определяется компараторным объектом-функцией less<тип>. В контейнере multimap допускается дублирование ключей. Это означает, что несколько значений могут быть ассоциированы с одним ключом (отношение ”один ко многим”). Например, ученик изучает много предметов, один человек может иметь несколько банковских счетов и т.д.

Контейнер multimap поддерживает двунаправленные итераторы. Для работы с контейнерным классом multimap необходимо подключить заголовочный файл <map>.

Multimap - это ассоциативный контейнер, который содержит отсортированный список пар ключ-значение и разрешает несколько записей с одним и тем же ключом. Элементами multimap являются такие объекты pair – пары ключей и соответствующих им значений.

Библиотека <map>

multimap<int,int,std::less<int>> mm; //объявление multimap

multimap <int,int,std::less<int>>::iterator iter; //итератор для multimap

mm.insert(1, 12); //добавление 1 – ключ, 12 – значение

mm.emplace(1, 23) //добавление 1 – ключ, 23 – значение mm.erase(1) //удаление всех значений с ключом 1

Остальные методы такие же как и в multiset

Обращение к значениям через итератор iter->first // ключ

iter->second //значение

# **96. Ассоциативный контейнер map**

Контейнерный класс mар используется для обеспечения быстрого сохранения и доступа к уникальным ключам и соответствующих значений. При этом между ними устанавливается взаимно однозначное соответствие. Попытка поместить в контейнер mар дубликат ключа игнорируется без идентификации ошибки. Контейнер ар поддерживает двунаправленные итераторы.

map<int,int,std::less<int>> mm;

std::map - это отсортированный ассоциативный контейнер, содержащий пары ключ-значение с уникальными ключами.

Библиотека <map>

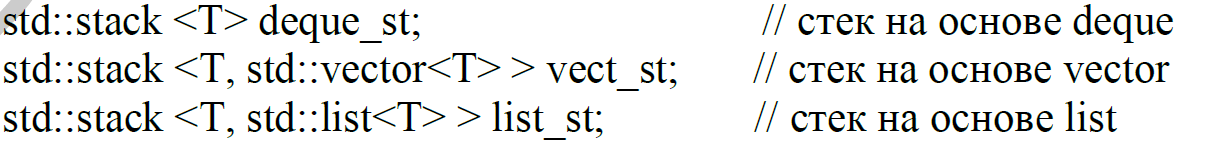
map<int,int> mm; //объявление map

Методы и работа с итератором как в multimap

# **97. Адаптер stack**

Адаптеры не предоставляют реализации фундаментальной структуры данных и не поддерживают работу с итераторами. Это отличает их от контейнеров первого класса. Преимущество класса адаптеров состоит в возможности выбирать требуемую базовую структуру данных. Все три класса адаптеров содержат компоненты-функции push и pop, реализуемые посредством вызова соответствующих функций базового класса. Класс stack обеспечивает возможность вставки и удаления данных в базовой структуре с одной стороны. Адаптер stack может быть реализован с любым из контейнеров последовательностей: vector, list и deque(по умолчанию реализуется через deque).

Для класса stack определены следующие операции (реализуемые через соответствующие функции базового контейнера):push — помещение элемента на вершину стека, pop —удаление элемента с вершины стека, tор — получение ссылки на вершину стека, empty — проверки на пустоту стека и size — получение числа элементов стека.



Библиотека <stack>

stack<int> st

Добавление push(2)

Удаление pop()

Достать вершину стека top()

Пуст ли стек empty()

Количество элементов в стеке size()

# **98. Адаптер queue**

Класс queue предназначен для вставки элементов в конец базовой структуры данных и удаления элементов из ее начала. Адаптер queue реализуется с контейнерами list и deque (по умолчанию).

Наряду с общими для всех классов адаптеров операциями push, pop, empty и size в классе queue имеются операции front — получения ссылки на первый элемент очереди. Back — ссылки на последний элемент очереди.

Библиотека <queue>

queue<int> qe

Добавление push(2)

Удаление pop()

Пуст ли стек empty()

Количество элементов в стеке size()

Получение ссылки на первый элемент front()

Получение ссылки на последний элемент очереди back()

# **99. Адаптер priority\_queue**

Класс priority\_queue используется для вставки элементов в отсортированном порядке в базовую структуру данных и удаления элементов из ее начала. Реализуется с контейнерами vector(По умолчанию) deque.

Элементы в очередь с приоритетом заносятся в соответствии со своим значением. Это означает, что элемент с максимальным значением помещается в начало очереди и будет первым из нее удален, а с минимальным − в конец очереди. Это достигается с помощью метода, называемого сортировкой кучи. Сравнение элементов выполняется функцией-объектом less<Тип> или другой компораторной функцией.

Библиотека <queue>

priority\_queue<int> qe

Добавление push(2)

Удаление pop()

Возвращает константную ссылку на элемент с наивысшим приоритетом(наибольший элемент в верхней части) priority\_queue top()

Пуст ли стек empty()

Количество элементов в стеке size()

# **100. STL-алгоритмы**

Алгоритмы были встроены в контейнерные классы. В STL алгоритмы отделены от контейнеров, что упрощает расширение их числа. Доступ к элементам контейнеров в STL осуществляется посредством итераторов. Каждый алгоритм использует итераторы определённого типа. Например, алгоритм простого поиска (find) просматривает элементы подряд, пока нужный не будет найден. Для такой процедуры вполне достаточно итератора ввода. С другой стороны, алгоритм более быстрого двоичного поиска (binary\_search) должен иметь возможность переходить к любому элементу последовательности и поэтому требует итератора с произвольным доступом. Алгоритмы были встроены в контейнерные классы. В STL алгоритмы отделены от контейнеров, что упрощает расширение их числа. Доступ к элементам контейнеров в STL осуществляется посредством итераторов. Каждый алгоритм использует итераторы определённого типа. Например, алгоритм простого поиска (find) просматривает элементы подряд, пока нужный не будет найден. Для такой процедуры вполне достаточно итератора ввода. С другой стороны, алгоритм более быстрого двоичного поиска (binary\_search) должен иметь возможность переходить к любому элементу последовательности и поэтому требует итератора с произвольным доступом. Вместо менее функциионального итератора можно передавать более функциональный, но не наоборот. Все стандартные алгоритмы описаны в заголовочном файле algorithm, в пространстве имён std.

RanIt − iterator прямого доступа

template <class RanIt>

void sort(RanIt first, RanIt last);

Пример

vector v(4);

v[0] = 3; v[1] = 1; v[2] = 5; v[3] = 2;

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

RanIt − iterator прямого доступа

template <class RanIt>

void sort(RanIt first, RanIt last);

Пример

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

void Print(int x) {cout << x << ‘ ‘;}

int main(){

vector<int> v(4);

v[0] = 3; v[1] = 1; v[2] = 5; v[4] = 2;

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

for\_each(v.begin(), v.end(), Print);

return 0 ;} // вывод: 1 2 3 5

# **101. Алгоритмы сортировки sort**

Алгоритмы сортировки sort, partial\_sort, sort\_heap RanIt − iterator прямого доступа

template <class RanIt>

void sort(RanIt first, RanIt last);

Пример

vector v(4); v[0] = 3; v[1] = 1; v[2] = 5; v[3] = 2;

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

Для использования алгоритма sort с тремя параметрами требуется в качестве третьего аргумента использовать указатель на функцию или функциональный объект. Например, для сортировки в обратном порядке требуется включить заголовок <functional>:

sort(v.begin(), v.end(), greater() );

Алгоритм partial\_sort предназначен для сортировки только части массива. partial\_sort(v1.begin( ), v1.begin( ) + 6, v1.end( ) );

Алгоритм sort\_heap предназначен для сортировки накопителя.

Пример

// **partial\_sort()**

#include <iostream> #include <algorithm>

#include <vector> using namespace std;

int main() {

vector<int> v = {5,8,1,3,6,9,7,2,4};

vector<int>::iterator itr;

//displaying before sorting

cout<<"Before sorting: ";

for (itr = v.begin(); itr != v.end(); ++itr)

cout << \*itr << " ";

partial\_sort(v.begin(), v.begin() + 4, v.end());

cout<<"\nAfter sorting: ";

for (itr = v.begin(); itr != v.end(); ++itr)

cout << \*itr << " ";

return 0;}/\*Before sorting: 5 8 1 3 6 9 7 2 4

After sorting: 1 2 3 4 8 9 7 6 5\*/

//**sort()**

#include <vector>#include <iostream>

#include <algorithm>using namespace std;

int main(){

vector<int> v(4);

v[0] = 3; v[1] = 1; v[2] = 5; v[4] = 2;

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

return 0;

// **sort\_heap()**

#include <iostream> #include <vector>

#include <algorithm> using namespace std;

int main(){

int a[] = {11, 22, 36, 17, 3, 25, 1, 2, 7};

vector<int> v(a, a+9);

for (vector<int>::size\_type i=0; i<v.size(); i++)

cout <<v.at(i)<<" ";

sort\_heap(v.begin(), v.end());

for (vector<int>::size\_type i=0; i<v.size(); i++)

cout <<v.at(i)<<" ";

return 0;}.\*11 22 36 17 3 25 1 2 7

result: 1 2 3 7 17 22 25 36 11\*/

# **102. Алгоритмы поиска find**

Алгоритмы поиска find, find\_if, find\_end, binary\_search В приведенной программе использован алгоритм find, выполняющий поиск в векторе v1 значения 5. itr=find(v1.begin(),v1.end(),5);

Далее в программе использована функция find\_if нахождения первого значения вектора v, для которого унарная предикатная функция fun возвращает true: itr=find\_if(v1.begin(),v1.end(),fun);

унарная предикатная функция fun bool fun(T i){return i%2==0;}

Каждый из алгоритмов find и find\_if возвращает итератор ввода на найденный элемент либо (если элемент не найден) итератор, равный v.end(). binary\_search(v1.begin(),v1.end(),3)

для поиска значения 3 в векторе v1 использована функция binary\_search. При этом последовательность элементов вектора в анализируемом диапазоне должна быть отсортирована в возрастающем порядке. Функция возвращает значение bool.

Пример с использованием алгоритмов:

#include <vector> #include <iostream>

#include <algorithm> #define T int

using namespace std;

bool fun(T i){return i%2==0;}

int main(){

T m1[] = {5,3,4,7,3,12};

vector<T> v1(m1,m1+sizeof(m1)/sizeof(T));

ostream iterator<T>out(cout, “ ”);

vector<T>:: iterator itr; cout << “v1: ”;

copy(v1.begin(),v1.end(), out);

itr = find(v1.begin(),v1.end(),5);/\* if itr!= v1.end() – найдено в позиции\*/

itr = find\_if(v1.begin(),v1.end(),fun());/\* if itr!= v1.end() – найден первый чёт эл. вектора\*/

if(binary\_serach(v1.begin(),v1.end(),3);

cout << “число 3 найдено” << endl;

else cout <<“число 3 не найдено”<<endl; retutn 0;}

# **103. Алгоритмы swap**

Алгоритмы swap, iter\_swap и swap\_ranges Данная группа алгоритмов предназначена для выполнения перестановки элементов контейнера. Ниже приведены прототипы данных алгоритмов: template<class A, class B> void swap(const vector<A,В> & L, const vector<A, B> & R); или по-человечески swap(a,b)

Аргументами алгоритма swap являются ссылки на элементы для замены template<class FIter, class SIter>

void iter\_swap(FIter firstIter, SIter secondIter); или по-человечески

deque<int> deq;

iter\_swap ( deq.begin( ), --deq.end( ) ); //– стоит, потому что deq.end( ) указывает на следующий за последним

swap\_ranges

Меняет местами элементы одного диапазона с элементами другого диапазона такого же размера.

Аргументами алгоритма являются два прямых итератора на элементы. template<class FIter, class SIter>

swap\_ranges(FIter firstIter, FIter lastIter, SIter xIter);

алгоритм swap\_ranges используется для перестановки элементов от firstIter до lastIter с элементами начиная с xIter.

или по-человечески vector<int> v1; vector<int> v2;

swap\_ranges ( v1.begin( ), v1.end( ), v2.begin( ) );

Пример

#include <iostream> #include <vector> #include <algorithm>

using namespace std;

int main() { // **swap\_ranges**

vector<char> V1 = { 'a', 'b', 'c', 'd', 'e', 'f' };

vector<char> V2 = { '0', '1', '2', '3', '4' };

vector<char>::iterator itBegin = V1.begin();

vector<char>::iterator itEnd = V1.begin();

itEnd++; itEnd++; // itEnd -> 'c'

vector<char>::iterator itBegin2 = V2.begin();

itBegin2 += 3; // itBegin2 -> '3'

swap\_ranges(itBegin, itEnd, itBegin2);

cout << "V1 => ";

for (char c : V1)

cout << c << " ";

cout << endl; return 0;} /\* V1 => 3 4 c d e f V2 => 0 1 2 a b \*/

// **swap**

vector<int> v1 = { 1, 3, 8, 4 };

vector<int> v2 = { 7, 9, 0 };

swap(v1, v2);

cout << "v1 => ";

for (int i : v1)

cout << i << " ";

cout << endl;

cout << "v2 => ";

for (int i : v2)

cout << i << " ";

cout << endl;/\* v1 => 7 9 0 v2 => 1 3 8 4\*/

// **iter\_swap**

vector<double> V1 = { 1.1, 2.2, 2.7, 2.8, 3.2 };

cout << "V1 => ";

for (double x : V1)

cout << x << " ";

cout << endl;

vector<double>::iterator it1 = V1.begin();

vector<double>::iterator it2 = V1.end();

iter\_swap(it1, it2--);

cout << "V1 => ";

for (double x : V1)

cout << x << " ";cout << endl;}

# **104. Алгоритм merge**

Объединяет все элементы из двух исходных упорядоченных диапазонов в один упорядоченный диапазон назначения.

vector<int> v1a, v1b, v1;

merge ( v1a.begin( ), v1a.end( ), v1b.begin( ), v1b.end( ), v1.begin( ) );

Первый аргумент – с какого элемента из первого диапазона начнем Второй аргумент – до какого элемента из первого диапазона объединять Третий аргумент – с какого элемента из второго диапазона начнем Четвертый аргумент – до какого элемента из второго диапазона объединять Пятый аргумент – в какой будем объединение записывать

Пример

char s[] = "aeiou";

int len = strlen(s);

list<char> list1(&s[0], &s[strlen(s)]);

// Initialize deque1 with 26 copies of the letter x:

deque<char> deque1(26, "x");

// Merge array s and list1, putting result in deque1:

merge(&s[0],&s[len],list1.begin(),list1.end(),

deque1.begin());

deque<char>::iterator i;

cout.precision(10);

for (i = deque1.begin(); i != deque1.end(); ++i)

cout << \*i << endl;

return 0;} /\* вывод: a a e e i i o o u u x x x x x x x x x x x x x x x x\*/

# **105. Алгоритм for\_each**

for-each принимает 3 параметра: 1)итератор на начало, 2)итератор на конец, 3)лямбда-функцию или указатель на функцию

Пример:

#include <map> using namespace std; int main()

{

multimap<int, int> mp; mp.emplace(1, 1);

mp.emplace(1, 4);

mp.emplace(3, 2);

mp.emplace(2, 4);

cout << mp.find(2)->second << endl; cout << mp.count(1) << endl;

for\_each(mp.begin(), mp.end(), [](pair<int, int> pr)

{cout << pr.second << "\t"; }); return 0;}

Данная программа выведет значения мультимапа.

Пример

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

void Print(int x) {cout << x << ‘ ‘;}

int main(){

vector<int> v(4);

v[0] = 3; v[1] = 1; v[2] = 5; v[4] = 2;

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

for\_each(v.begin(), v.end(), Print);

return 0 ; /”\* вывод: 1 2 3 5\*/}

# **106. Лямбда-функции**

Лямбда-выражение – более краткий синтаксис для определения объектов-функций. Имеет формат [ ] ( ) { } .

Пример: []() { std::cout << "Hello" << std::endl; }

Также можно сократить: []{ std::cout << "Hello" << std::endl;}

[=]() mutable{} изменяет значение только в своём {}

int n{10};

auto increment = [=]() mutable {

n++;

cout << n << endl; // n = 11};

increment();

cout << n << endl; // n = 10

Получение определенных переменных

По умолчанию выражения [=]/[&] позволяют захватить все переменные из окружения. Но также можно захватить только определенные переменные. Чтобы получить внешние переменные, применяется выражение [&имя\_переменной]:

[=, &m, &n]

// все по значению, а m и n - по ссылке

// можно [this]

Лямбда-функцию можно передать в качестве значения параметра в функцию:

void do\_operation(int a, int b, int (\*op)(int, int)){

std::cout << op(a, b) << std::endl; }

int main(){

auto sum { [](int a, int b) {return a + b;} };

auto subtract { [](int a, int b) {return a - b;} };

do\_operation(10, 4, sum); // 14

do\_operation(10, 4, subtract); // 6}

Универсальное лямбда-выражение принимает как минимум один auto/auto&/const auto& параметр:

auto add = [](auto a, auto b) {return a + b;};

auto print = [](const auto& val){ cout << val << endl; };

# **107-108. Конструктор перемещения, оператор присваивания перемещения, семантика перемещения**

lvalue ссылки – «обычные» ссылки

rvalue ссылки позволяют избежать логически ненуженого копирования

A a;

A& a\_ref1 = a; //lvalue

A&& a\_ref2 = a; //rvalue

Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

A& a\_ref3 = A(); // Ошибка!

A&& a\_ref4 = A(); // Ok

Устранение побочных копий

Пример 1: swap объектов через копирование

template <class T> swap(T& a, T& b){

T tmp(a); // полное копирование а

a = b; // полное копирование б

b = tmp; // полное копирование тмп}

Пример 2: swap объектов через перемещение

template <class T> swap(T& a, T& b){

T tmp(std::move(a)); // «передача» а

a = std::move(b); // «передача» б

b = std::move(tmp); // «передача» тмп}

Этот вызов move() возвращает значение объекта, переданного в качестве параметра, но не гарантирует сохранность этого объекта. К примеру, если в качестве параметра в move() передать vector, то можно обоснованно ожидать, что после работы функции от параметра останется вектор нулевой длины, так как все элементы будут перемещены, а не скопированы. Другими словами, перемещение – это считывание со стиранием (destructive read).

Главная задача rvalue ссылок состоит в том, чтобы позволить нам реализовывать перемещение без переписывания кода и издержек времени выполнения (runtime overhead).

Move

Функция move в действительности выполняет весьма скромную работу. Её задача состоит в том, чтобы принять либо lvalue, либо rvalue параметр, и вернуть его как rvalue без вызова конструктора копирования:

template <class T>

typename remove\_reference<T>::type&&

move(T&& a){

return a; }

Теперь всё зависит от клиентского кода, где должны быть перегружены ключевые функции (например, конструктор копирования и оператор присваивания), определяющие будет ли параметр lvalue или rvalue. Если параметр lvalue, то необходимо выполнить копирование. Если rvalue, то можно безопасно выполнить перемещение.

Пусть у нас есть объект clone\_ptr с реализованными конструкторами копирования и перемещения:

T\* ptr;

//копирование

clone\_ptr(const clone\_ptr& p) :

ptr(p.ptr ? p.ptr->clone() : 0) {}

//перемещение

clone\_ptr(clone\_ptr&& p) :

ptr(p.ptr) {p.ptr = 0;}

.clone() – метод, возвращающий копию значения.

Тогда, если нам надо передать значение, есть два варианта:

clone\_ptr<base> p1(new derived);

clone\_ptr<base> p2 = p1;

//для p2 вызывается куча конструкторов

или

clone\_ptr<base> p1(new derived);

// ...

clone\_ptr<base> p2 = std::move(p1);

// в р2 быстро передалась ссылка rvalue р1

Пример реализации конструктора и оператора перемещения для составных классов

class Derived : public Base{

public:

//…

Derived(Derived&& x) // объявлен как rvalue

: Base(std::move(x)),

vec(std::move(x.vec)),

name(std::move(x.name)) { }

Derived& operator=(Derived&& x) // объявлен как rvalue

{Base::operator=(std::move(x));

vec = std::move(x.vec);

name = std::move(x.name);

return \*this;}};

Перемещаемые, но не копируемые типы

К некоторым типам семантика копирования не применима, но их можно перемещать. Например:

• fstream

• unique\_ptr (не разделяемое и не копируемое владение)

• Тип, представляющий поток выполнения

ifstream find\_and\_open\_data\_file(/\* ... \*/);

...

ifstream data\_file = find\_and\_open\_data\_file(/\* ... \*/);

// Никаких копий!

В этом примере базовый дескриптор файла передан из одного объекта в другой, т.к. источник ifstream является rvalue. В любом момент времени есть только один дескриптор файла, и только один ifstream владеет им.

Перемещаемый, но не копируемый тип также может быть помещён в стандартные контейнеры. Если контейнеру необходимо “скопировать” элемент внутри себя (например, при реалокации vector), он просто переместит его вместо копирования.

vector<unique\_ptr<base>> v1, v2;

v1.push\_back(unique\_ptr<base>(new derived()));

// OK, перемещение без копирования

...

v2 = v1;

// Ошибка времени компиляции! Это не копируемый тип.

v2 = move(v1);

// Нормальное перемещение.

При использовании Lvalue ссылок может возникнуть проблема с const/не const аргументами. При использовании семантики пермещения и std::forward() в частности можно избежать этой дурки:

template <class T, class A1>

std::shared\_ptr<T> factory(A1&& a1){

return std::shared\_ptr<T>(new T(std::forward<A1>(a1)));}

Здесь, forward сохраняет lvalue/rvalue параметр, который был передан factory. Если factory был передан rvalue, то при помощи forward и конструктору T будет передан rvalue. Точно так же, если lvalue параметр передан factory, он же будет передан конструктору T как lvalue.

# **109. Паттерны проектирования**

Паттернами проектирования называют решения часто встречающихся проблем в области разработки программного обеспечения. Паттерны проектирования не являются готовыми решениями, которые можно трансформировать непосредственно в код, а представляют общее описание решения проблемы, которое можно использовать в различных ситуациях.

Существуют несколько типов паттернов проектирования, каждый из которых предназначен для решения своего круга задач:

• Порождающие паттерны, предназначенные для создания новых объектов в системе.

• Структурные паттерны, решающие задачи компоновки системы на основе классов и объектов.

• Паттерны поведения, предназначенные для распределения обязанностей между объектами в системе.

Порождающие паттерны отвечают за создание существующих объектов и возможность добавления новых.

В таких случаях на помощь приходит фабрика объектов, локализующая создание объектов. Работа фабрики объектов напоминает функционирование виртуального конструктора, - мы можем создавать объекты нужных классов, не указывая напрямую их типы. В самом простом случае, для этого используются идентификаторы типов. Следующий пример демонстрирует простейший вариант фабрики объектов - фабричную функцию.

enum Warrior\_ID { Infantryman\_ID=0, Archer\_ID, Horseman\_ID };

Warrior \* сreateWarrior( Warrior\_ID id ){

Warrior \* p;

switch (id)

{ case Infantryman\_ID:

p = new Infantryman();

break;

case Archer\_ID:

p = new Archer();

break;

case Horseman\_ID:

p = new Horseman();

break;

default:

assert( false);}

return p;}

Эта функция получает в качестве аргумента тип объекта, который нужно создать, создает его и возвращает соответствующий указатель на базовый класс. Несмотря на очевидные преимущества, у этого варианта фабрики также существуют недостатки. Например, для добавления нового вида боевой единицы необходимо сделать несколько шагов - завести новый идентификатор типа и модифицировать код фабричной функции createWarrior( ).

Структурные паттерны рассматривают вопросы о компоновке системы на основе классов и объектов. При этом могут использоваться следующие механизмы:

• Наследование, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.

• Композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Например, паттерн Composite группирует схожие объекты в древовидные структуры. Рассматривает единообразно простые и сложные объекты.

Паттерны поведения рассматривают вопросы о связях между объектами и распределением обязанностей между ними. Для этого могут использоваться механизмы, основанные как на наследовании, так и на композиции.

Например, паттерн Iterator предоставляет механизм обхода элементов составных объектов (коллекций) не раскрывая их внутреннего представления.

**110. Паттерн Абстрактная фабрика**

**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит:

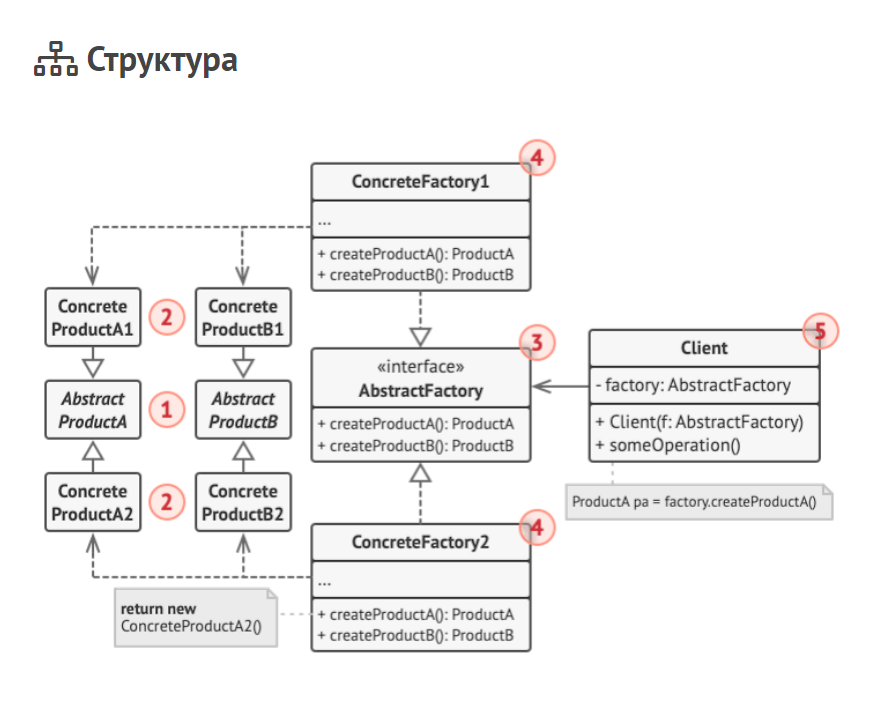
1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.
2. Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерне.

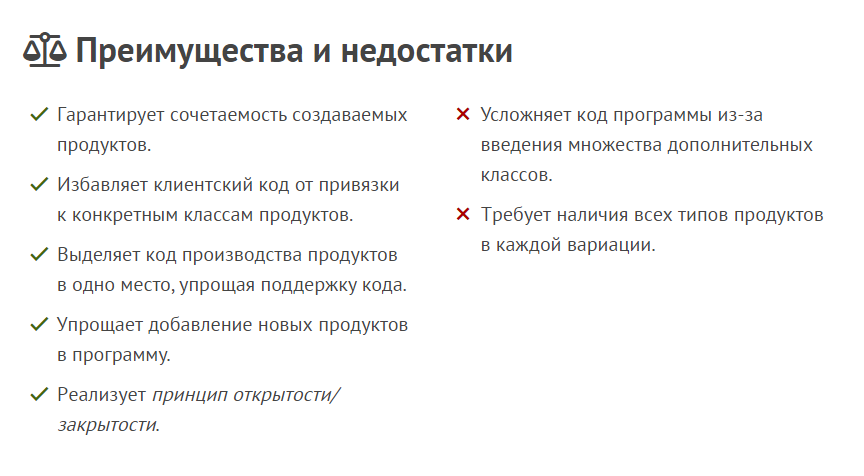
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.

Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные типы** продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн, ДиваныМодерн и СтоликиМодерн.





# **111. Умные указатели**

В С++ есть два типа выделения памяти: статическое и динамическое.

При статическом выделении память освобождается при выходе за scope. При динамическом же выделении память необходимо самостоятельно освобождать с помощью **delete.** Однако у этого есть свои проблемы:

void my\_func()

{ int\* valuePtr = new int(15);

int x = 45;

// ...

if (x == 45)

return;

// here we have a memory leak, valuePtr is not deleted

// ...

delete valuePtr;}

Чтобы избавиться от этих проблем, добавили умные указатели. Их есть три вида: **unique\_ptr, shared\_ptr, weak\_ptr.** Объекты под них принято создавать с помощь. функций **make\_unique()** и **make\_shared().**

Рассмотрим их по отдельности:

1. unique\_ptr является **единственным** указателем на созданный объект. При попытке создания второго указателя на ту же область памяти будет ошибка. Очевидно, что, как следствие, у него **отсутствует конструктор копирования.** Для использования в функциях объектов, созданных через unique\_ptr, есть метод **.get()** и std::experimental::observer\_ptr. Память объекта unique\_ptr очищается после последнего использования.
2. При создании объекта через shared\_ptr на него могут указывать несколько указателей типа shared\_ptr либо weak\_ptr. Создавать weak\_ptr можно только через shared\_ptr. Система ведёт счётчик кол-ва shared\_ptr, указывающих на блок памяти, и очищает её только после **последнего вызова последнего shared\_ptr.**
3. Таким образом, **weak\_ptr создан для указания на объекты shared\_ptr, когда не обязательно их наличие.**

Пример:

int main() {

shared\_ptr Student1 = new Student;

shared\_ptr Student2 = new Student;

Student1.friend = Student2;

Student2.friend = Student1;

return 0;}

В таком случае, если поле .friend – shared\_ptr, то память, выделенная под Student1 и Student2 не будет выделена, т.к. на каждый из них указывает по 2 объекта shared\_ptr, ни один из которых не может быть удалён. В случае, если friend – weak\_ptr, очищение пройдёт успешно, т.к. **weak\_ptr не учитывается в счётчике.**