Санкт-Петербургский
государственный университет телекоммуникаций
им. проф. М. А. Бонч-Бруевича

С. В. Козин

ЛЕКЦИИ ПО ДИСЦИПЛИНЕ ОБЪЕКТНО – ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Структура дисциплины

Дисциплина «Объектно-ориентированное программирование» (ООП) - это одно семестровый курс. По этой дисциплине предусматривается следующие виды занятий. Один раз в неделю читаются лекции, и проводится одна лабораторная работа. По дисциплине предусмотрена курсовая работа. Форма отчетности по дисциплине – экзамен.

Одним из наиболее востребованных объектно-ориентированных языков является язык C++. Это положение объясняет выбор данного языка для изучения объектно-ориентированного программирования. В связи с этим представляется начать изложение материала с рассмотрением структуры языка C++.

Структура языка C++

В языке C++ принято выделять следующие три направления:

- Процедурное программирование.
- Объектно-ориентированное программирование.
- Обобщенное программирование.

Приведем общую характеристику каждого направлений языка C++.

Процедурное программирование

Процедурную составляющую языка C++ иногда называют улучшенным языком C. К таким улучшениям можно отнести повышение типизации. Прежде всего, здесь следует отметить отмену возможности выполнения компиляции вызова функции, для которой при отсутствии прототип. Компилятор языка C в этой ситуации ограничивался выдачей предупреждения. Компилятор же языка C++ выдает сообщение об ошибке, что делает невозможным успешное компилирование исходного кода модуля. Это позволяет выявить ошибки, связанные с несоответствием между формальными и фактическими параметрами.

Другим важным элементом, улучшающим типизацию, является отмена возможности автоматического преобразования типа при присваивании типизированному указателю значения нетипизированного указателя. Например, в языке C является допустимой следующая последовательность инструкций:

```
int n;
printf("n=");
scanf("%d", &n);
double *p = malloc(sizeof(double) * n);
```

В последней строке приведенного выше код тип переменной `p` (**`double*`**) не совпадает с типом значения (**`void*`**), которое возвращает функция `malloc()`. Компилятор языка C при трансляции оператора присваивания выполнит автоматическое преобразование типов. Компилятор языка C++ такое преобразование типа выполнить не имеет права и поэтому выдаст сообщение об ошибке (`error`).

Другим важным новшеством, которое можно воспользоваться, работая в рамках процедурного программирования, являются ссылки. Отметим, что ссылки широко применяются и при объектно-ориентированном программировании. Предварительное знакомство с этим типом данных является необходимым. В связи с тем обстоятельством, что при изложении дисциплины «Программирование на языке высокого уровня» ссылки не рассматривались, представляется целесообразным восполнить этот пробел.

Ссылку иногда называют постоянно разыменованным указателем. Это позволяет ожидать, что между ссылками и указателями в языке C++ имеется некоторая связь. Такая связь действительно имеет место. Однако различия между ними очень значительны. К таким отличиям относится следующее:

- Ссылка должна быть обязательно инициализирована во время ее инициализации.
- Отсутствуют операции, которые можно выполнять с данными ссылочного типа.

Объявление ссылок строится по правилам, что и объявления указателей, если в объявлении указателя оператор разыменования * заменить оператором взятия адреса &. Приведем пример программного кода, содержащего объявление ссылки.

```
int n = 10;
int &r = n;
```

Во второй строке рассматриваемого кода объявлена переменная r, имеющая тип ссылки на объект типа int. Эта переменная во время объявления инициализирована именем уже существующей переменной n. Теперь переменная r может использоваться в качестве синонима (второго имени) переменной n. Все операции, которые программист выполнит с переменной r, будут относиться к ячейке памяти, выделенной для хранения значения переменной n.

Ссылки широко используются в языке C++ для организации способа передачи параметров в функцию, который в программировании принято называть передачей по ссылке. В языке C передачи по ссылке нет. Такой способ моделируется с помощью указателей. В качестве примера рассмотрим задачу обмена значений двух переменных, имеющих тип double. Решение задачи оформим в виде функции. Приведем два варианта реализации такой функции. В первом из них будут использованы указатели, а во втором – ссылки.

```
void swap(double *a, double *b)
{
   double temp = *a;
   *a = *b;
   *b = temp;
}

void swap(double &a, double &b)
{
   double temp = a;
```

```
    a = b;
    b = temp;
}

int main()
{
    double x = 2;
    double y = 3;

    swap(&x, &y);
    printf("x= %0.4g   y= %0.4g\n", x, y);

    swap(x, y);
    printf("x= %0.4g   y= %0.4g\n", x, y);
    return 0;
}
```

Перейдем к обсуждению приведенного выше кода. Следует обратить внимание на то обстоятельство, что обе функции, решающие поставленную задачу имеют одно и то же имя. Это недопустимо в языке С и допустимо в языке С++. Дело в том, что язык С++ разрешает выполнять так называемую перегрузку функций. Перегружаемые функции имеют одинаковые имена, но должны различаться сигнатурой. Сигнатура функции определяется списком объявлений ее параметров. При этом важным является количество параметров и их типы. Обе наши функции swap() различаются типом своих параметров.

Можно построить простую модель, объясняющую принцип работы функции swap(double &a, double &b.). Во время вызова этой функции параметры ссылки a и инициализируются именами своих аргументов x и y. При выполнении тела функции все обращения к параметрам функции будут направляться к их синонимам, которые находятся в вызывающей функции. Программный код функции swap(), использующей ссылки и ее вызова, выглядит более простым по сравнению с вариантом реализации этой функции, использующей указатели.

Объектно-ориентированное программирование

Объектно-ориентированный компонент языка С++ обладает весьма мощными возможностями. В языке С++ имеется развитый аппарат классов, позволяющий реализовать пользовательские типы данных. Перегрузка встроенных операторов для пользовательских типов позволяет наделить эти типы всеми возможностями, которые предусмотрены в языке для встроенных типов. Это дает основания считать систему встроенных типов языка С++ расширяемой.

Класс языка С++ является обобщением структур языка С. От структур языка С класс отличается в двух отношениях. Во-первых, в классе объединяются не только данные, но функции. Во-вторых, элементы класса наделяются правами доступа. Объединение данных и функций позволяет

связать данные и те операции, которые необходимо с ними выполнять, Права доступа позволяют инкапсулировать (скрыть) реализацию класса.

В языке C++ пользовательские типы могут быть наделены свойствами наследования и полиморфизма. Это позволяет программисту разрабатывать иерархические программные системы.

Важным является то обстоятельство, что имеется стандарт, утвержденный в 1998 поду, который определяет синтаксис языка C++ и его стандартную библиотеку.

Отметим, что в объектно-ориентированном программировании в центре внимания являются данные, а не алгоритмы как это имеет место в процедурном программировании.

Обобщенное программирование

В основе обобщенного программирования лежит использование так называемых шаблонов функций и шаблонов классов. Применение таких шаблонов связано с тем обстоятельством, что алгоритмы обработки данных часто слабо зависят от типа данных, которые они обрабатывают. Информацию о типе данных, которые следует обрабатывать, в этом случае удобно передавать через параметры, которые носят название обобщенных типов. Приведем пример шаблона функции, предназначенного для обмена значений двух переменных.

```
template<class T>
void swap(T &a, T &b)
{
   T temp = a;
   a = b;
   b = temp;
}

int main()
{
   double x = 2;
   double y = 3;
   swap(x, y);
   printf("x= %0.4g   y= %0.4g\n", x, y);

   int n = 4;
   int m = 5;
   swap(n, m);
   printf("n= %d   m= %d\n", n, m);

   return 0;
}
```

Приведенный выше программный код начинается с определения шаблона функции swap(). Шаблон функции отличается от определения обычной функции наличием дополнительного префикса, который начинается с зарезервированного слова template. В угловых скобках содержится так

называемый список параметров шаблона. В рассматриваемом примере этот список содержит только один элемент, состоящий из зарезервированного слова class имени параметра шаблона T. Зарезервированное слово class показывает, что параметр шаблона T относится к категории обобщенных типов (имеются и другие разновидности шаблонных параметров).

В вызывающей функции, которой

НА ПУТИ К ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПРОГРАММИРОВАНИЮ

Общеизвестно, что сложность программирования в первую очередь обусловлена сложностью решаемых задач. Основным способом преодоления этих трудностей является декомпозиция поставленной задачи. Декомпозиция состоит в замене исходной задачи совокупностью других менее сложных задач, совместное решение которых должно дать ожидаемый результат.

Следует учитывать тот факт, что не любая декомпозиция приводит к успеху. На практике часто приходиться сталкиваться с ситуацией, когда части программы, отвечающие за решение отдельных подзадач, работают, а вся программа в целом нет. Это имеет место при наличии сильных взаимосвязей между отдельными частями программы.

Применение абстракции позволяет сделать решение отдельных задач относительно независимыми. Действительно, абстрагирование связано с выделением в некотором явлении, событии или объекте существенных характеристик. Это в конечном итоге должно способствовать уменьшению взаимосвязей между отдельными компонентами программы.

В ранних языках программирования нашла применение так называемая абстракция действия, или процедурная абстракция. Абстракция действия реализуется с помощью подпрограмм (функциями языка C).

Существуют два основных вида абстракций:

- Абстракция действия (процедурная абстракция).
- Абстракция сущности (абстрактный тип данных).

Абстракция действия, или процедурная абстракция реализуется в языке C++ с помощью аппарата функций. Используя функции, программист имеет возможность расширять то множество встроенных операций, которые предусмотрены языком.

При использовании процедурного программирования является единственным поддерживаемым способом абстрагирования. Рассмотрим факторы, обеспечивающие поддержку языком C абстракции действия:

- наличие специальной синтаксической конструкции, предназначенной для реализации абстракции (определение функции).
- возможность отделения интерфейса от реализации.
- сокрытие реализации алгоритма, реализующего действие, от клиента.

Абстракция сущностей тесно связана с проблемой создания пользовательских типов данных. Дело в том, что на практике часто возникает

потребность в моделировании понятий, существующих в той предметной области, для которой разрабатывается программа. Это и приводит к необходимости иметь аппарат, позволяющий создавать пользовательские типы данных. Представляет интерес рассмотреть вопрос следующий вопрос: можно или нет на языке типа языка С реализовать абстракцию сущностей. Из изложенного ранее, можно отметить важный факт, который может негативно сказаться на попытке реализовать абстракцию действия: в языке С отсутствует синтаксическая конструкция, предназначенная для этих целей.

Вначале рассмотрим очень важный вопрос, каким образом можно описать абстракцию сущностей. Для этого обратимся к понятию, которое часто используется в современном программировании. Речь идет об абстрактном типе данных.

Абстракция сущностей и процедурный язык программирования

Рассмотрим возможность реализации абстракции сущности (пользовательский тип) средствами процедурного языка программирования (например, языка С). Основная сложность состоит в отсутствии в языке С специальной синтаксической конструкции, предназначенной для этой цели. Будем исходить из общих соображений. Тип данных задается двумя множествами:

- Множеством значений, которые могут принимать данные рассматриваемого типа.
- Множеством операций, которые можно применять к таким данным.

Для обеспечения требуемого множества значений атрибуты объектов разрабатываемого типа будем инкапсулировать в структуре (struct). Здесь за объектом сохраняется привычный для языка С смысл. Множество операций будем моделировать с помощью обычных функций языка С. Объект, для которого будет вызываться функция, необходимо передавать в функцию через отдельный параметр. Тип этого параметра – указатель на структуру, в которой будут инкапсулированы данные.

Разрабатываемый тип оформим в виде модуля. В заголовочном файле модуля следует объявить структуру как тип и привести прототипы всех функций, моделирующие операции разрабатываемого типа. В файле реализации модуля следует привести их определения.

Приведем пример. Разработаем пользовательский тип данных стек. Выберем одну из простейших реализаций стека. Стек будет реализован с помощью статического массива. Размер памяти массива будет определяться символической константой, задаваемой с помощью директивы define (#include MAXSIZE 20).

В структуре будем инкапсулировать две величины:

- Целочисленное значение вершины стека (int top)
- Объявление массива. Для определенности предположим, что стек будет работать с данными типа double (double ar[MAXSIZE]).

В число операций, с которыми должны работать переменные разрабатываемого пользовательского типа, следует включить:

- init() – инициализация стека (необходимо задать начальное значение переменной top).
- push() – затолкнуть в стек.
- pop() – вытолкнуть из стека.
- top() – возвращает значение, находящееся на вершине стека.
- isEmpty() – проверка состояния «Стек пуст»
- isFull() – проверка состояния «Стек полон»

Исходный текст модуля, предназначенного для реализации разрабатываемого пользовательского типа, приведен ниже.

```
// файл Stack.h
#define MAXSIZE 20
struct Stack
{
    int top;
    double ar[MAXSIZE];
};

void init(Stack *pstk);
void push(const Stack *pstk, double item);
double pop(Stack *pstk);
double top(const Stack *pstk);
bool isEmpty(const Stack *pstk);
bool isFull(const Stack *pstk);

// Файл Stack.c
#include “Stack.h”

void init(Stack *pstk)
{
   pstk-> top = -1;
}

void push(const Stack *pstk, double item)
{
      pstk->ar[++pstk->top] = item;
}

double pop(Stack *pstk)
{
      return pstk->top--;
}

double top(const Stack *pstk)
{
   return pstk->ar[pstk->top];
}

bool isEmpty(const Stack *pstk)
{
```

```
   if(pstk->top == -1)
      return true;
   return false;
}

bool isFull(const Stack *pstk)
{
   if(pstk->top >= SIZEMAX - 1)
      return true;
   return false;
}

// Файл main.c   Клиентский код
#include <stdio.h>
#include <stdlib.h>
#include "Stack.h"

Stack stk1;
init(&stk1);

int n;
printf("n=");
scanf("%d", &n);

for(int i = 0; i < n; i++)
{
    if(!isFull(&stk1)
       push(&stk1, i * i);
    else
       {
           printf("Переполнение стека\n");
           exit(1);
       }
}

int m;
printf("m=");
scanf("%d", &m);


for(int i = 0; i < n; i++)
{
    if(!isEmpty(&stk1()))
       printf("%0.4g\n", pop(&stk1));
    else
       {
           printf("Стек пуст\n");
           exit(1);
       }
}
return 0;
}
```

Создается иллюзия того, что поставленная задача решена. Удалось реализовать тип данных, который имеет многие элементы пользовательского типа:

- имеется возможность создавать переменные заданного типа.
- к этим переменным можно применять операции, присущие стеку; реализован интерфейс стека.

Однако имеется и ряд существенных недостатков:

- реализация типа оказалась незащищенной. Клиентский код имеет доступ к полям переменной. Отсутствует инкапсуляция абстракции.
- функции, реализующие интерфейс, находятся в глобальном пространстве имен.

Абстрактный тип данных

Несмотря на широкое использование этого понятия, его единая трактовка отсутствует. Некоторые авторы под абстрактным типом данных (АТД) понимают математическую модель абстракции. Несомненно, построение математической модели абстракции позволило бы построить ее описание. Наиболее последовательно этот подход представлен в книге Бертрана Мейера[21].

Спецификация АТД, которая предлагается Бертраном Мейером в несколько упрощенном виде состоит из четырех разделов:

- наименование типа.
- прототипы функций.
- аксиомы.
- предусловия.

Назначение первого раздела не требует пояснения. Второй раздел определяет интерфейс, предоставляемый АТД. Третий раздел предназначен для описания функциональности АТД. Последний раздел спецификаций определяет условия, которые должен обеспечить клиент АТД

Реализация абстракции сущности требует наличия аппарата, который позволял бы программисту разрабатывать собственные типы данных. Такой аппарат предоставляет объектно-ориентированное программирование. Попытаемся ответить на вопрос: можно ли оставаясь в рамках процедурного языка, например, языка С, реализовать пользовательский тип данных.

Для работы с абстракциями общего вида, которыми являются абстракции сущности, часто используются понятие абстрактного типа данных.

Рассмотрим несколько упрощенно процесс абстрагирования. В нем можно выделить две задачи. Целью первой из них является отделение в отделении интерфейса от реализации. Вторая задача состоит в инкапсуляции реализации, которая сделала бы реализацию недоступной для клиента.

Понятие абстрактного типа данных (АТД) широко применяется на практике. Однако различные авторы по-разному определяют это понятие. Некоторые из них под АТД понимают математическую модель абстракции [20]. Очень важным является то обстоятельство, что нет необходимости пересматривать модель при смене реализации и более того это не надо делать при переходе на другой язык программирования. Построив такую модель, можно познакомиться с интересующей вас сущностью. Такое определение АТД чрезвычайно полезно. Полезность его в полной независимости от реализации. Однако имеется и существенный недостаток. Оно не позволяет ответить на вопрос: можно ли реализовать рассматриваемую абстракцию на конкретном языке программирования.

```
/* Файл Stack.h    */
struct Stack
{
    double *ptemp;
    int     size;
    int      top;
};

void allocate_memory(struct Stack* p, int size)
{
      p -> size = size;
      p -> ptemp = (double*)malloc(sizeof(double) * size);
      p -> top   = -1;
}

void free_memory(struct Stack* p)
{
     free(p -> ptemp);
}
```

ОРГАНИЗАЦИЯ КЛАССА

В языке C++ классы предназначены для частичной или полной реализации пользовательского типа (абстрактного типа данных). Различают конкретные и абстрактные классы. Конкретные классы предназначены для создания объектов. С помощью конкретных классов создаются пользовательские типы данных. Абстрактные классы не позволяют создавать объекты. В настоящем разделе рассматриваются конкретные классы.

Класс языка C++ является развитием (обобщением) структур языка C. Класс отличается от структуры в следующих отношениях:

- В классе объединяются не только данные (как это имеет место в структуре), но и функции.
- Элементы класса наделяются правами доступа.
- Класс наделяется отдельной областью видимости.

Определение и объявление класса

Определение класса полностью описывает (специфицирует) класс, а объявление специфицирует категорию имен, к которой относится объявляемое имя. В тех точках, где в клиентском коде создаются экземпляры класса (объекты), должно доступно определение класса. Для объявления указателя на класс достаточно располагать объявлением класса.

В связи с тем, что класс языка C++ является обобщением структур языка C его объявления и определения имеют много общего с соответствующими конструкциями для языка C.

Первое отличие классов языка C++ от структур языка C, на котором следует остановиться, состоит в следующем. Для объявлений и определений структур языка C используется зарезервированное слово struct. Для этих же целей в классе языка C++ используются три зарезервированные слова. Прежде всего, это новое зарезервированное слово class. Этого слова нет в языке Си. Оно введено в язык C++ специально для работы с классами. Однако наряду с этим зарезервированным словом допустимо использовать два зарезервированные слова языка C. Речь идет о словах struct и union.

Отметим, что классы, построенные с применением зарезервированных слов class и struct, имеют одинаковые возможности. Отличия между ними состоит в трактовке прав доступа по умолчанию. Классы, построенные с применением зарезервированного слова union, располагают существенно меньшими возможностями. Их следует применять в тех же ситуациях, что и объединение (union) языка C, когда необходима экономия памяти.

Из изложенного выше следует, что объявление класса должно иметь следующий формат:

<α> <β>;

Здесь α – одно из следующих зарезервированных слов :

- class,
- struct,
- union.

β – имя класса (идентификатор).

Примеры объявлений классов.

class stack;

struct complex;

Определение класса языка C++ имеет много общего с определением структуры языка C. Оно, так же как и определение структуры языка C, должно иметь заголовок и тело.

Отметим, что заголовок порожденного класса может иметь дополнительный элемент, называемый базовым списком. Базовый список указывает, от каких базовых классов порождается определяемый класс и задает вид порождения (public, protected или private).

Таким образом, заголовок класса имеет следующий формат (в квадратных скобках записан необязательный элемент):

<α> <β> [: <γ>]

Здесь α – одно из следующих зарезервированных слов:

- class,
- struct,
- union.

β – имя класса (идентификатор).

γ – базовый список (необязательный элемент).

Следом за заголовком необходимо записать тело класса, которое должно содержать последовательность объявлений всех компонентов, принадлежащих классу. Такие компоненты принято называть членами класса. Указанные объявления должны быть заключены в фигурные скобки. Тело класса, как и тело структуры должно заканчиваться точкой с запятой.

Члены класса

В любом классе может быть определено произвольное количество членов. Это могут быть данные, функции и определения типов Илии их синонимов.

Таким образом, тело класса в общем случае может содержать следующие объявления:

- переменных,
- либо прототип, либо определение каждой функции, инкапсулированной в классе,
- определения или объявления других классов.
- объявления typedef-синонимов типов.

Маркеры доступа

Для обеспечения инкапсуляции в языке C++ используются так называемые маркеры доступа (access label). В качестве маркеров доступа используются следующие зарезервированные слова языка C++:

- public (общедоступный),
- protected (защищенный),
- private (закрытый).

Указанные зарезервированные слова применяются в качестве меток к группе объявлений членов класса. Маркер доступа отделяется от объявления члена класса двоеточием. В теле класса обычно образуются разделы объявлений членов класса, наделенных одинаковыми правами доступа.

Все члены класса, находящиеся в public-разделе, доступны в любом клиентском коде, использующем возможности рассматриваемого класса. Члены класса, находящиеся в private-разделе доступны только функциям членам данного класса. Защищенный (protected) доступ имеет отношение к наследованию классов, и будет рассматриваться в соответствующем разделе курса

Если в теле класса нет ни одного маркера доступа, то права доступа определяются принципом умолчания. Для классов, использующих в заголовке зарезервированное слово class, в соответствии с этим принципом

все его члены будут закрытыми. Для классов же, использующих в заголовке зарезервированное слово struct, в соответствии с этим принципом все его члены будут открытыми. Этот же принцип умолчания действует для объявлений тех членов, которые записаны до первого маркера доступа.

Конструкторы

Здесь приводятся начальные сведения о конструкторах. В дальнейшем конструкторам будет посвящен специальный раздел.

Конструктор – специальная член-функция класса, которая обычно вызывается в момент создания объекта и служит для инициализации объекта. Особенность конструктора как члена-функции состоит в следующем:

- имя конструктора совпадает с именем класса
- конструктор не имеет типа возвращаемого значения.
- конструктор нельзя вызвать для существующего объекта, как обычную функцию-член класса.

Один из способов инициализации объекта с помощью конструктора (полей объекта) состоит в присваивании в теле конструктора полям объекта значений, передаваемых через параметры конструктора.

Пример. Конструкторы класса «Комплексные числа»

```
class Complex
{
    public:
      Complex()
      {
         re_ = 0;
         im_ = 0;
      }

      Complex(double re, double im)
      {
         re_ = re;
         im_ = im;
      }
    // Определения других членов-функций
    private:
      double re_, im_

};
```

В теле определения класса Complex , приведенного выше, имеются два раздела объявлений членов класса. Вначале находится общедоступный раздел (public-раздел или public-секция), который начинается после маркера и заканчивается перед маркером private. Этот раздел содержит определения двух конструкторов. Первым записано определение так называемого конструктора умолчания. Конструктор умолчания по определению – это конструктор, не принимающий параметров. Вторым записано определения конструктор, принимающий два параметра.

Закрытый раздел класса (public-раздел) содержит объявления двух членов-переменных.

Понятие об объекте

Объект – это экземпляр класса. Для создания объекта достаточно написать определение переменной классового типа. Примеры определений классовых переменных для класса Complex,

Организация кода при работе с классами

Возможны следующие способы организации программного кода, использующего классов:

- Определение класса полностью интегрировано в клиентский код.
- Определение класса может быть инкапсулировано в заголовочном файле.
- Определение помещено в модуль, состоящий из заголовочного файла и файла реализации.

По умолчанию все функции, определения которых содержатся в теле класса, считаются объявленными с зарезервированным словом inline.

С целью минимизации текста, находящегося в теле класса, рекомендуется помещать в нем прототипы функций, а их определения выносить за его пределы. Таким образом, определение класса может содержать дополнительный структурный компонент, которого нет в определении структуры языка С – вынесенные из тела класса определения его функций.

В простейших случаях, когда тело класса невелико и отсутствуют вынесенные определения функций, определение класса помещается в заголовочный файл. В заголовочном файле помещается и определения классов в тех случаях, когда все вынесенные определения функций реализуются в виде inline-функций. При отсутствии вынесенных из тела определений функций определение класса, так же как и определение структуры языка С должно быть записано в заголовочном файле.

Определения выносимых из тела класса функций, которые должны быть реализованы как обычные функции, должны быть помещены в файл реализации с расширением .cpp.

Таким образом, чаще всего определение класса чаще всего оформляется в виде модуля. В его интерфейсной части должна содержаться информация, представляющая интерес для клиента. Определения функций-членов класса к такой информации не относятся.

В интерфейсном файле должны находиться заголовок, тело класса и определения inline – функций, вынесенных из тела класса. В файле реализации модуля должны содержаться определения обычных (не-inline функций).

Имена всех функций, определения которых вынесены из тела функций, должны быть уточнены с помощью оператора ::.

Например.

// Файл MyClass.h

```
class MyClass
{
    public:
      void f();
      void g();

    // …
};
inline void MyClass :: f()
{
    // ..
}

// Файл MyClass.cpp
void MyClass :: g()
{
   // ..
}
```

Статические компоненты класса

Достаточно часто на практике используются так называемые статические компоненты класса. Статическими могут быть как члены-данные (поля), так и члены-функции. Особенностью статического поля является то обстоятельство, что они создаются в единственном экземпляре. Статическое поле принадлежит как бы всем объектам сразу. Статические поля представляют своеобразную разновидность глобальных переменных, область видимости которых ограничивается классом.

Компонент становится статическим в том случае, когда в его объявлении используется зарезервированное слово static.

Особенности при работе со статическими компонентами класса:

1. Статическое поле существует в единственном экземпляре и является общим для всех объектов.
2. Статическое поле инициализируется на внешнем уровне, т.е. вне функций и вне тела класса.
3. Для работы со статическими полями используются статические функции-члены класса.
4. Обращаться к статическим компонентам можно двумя способами. Во-первых, через объект (как это принято для нестатических компонентов). Во-вторых, используя имя класса (<имя класса> :: <имя компонента>).

Рассмотрим следующий пример. Необходимо создать класс Point, в котором предусмотрена возможность контроля количества созданных объектов.

```
//Файл Point.h
   class Point
   {
       public:
         Point(int x = 0, int y = 0);
         ~Point();
         static int HowManyObj();
       private:
         int x_, y_;
         static int count_;
   };

   // Файл Point.cpp
   int Point :: count = 0; // Инициализация статического поля

   Point :: Point(int x, int e) : x_(x), y_(y)
   {
       ++count_;
   }

   Point :: ~Point()
   {
       --count_;
   }

   int Point :: HowManyObj()
   {
      return count_;
   }

   // Клиентский код. Файл main.cpp
   int main()
   {
      Point pnt1(1, 2);
      Point* p1 = new Point(3, 4);
      cout << Point :: HowManyObj << endl;
      delete p1;
       cout << Point :: HowManyObj() << endl;
   }
```

Конструкторы. Детальное рассмотрение

В общем случае формат определения конструктора имеет следующий вид:

<α> (<β>) [<γ>]
{
  // Тело конструктора
}

Здесь:

- α- имя класса,
- () – аргументные скобки функции,

- β- список объявлений формальных параметров,
- [] – синтаксическое указание на то положение, что элементы, находящиеся внутри этих скобок, являются необязательными.
- строка инициализации.

Необязательная строка инициализации должна начинаться с двоеточия. Отдельные элементы этой строки отделяются запятыми. Отдельные элементы списка инициализации имеют следующий формат:

<δ>(<μ>)

Здесь:

- δ - имя инициализируемого элемента,
- μ - инициализирующее выражение.

Например

```
//Файл реализации класса complex.
// Здесь необходимо инициализировать поля re_ и im_

// Конструктор умолчания
complex :: complex() : re_(0), im_(0)
{
   // Пустое тело
}

// Конструктор копирования
complex :: complex(const complex& c) : re_(c.re_), im_(c.im._)
{
   // Пустое тело
}
```

С учетом изложенного выше, имеются два способа реализации конструктора. В одном из них используется строка инициализации, а в другом эта строка не используется, а выполняются присваивания в теле конструктора. Какой из этих способов является предпочтительным? Здесь следует учитывать следующие положения:

- Действия, выполняемые в строке инициализации, являются «настоящей инициализацией», а действия в теле конструктора – подменяют инициализацию присваиванием.
- Имеются типы данных, которые не допускают присваивания. К ним относятся: ссылочные типы и константные типы. Эти типы могут быть инициализированы только в строке инициализации.
- Оба способа инициализации имеют одинаковую эффективность для встроенных типов.
- Инициализацию пользовательских типов предпочтительнее выполнять в строке инициализации.

Функции getter и setter

Эти функции позволяют программисту работать с реализацией класса. Например:

```
double complex :: getRe()
{
   return re_;
}

void complex :: setRe(double re)
{
    re_ = re;
}
```

Функция getRe() позволяет работать с полем re_ в режиме чтения, а функция setRe() -позволяет работать с этим полем в режиме записи. Отношение программистов к рассматриваемым функциям не отличается однозначностью. Приведем цитату из книги Мартина Фаулера «Рефакторинг. Улучшение существующего кода»: « даже внутри этого класса следует всегда применять методы доступа (косвенный доступ к переменной). … преимущество косвенного способа к переменной состоит в том, что он позволяет переопределить в подклассе метод получения информации и обеспечивает большую гибкость в управлении данными»

Виды отношений между классами

В программе классы «действуют» не изолировано, а «вступают» в некоторые отношения друг с другом. Рассмотрим ряд таких отношений:

- зависимость,
- целое / часть, или включение, или агрегация,
- композиция - частный случай агрегации,
- наследование,
- обобщение / специализация (в языке C++ - это частный случай наследования)
- реализация

Отношение зависимость

Это один из наиболее слабых, но при этом и очень распространенный вид отношений между классами. Он предполагает, что один классов (зависимый класс) использует возможности, предоставляемые другим классом (независимым классом). Эта отношение на диаграммах классов в унифицированном языке моделирования отображается с помощью пунктирной линии, снабженной стрелкой. Эта линия имеет следующий вид --------->. Стрелка должна указывать на независимый класс. Например, такое отношение будет иметь между пользовательским классом complex и библиотечным классом istream в том случае, когда в классе complex для

реализации функции – члена void complex :: input() будет использоваться библиотечный потоковый класс istream.

Для зависимости часто характерен кратко временный характер зависимости. Действительно, это имеет место в случае классов complex и istream. Зависимость проявляется только во время выполнения функции input().

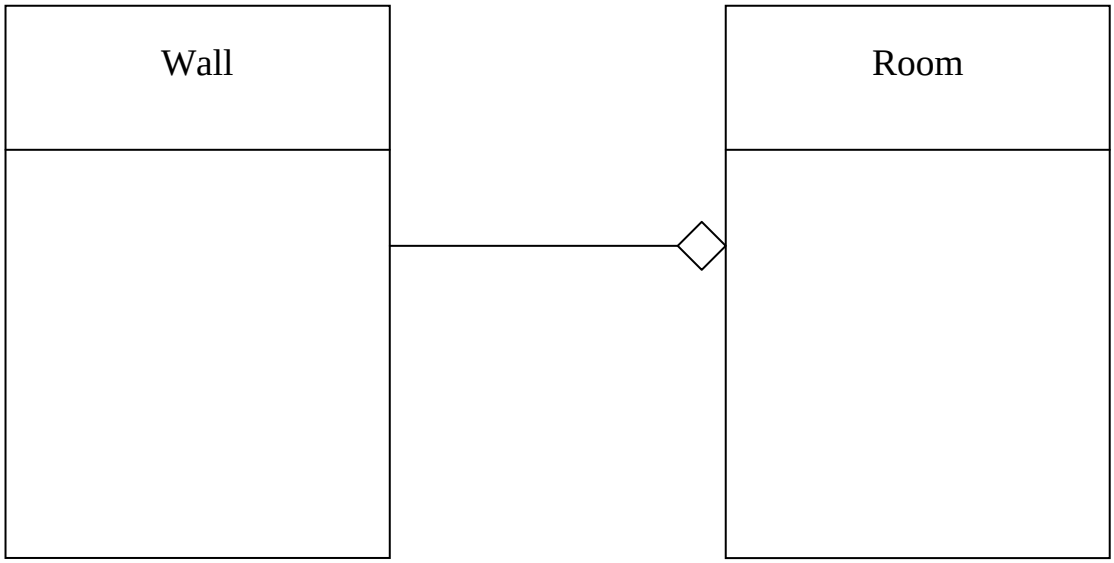
Отношение целое / часть

Очень часто встречающийся вид отношения при работе на языке C++. Рассмотрим конкретный пример. Имеются два класса:

- class room (комната),
- class wall (стена).

Очевидно, что объект класса wall содержится в объекте класса room. Здесь объект класса room – целое, а объект класс wall – часть.

Часто это отношение справедливо называют агрегацией. На диаграммах классов UML отражают с помощью сплошной линией, которая заканчивается пустым ромбом. Ромб должен быть расположен в конце той части линии, которая находиться со стороны класса – агрегата.



Некоторые авторы — это же отношение называют включением. Рассмотрим конкретный пример. Пусть имеется класс A. Нашей задачей является разработка нового класса B, в котором предполагается воспользоваться возможностями, предоставляемыми классом B. Одним из возможных способов решить эту задачу является использование рассматриваемого вида отношения между класса. Причем возможны три варианта организации включения. Для каждого из этих вариантов для класса B выберем свое уточненное имя. В первом из вариантов в разделе реализации нового класса объявляется поле типа A (class B_1), во втором – поле типа A*

(class B_2), а в третьем – поле типа A&(class B_3). Нашей задачей является рассмотрение особенностей организации конструкторов, деструктора и функции display(), выводящей на экран содержимое полей класса – агрегата.

```
class A
{
    public:
      A();
      A(int n);
      A(const A& rhs);
      void display();
    private:
     int n_;
};

class B_1
{
    // …
    private:
      A  a;
      int  m_;
};

class B_2
{
   // …
   private:
       A*  pa_;
        int  m_;
};

class B_3
{
    private:
       A&  r_;
        int   m_;
};
```

Ограничимся рассмотрением организации классов B_1 и B_2, в которых используются нашедшие наиболее широкое применение варианты использования отношения включения. Реализацию класса A приводить не будем. Начнем с разработки класса B_1.

Разработка класса B_1.

Напишем вначале интерфейсную часть класса B_1.

```
// Файл B_1.h
```

```
class B_1
{
      public:
         B_1();
         B_1(const A&, int m);
         B_1(const B& rhs);
         void display();
      private:
         A  a_;
         int  m_;
};
```

Замечание. В интерфейс нового класса включен один из возможных вариантов реализации конструктора с параметрами: B_1(const A&, int m);В порядке самостоятельной работы рассмотрите другую разновидность этого конструктора: B_1(int n, int m);

Перейдем к реализации класса B_1. Прежде всего, следует принять во внимание, что реализация должна находиться в отдельном файле с расширением .cpp. Выберем для него полное имя B_1.cpp. Будем учитывать много альтернативность возможных решений.

Начнем с рассмотрения возможных реализаций конструктора умолчания.

Конструктор умолчания.

Первый вариант реализации.

```
B_1 :: B_1() : m_(0)
{
}
```

Второй вариант реализации

```
B_1 :: B_1() : m_(0), a_()
{
}
```

Обсуждение. Это эквивалентные реализации. В первом варианте отсутствуют элементы инициализации для пользовательского поля a_. В этом случае компилятор должен вызвать конструктор умолчания класса, к которому относится пользовательское поле (в рассматриваемом примере это класс A). Если бы в нашем примере в классе A не был бы предусмотрен такой конструктор, то было бы выдано сообщение об ошибке.

Конструктор с параметрами

Здесь можно также рассмотреть две реализации. В первой из них инициализация выполняется в строке инициализации, а во второй инициализация заменяется присваиваниями в теле конструктора.

Первый вариант реализации

```
B_1 :: B_1(const A& a, int m) : a_(a), m_(m)
{
}
```

Второй вариант реализации

```
B_1 :: B_1(const A& a, int m)
{
   a_ = a;
   m_ = m;
}
```

Обсуждение. Первый вариант реализации обладает заметными преимуществами по сравнению со вторым вариантом. Дело в том, что инициализация в этом случае выполняется вызовом конструктора копирования класса A. Этот вызов будет иметь место при компиляции выражения a_(a), находящегося в строке инициализации. Сложнее будет обстоять дело при компиляции присваивания вида `a_ = a;` Такое присваивание возможно только в том случае, когда уже существует объект, стоящий слева от оператора присваивания. Для этой цели компилятор вызывает конструктор умолчания класса A.

Конструктор копирования.

Ограничимся одним вариантом реализации – использования строки инициализации.

```
B_1 :: B_1(const B_1& rhs) : a_(rhs.a_), m_(rhs.m_)
{
}
```

Рекомендация 1. В конструкторах предпочитайте инициализацию присваиванию.

Рекомендация 2. В строке инициализации записывайте отдельные элементы списка инициализации в том порядке, в котором появляются инициализируемые поля в объявлении класса. Заметим, что именно в этом порядке записаны элементы инициализации в рассмотренном примере. Дело в том, элементы строки инициализации выполняются в том порядке, в каком они объявляются, а в не том порядке, в котором они встречаются в строке инициализации.

Функция вывода на дисплей. display()

В теле этой функции следует вначале с помощью функции A :: display() вывести поля внутреннего объекта A :: a_, а затем собственного поля m_. Приведем реализацию этой функции.

```
void B_1 :: display()
{
    a_.disply();
    cout << "m = " << m_ << endl;
}
```

Деструктор

Деструктор в рассматриваемом классе можно не реализовывать. Это обусловлено тем обстоятельством, что отсутствуют операции, которые он должен выполнять.

Приведем теперь окончательную реализацию разработанного класса B_1.

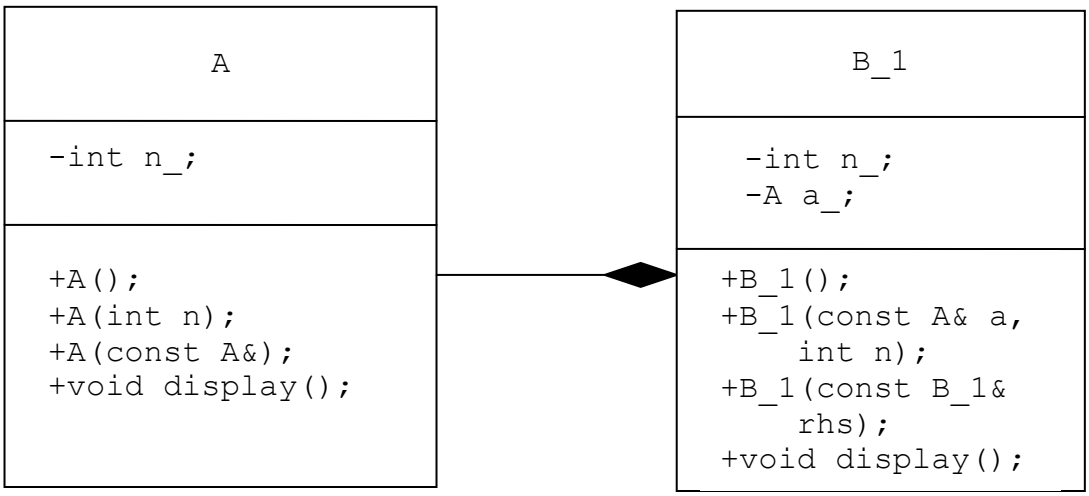
```
//Файл B_1.cpp
B_1 :: B_1() : m_(0)
{
}

B_1 :: B_1(const A& a, int m) : a_(a), m_(m)
{
}

B_1 :: B_1(const B& rhs) : a_(rhs.a_), m_(rhs.m_)
{
}

void B_1 :: display()
{
    a_.disply();
    cout << "m = " << m_ << endl;
}
```

Отметим, что класс A оказывается включенным в класс B1 в соответствии с принципом композиции. Приведем законченную диаграмму классов для рассматриваемого примера.



Разработка класса B_2.

Рассмотрим две версии этого класса. Для первой версии этого класса выберем имя B_2_1, а для второй – имя B_2_2. Прежде всего, следует отметить, что при разработке этого класса следует отдельно рассмотреть вопрос о необходимости деструктора. Интерфейс первой версии будет совпадать с интерфейсом класса B_1, а во второй , будет отличаться.

Начнем с разработки класса B_2_1. Напишем для него заголовочный файл. Обратите внимание на то, что добавлен деструктор.

```
// Файл B_2_1.h
class B_2_1
{
      public:
         B_2_1();
         B_2_1(const A& a, int m);
         B_2_1(const B_2_1& rhs);
         ~B_2_1();
         void display();
      private:
         A  *pa_;
         int  m_;
};
```

Перейдем к разработке реализации класса B_2_1. Отметим ряд важных исходных положений для реализации. При работе с указателями одним из центральных вопросов является вопрос о том, кто «владеет» объектом, на который установлен указатель (указатель pa_). Решение упрощается, если полагать, что права «владения» находятся у агрегатного объекта (объекта класса B_2_1). Будем предполагать, что в любом своем состоянии объект агрегатного класса содержит объект класса A.

Реализация класса B_2_1.

```
// Файл B2_1.cpp
B_2_1 :: B_2_1() : pa_(new A), m_(0)
{
}

B_2_1 :: B_2_1(const A& a, int m) : pa_(new A(a)), m_(m)
{
}

B_2_1 :: B_2_1(const B_2_1& rhs) : pa_(new A(*rhs.pa_)),
               m_(rhs.m_)
{
}

B_2_1 :: ~B_2_1()
{
   delete pa_;
}

void B_2_1 :: display()
{
     pa_->display();
     cout << "m = " << m << endl;
}
```

Обсуждение. В этой реализации класс целое (B_2_1) «владеет» объектом, на который установлен его указатель pa_. После уничтожения

объекта этого класса будет уничтожен и объект его компонента класса A. Можно считать, что классы A и B2_2 связаны отношением композиции.

Предлагаем диаграмму классов студенту нарисовать самостоятельно.

Дружественные функции и дружественные классы

Одним из основных принципов объектно-ориентированного программирования является инкапсуляция (сокрытие информации).

Однако в ряде случаев селективное (выборочное) предоставление прав доступа отдельным функциям или даже классам может быть полезным. Для этих целей и используется аппарат дружественных функций и классов.

Дружественное отношение устанавливается с помощью зарезервированного слова friend.

Типичные ситуациями, когда используются дружественные ситуации:

- Функция должна получить привилегированный доступ к закрытой части более чем одного класса.
- Перегрузка операторов.

Дружественной функцией по отношению к некоторому классу может быть объявлена как внешняя функция, так и функция-член другого класса. В последнем случае необходимо использовать оператор уточнения области действия «::».

Пример.

```
class One
{
    public:
      friend void f1(One& one);// friend функция
      int f2(Two& two);        // Функция-член класса One
      // ...
};

class Two
{
     public:
       friend int One :: f2(Two& two);// friend функция
       // ...
};

class Three
{
     public:
       friend class One;// friend класс
};
```

Обсуждение.

1. В классе One объявлена внешняя дружественная функция f1().
2. В классе Two объявлена дружественной функция-член f2() класса One

3. В классе Three дружественным объявлен класс One. Это означает, все функции класса One получают статус дружественности по отношению к классу Three.

Особенности применения дружественных функций и классов

Отметим ряд особенностей, которые следует учитывать при работе с дружественными функциями.

Дружественные функции не передается неявный указатель this. Поэтому при вызове дружественной функции нельзя использовать операторы:

.
->

Это обусловлено тем фактом, что дружественная функция не является компонентом класса.

Что же дает отношение дружественности? Ответ состоит в следующем.

Пусть имеется некоторая функция f(), которая объявлена дружественной в классе SomeClass. Это означает, что функция f() получает доступ к закрытым компонентам всех объектов класса SomeClass. Такие объекты могут передаваться в функцию f() через ее параметры или объявляться в теле рассматриваемой функции.

Место расположения прототипа функции, объявляемой в качестве дружественной в теле не имеет значения.

Примеры использования дружественных функций

```
// Файл One.h
class One
{
    private:
      int n;
      int m;
      friend void f(One& one);
      // ...
};

// Клиентский код.
void g()
{
   One one1, one2;
   one1.f(one2);  // недопустимо использовать оператор «.»
   // ...
}

void f(One& one)
{
   one.n = 0;   // Нормально
   m     = 10;  // Ошибка
   One one1;
   one1.n = 20; // Нормально
}
```

Дружественность не обладает свойством взаимности.

```
class One
{
    friend class Two;
    // ...
};
```

Класс Two является «другом» класса One, но класс One не становится «другом» класса Two.

Отношение дружественности не обладает свойством транзитивности. Друзья друзей класса One не являются друзьями класса One.

Пример.

```
class One
{
   private:
      int n;
      friend class Two;
      // ...
};

class Two
{
    friend class Three;
    // ...
};

class Three
{
    void f(One& one)
    {
        one.n = 20; // Ошибка
    }
};
```

ПЕРЕГРУЗКА ОПЕРАТРОВ

Перегрузка операторов делает систему встроенных типов языка C++ расширяемой. Это обусловлено тем обстоятельством, что программист может наделять создаваемые им пользовательские типы данных всеми возможностями, которые предусмотрены для встроенных типов. Это обеспечивает аппарат перегрузки операторов.

Перегрузку операторов можно рассматривать как частный случай перегрузки функций.

Рассмотрим ограничения, которые следует учитывать при перегрузке операторов:

- Перегружать операторы можно только для пользовательских типов; перегружать операторы для встроенных типов недопустимо.
- Нельзя при перегрузке изменять арность, приоритет и ассоциативность операторов. например, в выражении a * b +c приоритет операторов «*» и «+» не должен зависеть от того к

какой категории типов (встроенных или пользовательских) относятся переменные a, b и c/

- Нельзя вводить новые операторы, например оператор «^».для возведения в степень.
- Имеются операторы перегрузка, перегрузка которых не допускается. К ним относятся следующие операторы: операторы доступа «.» и «.*», оператор уточнения области видимости «::» и тернарный оператор «?:».

Остановимся на общих элементах синтаксиса перегрузки операторов.

Имя функции, используемой для перегрузки, имеет специальный формат: operator @. Здесь operator – зарезервированное слово языка C++, а @ – перегружаемый оператор.

Пример. Имя функции, используемой для перегрузки оператора присваивания: operator=.

Существуют две разновидности перегружающих функций:

- Функции – члены класса.
- Дружественные внешние функции.

Некоторые операторы следует перегружать с помощью функций – членов. К числу таких операторов относятся операторы, изменяющие объект:

- Оператор присваивания,
- Инкремент и декремент.

С другой стороны операторы ввода (>>) и вывода (<<) следует перегружать с помощью дружественных функций.

Рассмотрим вопрос о том, как компилятор вызывает функции, перегружающие операторы. Этот вопрос следует рассмотреть отдельно для перегрузки с помощью функций – членов класса и дружественных функций.

Обратимся к конкретному программному коду.

```
class Complex
{
    public:
      Complex operator+ (const Complex& rhs)
      //…
};

// Код клиента
int main()
{
   Complex c1(1, 3), c2(2, 4);
   Complex c3 = c1 + c2;
   // . . .
}
```

Строка кода: `Complex c3 = c1 + c2;` будет выполняться так, как будто она имеет следующий вид Complex `c3 = c1.operator=(c2).`

Предположим теперь, что перегрузка оператора + выполняется с помощью дружественной функции.

```
class Complex
{
```

```
   public:
      friend Complex operator+(const Complex& lhs,
                               const Complex& rhs);
```

Пусть клиентский код остался без изменения. Тогда рассматриваемая строка клиентского кода будет выполняться так, как будто она имеет следующий вид:

```
Complex c3 = operator+(const Complex& lhs,
                       const Complex& rhs));
```

Рассмотрим вопрос о количестве аргументов перегружающей функции. Это вопрос следует рассматривать отдельно для случая, когда перегрузка осуществляется с помощью функции – члена класса и внешней функции.

В первом случае, когда перегрузка выполняется с помощью функции – члена класса количество аргументов перегружающей функции на 1 меньше арности перегружающего оператора, а во втором случае – совпадает с арностью оператора.

Перегрузка оператора присваивания

Пусть имеется некоторый класс X. Функция – член класса, выполняющая перегрузку оператора присваивания для этого класса должна иметь следующий вид:

```
X& X :: operator=(const X& rhs);
```

Перегружающая функция должна возвращать ссылку на объект, находящийся слева от знака присваивания , должна принимать в качестве своего единственного параметра константную ссылку на правый операнд.

Это стандартный формат. Рассмотрим, какие будут последствия при отступлении это формата:

1. Функция возвращает void. Ответ: оказывается невозможным каскадное присваивание.
2. Функция возвращает значение типа X. Ответ: создается временный объект.
3. Функция принимает в качестве своего параметра значение типа X. Ответ: для хранения фактического параметра будет создаваться временный объект.

Реализация перегруженного оператора присваивания для класса Array

```
// Файл Array.h
class Array
{
    public:
      Array& operator=(const Complex& rhs);
      // ...
    private:
      double* pdata_;
      int     size_;
};
```

```
// Файл Array.cpp
Array& Array :: operator=(const Complex& rhs)
{
    if(this == &rhs)
       return *this;
    delete []pdata_;
    size_ = rhs.size_;
    pdata_ = new double[size_];

    for(int i = 0; i < size_; i++)
       pdata_[i] = rhs.pdata_;
    return *this;
}
```

Инструкция if в теле перегруженного оператора присваивания обрабатывает ситуацию, которая называется самоприсваиванием. Эта ситуация имеет место в том случае, когда слева и справа в операторе присваивания находится один и тот же объект. Эта ситуация должна обрабатываться специальным образом. Действительно, при удалении памяти с помощью инструкции delete []pdata_; имеет место потеря связи с областью памяти, выделенной для объекта, находящегося справа от оператора присваивания. С другой стороны, в рассматриваемой ситуации нет необходимости в обработке.

Перегрузка оператора индексирования

Рассмотрим этот вопрос применительно к классу Array.

```
double& Array :: operator[](int index)
{
    return pdata_[index];
}
```

Функция, осуществляющая перегрузку, должна возвращать ссылку. Это необходимо для того, чтобы вызов функции можно было писать слева от знака присваивания.

```
// Клиентский код
int main()
{
    double a[] = {1, 3, 5, 7, 9};
    Array ar(a, sizeof(ar) / sizeof(double));
    ar[1] = 21;
    // ...
}
```

Понятие о константной функции

Константной функцией называется член – функция класса, в определении которой между заголовком и ее телом записывается зарезервированное слово const.

Константный вариант перегруженного оператора индексирования

```
const double& Array :: operator[](int index)const
{
    return pdata_[index];
}


// Клиентский код
int main()
{
    double a[] = {1, 3, 5, 7, 9};
    Array ar1(a, sizeof(ar) / sizeof(double));
    ar1[1] = 21;// Ok, в режиме записи вызывается
                // неконстантная функция
    // ...
    double a2[] = {1, 3, 5, 7, 9};
    const Array ar2(a, sizeof(ar2) / sizeof(double));
    ar2[1] = 21; // Ошибка, попытка вызвать константную
                 // функцию в режиме записи
}
```

Перегрузка операторов << и >>

Эти операторы перегружаются с помощью дружественных функций. Это обусловлено тем обстоятельством, что при перегрузке бинарных операторов с помощью функции – члена класса левый операнд должен иметь тип, совпадающий с типом того класса, для которого выполняется перегрузка.

Пусть перегрузка выполняется для класса AnyClass. Тогда в выражении <операнд1> <оператор> <операнд2> должен иметь тип AnyClass

Предположим теперь, что в программе имеется определение класса, в котором необходимо переопределить операторы << и >>.

```
// Файл AnyClass.h
class AnyClass
{
     public:
        AnyClss();
        friend istream& operator>>(istream& is, AnyClass&
                                   rhs);
        friend ostream& operator>>(ostream& os, const
                                   AnyClass& rhs);
        // ...
     private:
       int n_;
       double x_;
};

..../// Файл реалтзации AnyClass.cpp
    istream& operator>>(istream& is, AnyClass& rhs)
    {
```

```
    cout << "n=";
    is >> rhs.n_;
    cout << "x=";
    is >> rhs.x_;
    return is;
}

ostream& operator<<(ostream& os, const AnyClass& rhs)
{
    os << "n=" << rhs.n_;
    os << "x=" << rhs.x_;
    return os;
}
```

Перегрузка унарных операторов ++ и –

Отметим основные особенности этих операторов:

1. Перегрузку следует выполнять с помощью функции – члена класса.
2. Для перегрузки постфиксного варианта этих операторов следует в перегружающей функции использовать фиктивный целочисленный параметр типа int.
3. Функция, перегружающая префиксный вариант этих операторов, должна возвращать ссылку, а постфиксный вариант должен возвращать значение классового типа.

В качестве примера рассмотрим перегрузку оператора ++ для класса Complex.

```
// Префиксный вариант
Complex& Complex :: operator++()
{
    ++re_;
    ++im_;
    return *this;
}
// Постфиксный вариант
Complex Complex :: operator++(int)
{
    Complex temp(*this);
    ++(*this);
    return temp;
}

// Клиентский код
int main()
{
     Complex c(1, 2);
     cout << ++c << endl;
     cout << c++ << endl;
     cout << c   << endl
     //
```

```
    }
// Вывод компьютера
(2, 3)
(2, 3)
(3, 4)
```

Если операторы ++ и -- используются только для получения побочного эффекта, то префиксным операторам следует отдать предпочтение перед постфиксными операторами. Это обусловлено тем обстоятельством, что при перегрузке постфиксных операторов создаются временные объекты, что негативно сказывается на временных характеристиках программы.

*Перегрузка бинарных операторов +, -, * и /*

Отметим две проблемы, которые возникают при организации перегрузки этих операторов.

1. В связи с тем, что перегружающая функция должна возвращать значение классового типа возникает проблема борьбы с временными объектами.
2. Организация вычисления смешанных вычислений.

```
// Неудачное решение
Complex Complex :: operator+(const Complex& rhs)
{
    Complex temp(rhs);
    temp.re_ += re_;
    temp.im_ += im_;
    return temp;
}
```

Этот пример может служить иллюстрацией обычного подхода к решению задач о возврате значения. Требуется выполнить следующие действия:

1. Объявить локальную переменную.
2. Записать действия, фиксирующие в этой переменной ожидаемый результат.
3. Инструкцией return вернуть результат, зафиксированный в локальной переменной.

```
// Рекомендуемое решение
Complex Complex :: operator+(const Complex& rhs)
{
    return Complex(re_ + rhs.re_, im_ + rhs.im_);
}
```

Преимущество состоит в том, что нет создания локального объекта.

```
// Первый вариант клиентского кода.
Complex c1(1, 2), c2(3, 4), c3;
// ...
c3 = c1 + c2;

// Второй вариант клиентского кода
Complex c1(1, 2), c2(3, 4), c3;
```

```
// ...
c3 = c1 + c2;
```

Для последнего варианта клиентского кода при использовании рекомендуемого решения относительно перегружающей функции компилятор может выполнить оптимизацию возвращаемого значения. Временные объекты при этой оптимизации создаваться не будут. Результат сложения непосредственно должен сформироваться в переменной c3.

Вычисление смешанных выражений

В ряде случаев важно иметь возможность записи программного кода следующего вида

```
Complex c1(1, 2);
Complex c2 = c1 + 4;
Complex c3 = 5 + c1;
```

Возможны два подхода к решению проблемы вычисления смешанных выражений:

1. Использование неявных преобразования типов.
2. Использование нескольких вариантов перегружающих операторов + . Речь идет о перегрузке перегружаемых операторов optrftor+.

```
// Первый вариант
class Complex
{
    public:
       Complex(double re = 0, double im = 0);
       friend Complex operator+(const Complex& lhs,
                                const Complex& rhs);
       // ...
};

// Второй вариант
class Complex
{
    public:
      explicit Complex(double re, double im = 0);
      friend Complex operator+(const Complex& lhs,
                               const Complex& rhs);
      friend Complex operator+(double lhs,
                               const Complex& rhs);
      friend Complex operator+(const Complex& lhs,
                               double rhs);
      // ...

};
```

Замечание. В черновике последний прототип отсутствует !!!

Зарезервированное слово explicit запрещает неявное преобразование типа. Возможности у второго варианта по работе со смешанными выражениями такие, как и у первого варианта, но временные объекты не появляются.

Перегруженный оператор ->

В качестве примера рассмотрим применение перегрузки этого оператора для реализации отношения делегирования.

Постановка задачи. Имеется класс A/

```
class A
{
    public:
       void fa();
       void fab();
       void vab();
       // ...
};
```

Требуется реализовать класс B, в котором была бы возможность использовать функции fab() и vab() из класса A. Для этой цели можно воспользоваться перегрузкой оператора ->.

```
class B
{
   public:
     B();
     void fb();
     A* operator->()
     {
         return delegate_;
     }

      private:
         A* delegate_;
};
```

Особенности. Бинарный оператор -> перегружен как унарный.

```
// Код клиента
int main()
{
    B b;
    b.fb();   // Обращение к собственной функции
    b->fab(); // Делегирование
    // ...
}
```

Недостатки:

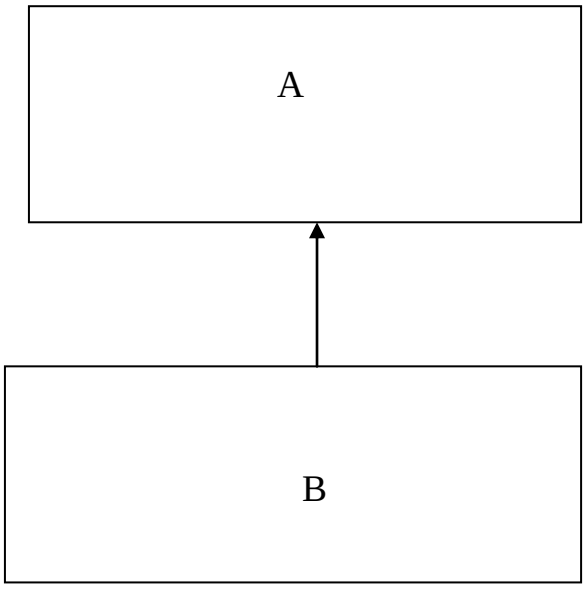
1. Различный синтаксис обращения к собственным функциям и функциям делегатам.
2. Возможна избыточность интерфейса.

НАСЛЕДОВАНИЕ

Наследование – специальный языковой механизм, предназначенный для выражения особых отношений между классами. Наследование обычно используется в следующих двух случаях:

- Для построения иерархических систем.
- Создания новых классов, сходных с уже существующими классами.

Остановимся на формализме, связанном с наследованием. Пусть имеется некоторый класс A, от которого наследуется или порождается класс B. Графически это отношение отображается следующим образом.

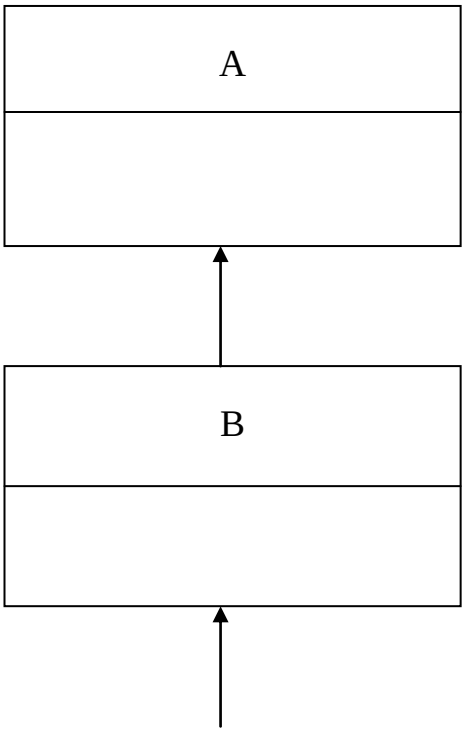


Базовый класс

Порожденный класс

В настоящее время общепринятым является направлять стрелку от порожденного класса к базовому.

Различают прямую и косвенную базу. Пусть имеются три класса (A, B и C), связанных отношением наследования



Класс C имеет два базовых класса (A и B).Причем класс B является прямой базой, а класс A – косвенной базой.

Различают одиночное и множественное наследование. При одиночном наследовании все классы имеют одну прямую базу. При множественном наследовании некоторые классы имеют несколько прямых баз.

В языке C++ имеются три разновидности наследования:

- public – наследование,
- protected – наследование.
- private – наследование.

Вид наследования влияет на степень доступности элементов базового класса для функций – членов порожденного класса.

Прямые базовые классы указываются в заголовке порожденного класса в специальном элементе, который называют базовым списком. В базовом списке указывается также и вид наследования. Например:

```
class C : public A, private B
{
   // ...
};
```

Базовый список отделяется от остальной части заголовка класса двоеточием. В рассматриваемом примере базовый список порожденного класса C состоит из двух элементов (класс C имеет две прямые базы). Из первого элемента следует, что класс C порождается путем public – наследования от класса A, а из второго – путем private – наследования от класса B. Вид наследования допускается не указывать. В этом случае он (вид наследования) может быть определен в соответствии с принципом умолчания. При этом для классов, в заголовке которых используется зарезервированное слово class, в соответствии с принципом умолчания предполагается private – наследование, а для классов, использующих в заголовке слово struct – предполагается public – наследование. Например:

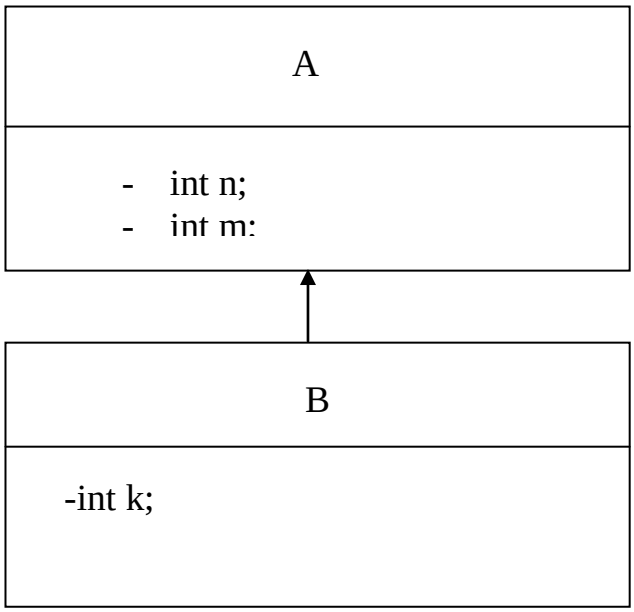
```
class A : B
{
   // ...
};

struct C : D
{
   // ...
};
```

В этом примере класс A порождается от класса B с помощью private – наследования, а класс C порождается от класса D –с помощью public – наследования.

Структура объекта порожденного класса

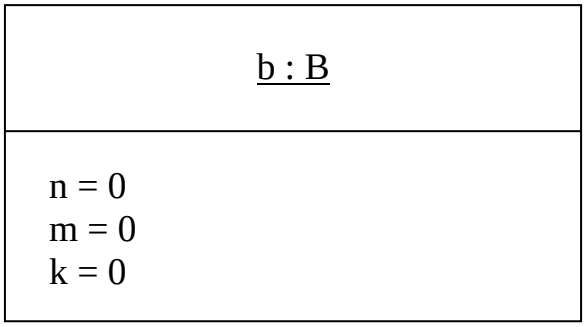
Пусть имеются два класса A и B, связанные отношением наследования.



Предположим, что в клиентском коде имеется следующее определение переменной b:

B b;

Изобразим диаграмму объекта b.



Объект b ,будет содержать три поля с именами n, m и k. Два из них (n и m) рассматриваемый объект наследует от базового класса, а поле k является его собственным полем. На диаграмме показаны значения, которыми были предположительно инициализированы эти поля.

Доступ к элементам базового класса

Доступность элементов базового класса в порожденном классе зависит от вида наследования.

Вид наследования	Права доступа в базовом классе	Права доступа в порожденном классе
public	private	private
	protected	protected
	public	public
Protected	private	private
	protected	protected
	public	
Private	private	private
	protected	
	Public	

Обсуждение.

В языке C++ для обеспечения сокрытия информации используется принцип управления доступом, а не управление видимостью. Для уяснения различия между этими двумя подходами, рассмотрим следующий пример.

```
// Файл Classes.h
class A
{
    private:
      int n;
};

class B : public A
{
    public:
      void foo();
};

// Файл B.cpp
int n;
void B :: foo()
{
   int n = 5;
   // ...
}
```

При компиляции рассматриваемого программного кода будет выведено сообщение об ошибке Дело в том, что в теле функции foo() видна закрытая переменная – член базового класса A. Эта переменная скрывает глобальную переменную n, объявленную в файле B.cpp. Если бы сокрытие информации осуществлялось за счет управления видимостью, то сообщение об ошибке появляться не должно.

Итак, очень важный вывод состоит в том, что все закрытые элементы базового класса в порожденном классе видны, но недоступны. Заметим, что в других объектно-ориентированных языках могут быть приняты свои способы сокрытия информации. Например, в Object Pascal Delphi это осуществляется за счет управления видимостью.

Требует осуждения следующий вопрос. Связи с чем функциям порожденного класса не предоставлен доступ к реализации базового класса. На это есть определенные причины. Дело в том, что в противном случае клиент также может сравнительно легко получить такой же доступ. Для этого ему достаточно породить класс.

В разделе protected следует размещать те элементы базового класса, которые должны быть доступны порожденному классу, но не должны быть доступны его клиентам. Не рекомендуется в этот раздел базового класса включать элементы его реализации.

При public – наследовании открытый интерфейс базового класса входит составной частью в интерфейс порожденного класса. Можно говорить о том, что при public – наследовании наследуется интерфейс базового класса.

При использовании protected – наследования и public – наследования интерфейс базового класса клиентам базового класса не предоставляется. В этом случае можно говорить о наследовании реализации.

Имеется ряд элементов базового класса, которые не наследуются. К их числу относятся:

- конструкторы.
- Деструктор.
- Перегруженный оператор присваивания.

Основным видом наследования в языке C++ является public – наследование. Именно при этом виде наследования реализуются все фундаме

Конструкторы порожденного класса

Сформулируем ряд принципиальных положений, которые следует учитывать при разработке конструкторов порожденного класса.

Пусть B – базовый класс, объявление которого имеет следующий вид.

```
class B
{
   public:
     B();
     B(int b);
     B(const B& b);
     // ...
   private:
     int b_;
};
```

Предположим теперь, что от класса B с помощью public – наследования порождается класс D.

```
class D : public B
{
```

```
    public:
      D();
      D(int b, int d);
      D(const D& r);
      // ...
    private;
      int d_;
};
```

Наша задача состоит в том, чтобы написать определения каждого из трех конструкторов класса D.

Конструктор порожденного класса (класс D) должен инициализировать собственные поля (в нашем случае поле d_), а для инициализации базовых элементов следует вызывать конструктор прямой базы. Не рекомендуется пытаться выполнять инициализацию базовых полей вручную.

Вызов конструктора базового класса должен быть размещен в строке инициализации разрабатываемого конструктора. Недопустимо для этой цели вызывать конструктор базового класса в теле разрабатываемого конструктора.

Если в строке инициализации конструктора порожденного класса отсутствует вызов конструктора базового класса, то будет вызван конструктор умолчания базового класса.

Отдельно следует остановиться на инициализации базовой части для конструктора копирования. Здесь можно воспользоваться тем обстоятельством, что конструктору копирования базового класса в соответствии с принципом подстановке Лисков можно передавать объект порожденного класса. С учетом высказанных соображений можно записать следующие определения конструкторов класса D.

```
D :: D() : B(), d_(0) // Вызов конструктора умолчания
                      // базового B()можно не писать
{
}

D :: D(int b, int d) : B(b), d_(d)
{
}

D :: D(const D& r) : B(r), d_(r.d_)
{
}
```

Порядок создания объекта порожденного класса

1. Выделяется память для всего объекта (для собственных полей и базовой части объекта).
2. Вызывается конструктор базового класса для инициализации базовой части объекта.
3. Инициализируются элементы производного класса в соответствии с компонентами списка конструктора порожденного класса. Порядок выполнения элементов списка инициализации

определяется порядком, в котором они встречаются они объявляются .

4. Выполняется тело конструктора порожденного класса.

Перегруженный оператор присваивания порожденного класса

Пусть имеется базовый класс B и порожденный от него с помощью public – наследования класс D. Предположим, что в классе B имеется перегруженный оператор присваивания. Реализация перегруженного оператора присваивания упрощается, если предусмотреть в его теле вызов перегруженного оператора присваивания базового класса. Перегруженный оператор присваивания для порожденного класса D будет иметь следующую структуру.

```
D& D :: operator=(const D& rhs)
{
      if(this == &rhs)   // Защита от самоприсваивания
         return *this;
      B :: operator=(rhs); // Вызов оператора присваивания
                           // базового класса
      // Копирование собственных полей класса D
      return *this;
}
```

Вызов виртуальной функции из тела невиртуальной функции

Речь идет о наделении невиртуальной функции виртуальными возможностями.

Рассмотрим конкретный пример.

```
// Файл Classes.h
class B
{
   public:
     B();
     void f()
     {
         g1();
         g2();
     }
     virtual void g1();
     virtual void g2();
     // ...
};

class D : public B
{
    public:
      D();
      virtual void g1();
      virtual void g2();
      // ...
};
```

```
// Клиентский код. Файл main.cpp
int main()
{
    D d;
    d.f();
    // ...
}
```

При выполнении функции f() будут вызваны вызовы следующих функций: D :: g1() и g2(). Таким образом, обычную невиртуальную функцию можно заставить работать виртуально. Более того, можно это же проделать с обычной внешней функцией, которую принципиально нельзя объявить виртуальной.

Виртуализация функций не-членов класса

Такая функция не может быть объявлена виртуальной, но ее можно наделить свойствами виртуальности.

Для решения этой задачи можно использовать следующий подход. Вначале разрабатывается виртуальная функция-член класса, а затем определяется внешняя функция, которая вызывает эту функцию-член класса.

Пример. Наделение свойствами виртуальности перегруженного оператора << .

Пусть имеются три класса Two (базовый), Three (порожденный с помощью public – наследования от Two) и Four (порожденный с помощью public – наследования от Three). Предположим далее, что в каждом из этих классов определена виртуальная функция print() – член класса, выводящая на экран дисплея значения полей своего класса. Функция print() должна быть объявлена следующим образом.

```
virtual ostream& print(ostream& os)const;
```

Затем необходимо написать определение следующей внешней функции

```
ostream& operator<<(ostream& os, const Two& rhs)
{
     return rhs.print();
}
```

Идиома невиртуального интерфейса (NVI)

Один из переходов от использования открытых виртуальных функций к закрытым виртуальным функциям состоит в следующем. Выполним структурирование виртуальных функций. Во-первых, выделим в них постоянную и переменную части. Речь идет о том, чтобы отделить части, которые будут повторяться один к одному в порожденных классах от индивидуальных в каждом из порожденных классах.

Процесс структуризации можно распространить и на переменную часть виртуальных функций.

Пусть общий интерфейс, задаваемый базовым классом, должен содержать операцию, для которой выберем имя Action_1.

Предположим, что для любого порожденного класса вначале необходимо выполнять некоторые стандартные действия. Например,

требуется выделять определенные ресурсы. Выберем для этих действий имя StartAction_1/

Естественно, что в завершающей стадии операции Action_1 могут потребоваться некоторые стандартные действия. Например, по возвращению занятых ресурсов. Выберем для этих действий имя EndAction_1.

Обозначим через DoAction_1 переменную часть операции Action_1. Эта часть зависит от порожденного класса. Именно это действие оформим в виде виртуальной функции.

```
// Файл B.h
class B
{
    public:
      B();
      void Action_1();
    private:
      virtual void DoAction_1();
      void StartAction_1();
      void EndAction_1();
      // ...
}

// Файл B.cpp
void B :: Action_1()
{
    StartAction_1();
    DoAction_1();
    EndAction_1();
}

// Файл D.h
class D :public B
{
    public:
      D();
      // ...
    private:
      virtual void DoAction_1();
      // ...
};
```

Преимущества

1. Уменьшается общий объем исходного кода.
2. Упрощается отладка и модификация программного кода.

Реализация механизма виртуальных функций

В основе поддержки виртуальных функций лежит позднее связывание. Речь идет о моменте времени, когда устанавливается связь между вызовом функции и ее программным кодом.

Различают два способа связывания:

- Статическое (раннее).
- Динамическое (позднее).

Для обычных (невиртуальных) функций используется раннее связывание. Это связывание реализуется на этапе компиляции.

Преимущество раннего связывания состоит в возможности выполнить на этапе компиляции контроль типов.

Раннее связывание для виртуальных функций не подходит. Это связано с тем, что тип объекта, с которым работают указатель или ссылка к моменту времени компиляции еще неизвестен.

Позднее связывание, которое используется при работе с виртуальными функциями, осуществляется во время выполнения программы.

Реализация виртуальных функций, которая характерна для языка C++, основана на использовании таблиц виртуальных функций (vtable). Таблицы виртуальных функций организуются на основе массива указателей.

Таблицы виртуальных функций строятся для каждого класс, который работает с виртуальными функциями.

Объем таблицы виртуальных функций vtable определяется общим количеством виртуальных функций, а не количеством переопределенных функций. Например, в базовом классе имеется 100 виртуальных функций, в порожденном от него классе переопределяется только одна из них. Таблица виртуальных функций порожденного класса будет содержать те же 100 строк, что и таблица виртуальных функций его базового класса.

Средством обращения к таблице виртуальных функций является указатель виртуальных методов (vptr). В каждый объект класса, работающего с виртуальными функциями, помещается скрытый указатель vptr.

Указатель vptr инициализируется конструктором во время создания объекта.

Накладные расходы при работе с виртуальными функциями

Имеются два вида таких расходов:

- Временные
- Расходы на память.

Временные накладные расходы сравнительно невелики. Они обусловлены косвенным вызовом методов через указатель vptr.

С другой стороны дополнительные расходы на память могут оказаться значительными. Они складываются из двух частей:

- Общие.
- Индивидуальные.

Общие расходы – это расходы на хранении таблиц виртуальных методов.vtable. Эти таблицы устроены таким образом, чтобы устранить поиск. При этом увеличивается их объем.

Индивидуальные расходы на память связаны с хранением в объекте указателя vptr. Они велики в том случае, когда создается большое количество объектов.

Чисто виртуальные функции. Абстрактные базовые классы

Иногда требуется , чтобы базовый класс определял только интерфейс лля порожденных классов. В этом случае следует запретить в клиентском коде создание объектов базового класса. С этой целью базовый класс объявляется абстрактным. Для этого в базовом классе необходимо объявить хотя бы одну чисто виртуальную функцию.

Объявление чисто виртуальной функции должно начинаться с зарезервированного слова virtual и заканчиваться символами «= 0;». Такую виртуальную функцию иногда ассоциируют с нулевым указателем.

Пример.

```
class Shape
{
    public:
       virtual void Draw();
    // ...
};
```

Если попытаться создать объект для класса, содержащего хотя бы одну чисто виртуальную функцию, то компилятор выдаст сообщение об ошибке.

Пример.

```
Shape shape;  // Ошибка
```

Объект абстрактного класса не может быть параметром функции.

Например.

```
void foo(Shape sh);  // Ошибка
```

Абстрактный класс нельзя использовать в качестве значения, возвращаемого функцией.

Например.

```
Shape foo();  // Ошибка
```

Допускается использовать указатель и ссылки для абстрактных классов.

Например.

```
Shape& f(Shape& sh); // Нормально
Shape* ps;           // Нормально
```

При наследовании от абстрактного класса все чисто виртуальные функции должны быть переопределены, иначе производный класс будет абстрактным.

Абстрактные классы вида Shape воплощают отвлеченные понятия.

Для абстрактных классов таблица виртуальных методов оказывается неполной. Строки этой таблицы, соответствующие чисто виртуальным функциям, оказываются незаполненными.

Отметим, что чисто виртуальные функции могут иметь реализацию.

Виртуальные деструкторы

Виртуальный деструктор оказывается необходимым в тех случаях, когда необходимо предусмотреть полиморфную работу с объектами через указатель или ссылку на базовый класс.

Пусть имеются два класса B и D, связанные public – наследованием.

```
class B
{
```

```
    public:
      B();
     ~B();
};

class D : public B
{
    public:
      D();
     ~D();
}
```

Предположим теперь, что имеется следующий клиентский программный код

```
int main()
{
   B* pb = new B;
   // ...
   delete pb;
   // ...
}
```

Рассмотрим два случая. Вначале предположим, что деструктор в классе B является виртуальным. В этом случае оператор delete вызовет деструктор класса D, который после завершения своей работы вызовет деструктор базового класса. Это обусловлено тем обстоятельством, что при работе с виртуальными функциями определяющим является динамический тип указателя.

Перейдем к рассмотрению второго случая, когда деструктор базового класса является невиртуальным. В этом случае определяющим является статический тип указателя. В результате будет вызван только деструктор базового класса.

Автономные и базовые классы

Автономный класс – это одиночный класс, от которого не предполагается порождать другие классы. Это класс, который не предполагается использовать в качестве базового класса. Как отличить такой класс от класса, от которого предполагается порождать другие классы.

Рекомендация.

Признаком того, что рассматриваемый класс может служить в качестве базового класса является наличие в нем виртуального деструктора.

Чисто виртуальный деструктор

Деструктор может быть сделан чисто виртуальным. Особенности использования чисто виртуальных деструкторов состоят в следующем:

1. Чисто виртуальные деструкторы должны иметь реализацию.
2. Чисто виртуальные деструкторы не обязательно переопределять в порожденном классе. Компилятор сгенерирует недостающий деструктор.

3. Объявление деструктора чисто виртуальным позволяет сделать класс абстрактным.

Принцип подстановки Барбары Лисков (LSP)

Этот принцип используется в качестве критерия, определяющего возможность использования public – наследования. В соответствии с этим принципом public – наследование следует использовать в тех случаях, когда можно вместо в любой программе,

Принцип открыто – закрыто

Множественное наследование

Множественное наследование используется в тех случаях, когда возникает необходимость воспользоваться возможностями, которые могут предоставить несколько классов. Подобная ситуация характерна при использовании нескольких библиотек.

При применении множественного наследования возникает ряд проблем.

Конфликт имен.

Это одна из проблем, которая возникает при использовании множественного наследования. Рассмотрим пример.

```
class B1
{
     public:
       void f();
       // ...
};

class B2
{
     public:
       void f();
       // ...
};

class D : public B1, public B2
{
      public:
      void foo()
      {
         f();
      }
      // ...
};
```

При компиляции функции-члена foo() из класса D появляется сообщение об ошибке. Это обусловлено неоднозначностью, которую не может разрешить компилятор. Для устранения этой неоднозначности программист

может перед именем функции f() написать оператор уточнения области видимости. Исправленный вариант определения класса D приведен ниже.

```
// Исправленный вариант
class D
{
   public:
     void foo()
     {
        B1 :: f():
     }
};
```

Дублирование подобъектов

Рассмотрим следующую систему классов.

```
class Top
{
   public:
     // ...
   private:
     int top_;
};

class Left : public Top
{
   public:
     // ...
   private:
     int left_;
};

class Right : public Top
{
   public:
     // ...
   private:
     int right_;
};

class Bottom : public Left, public Right
{ kj
   public:
     // ...
   private:
     int bottom_;
};

// Код клиента
int main()
{
    Bottom bm;
    // ...
}
```

Объект bm класса Bottom будет содержать пять полей. В их число будут входить:

1. bottom_ – собственное поле,
2. left_ – поле, наследуемое от класса Left,
3. right_ – поле, наследуемое от класса Right,
4. top_ – поле, наследуемое от класса Top по ветви Left – Top,
5. top_ – поле, наследуемое от класса Top по ветви Right – Top,

Для устранения дублирования базового подобъекта применяют так называемое виртуальное наследование. Для реализации виртуального наследования в рассматриваемом примере необходимо в базовые списки классов Left и Right необходимо включить зарезервированное слово virtual.

```
class Top
{
    // ...
};
class Left : virtual public Top
{
     // ...
};

class Right : virtual public Top
{
     // ...
};

class Bottom : public Left, public Right
{
     // ...
};

// Код клиента
int main()
{
    Bottom bm2;
    // ...
}
```

Теперь объект (bm2) класса Bottom будет содержать четыре поля. В их число будут входить:

1. bottom_ – собственное поле,
2. left_ – поле, наследуемое от класса Left,
3. right_ – поле, наследуемое от класса Right,
4. top_ – поле, наследуемое от класса Top.

При использовании виртуального наследования возникает дополнительная проблема, связанная с построением конструкторов.

Конструкторы при виртуальном наследовании

Для вызова конструкторов базовых классов обычно используется список инициализации. При этом следует вызывать только конструктор прямой базы. Параметры, необходимые для инициализации косвенных баз, при этом

передаются через параметры конструктора прямой базы. При использовании виртуального наследования этот механизм компилятором отключается и необходимо вызывать непосредственно конструктор базового класса.

Реализация конструктора с параметрами для класса Bottom.

```
Bottom :: Bottom(int top, int left, int right, int bottom) :
          Left(one, left), Right(one, right), Top(one),
          bottom_(bottom)
{
}
```

Работа с данными при виртуальном наследовании

Предположим, что необходимо реализовать для каждого из классов, входящих в рассматриваемую в настоящем разделе систему классов, функцию Print(). Решение этой задачи показано ниже.

```
class  Top
{
    public:
      Top();
      Top(int top);
      void Print()
      {
         Data();
      }
      virtual ~Top();

    protected:
      void Data()
      {
         cout << "one=" << one << endl;
      }

    private:
      int one_;
};

class Left : virtual public Top
{
    public:
      Left();
      Left(int top, int left): Top(top), Left(left)
      {
      }
      void Print()
      {
          Top :: Print();
          Data();
      }
      virtual ~Left();

    protected:
       void Data()
```

```
        {
           cout << "left=" << left_ << endl;
        }

    private:
        int left_;
};

class Right : virtual public Top
{
   <самостоятельно написать реализацию>
};

class Bottom : public Left, public Right
{
    public:
      Bottom();
      Bottom(int top, int left, int right, int bottom) :
            Top(top), Left(one, left), Right(one, right),
            bottom_(bottom)
      {
      }
      void print()
      {
         One  ::  Data();
         Left ::  Data();
         Right :: Data();
         Data();
      }

    private:
      Data()
      {
         cout << "bottom=" << bottom_ << endl;
      }
      int bottom_;
};
```

ИНТЕЛЛЕКТУАЛЬНЫЕ УКАЗАТЕЛИ

Интеллектуальные указатели – это объекты класса, которые действуют подобно обычным указателям, но обладают рядом дополнительных возможностей.

В стандартную библиотеку языка C++ включен ряд реализаций интеллектуальных указателей. Интеллектуальные указатели используются для устранения утечек памяти и обеспечения безопасности при работе с исключениями.

Разновидности интеллектуальных указатели

Интеллектуальные указатели, предусмотренные стандартом C++ :

1. unique_ptr.
2. shared_ptr.

3. week_ptr.

Используйте эти смарт-указатели как первый вариант для инкапсуляции указатели на обычные старые объекты C++ (POCO).

- **unique_ptr**
  Позволяет точно один владелец базового указателя. Использовать в качестве элемента по умолчанию POCO не известно то, что требуется, **shared_ptr**. Могут быть перемещены в новый владелец, но не копируются или совместно. Заменяет **auto_ptr**, который является устаревшим. Сравнение с **boost::scoped_ptr**. **unique_ptr**является небольшим и эффективным; размер составляет один указатель и поддерживает rvalue ссылки для быстрой вставки и извлечения из коллекции библиотеки STL. Файл заголовка: <memory>. Дополнительные сведения см. в разделах Практическое руководство. Создание и использование экземпляров unique_ptr и unique_ptr Class.
- **shared_ptr**
  Со счетчиком ссылок смарт-указатель. Следует использовать, когда нужно назначить один исходный указатель нескольких владельцев, например, когда возвращать копию указатель из контейнера, но сохранить исходное. Исходный указатель не удаляется до всех **shared_ptr** владельцев выходе за пределы области действия или в противном случае заданные настройки владельца. Имеет размер двух указателей; один объект и блок общий элемент управления, который содержит счетчик ссылок. Файл заголовка: <memory>. Дополнительные сведения см. в разделах Практическое руководство. Создание и использование экземпляров shared_ptr и shared_ptr Class.
- **weak_ptr**
  Особой смарт-указатель для использования в сочетании с **shared_ptr**. А **weak_ptr** предоставляет доступ к объекту, который владеет одним или несколькими **shared_ptr** экземпляров, но не участвует в подсчет ссылок. Используется, когда необходимо наблюдать за объектом, но не требуют его оставаться в активном состоянии. Требуется в некоторых случаях прерывание циклических ссылок между **shared_ptr** экземпляры. Файл заголовка: <memory>. Дополнительные сведения см. в разделах Практическое руководство. Создание и использование экземпляров weak_ptr и weak_ptr Class.

ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Под исключением понимается ситуация, не позволяющая программе нормально работать. Например, программа может пытаться открыть недоступный ей файл. Рассмотрим вначале недостатки традиционных методов, связанные с анализом программой кодов ошибок, которые возвращаются функцией. К их числу можно отнести следующее:

1. Отсутствие структурированности. Код обработки ошибок не отделен от кода нормальной работы программы.
2. Возможность игнорирования кода ошибок.

3. Сложность обработки ошибок, возникающие при разработке библиотек. Разработчик библиотеки часто не в курсе как обрабатывать ошибки.

. Аппарат исключительных ситуаций предназначен для преодоления недостатков, присущих классическим методам обработки ошибок. Его преимущества состоят в следующем:

- Код нормальной работы программы и код обработки ошибок структурно отделены.
- Выброс исключения игнорировать нельзя. Отсутствие обработчика приводит к завершению работы программы.

Завершение или продолжение

В теории обработки исключений имеются две модели:

- обработка с завершением работы программы.
- обработка с продолжением работы программы.

В модели с завершением выполнения программы предполагается, что ошибка настолько серьезна, что возобновлять ее выполнение с точки возникновения ошибки нельзя. Необходимо позаботиться о корректном завершении программы. Например, сохранить не записанные на диск данные. Эта модель и принята в языке C++.

Распределение обязанностей между разработчиком и клиентом

Разработчик должен обеспечить:

- выброс исключения при возникновении исключительной ситуации в разработанной им функции.
- уведомить клиента о том, какие исключения может «выбросить» его функция.

Клиент должен предусмотреть обработчики исключений, которые могут выбросить используемые им функции.

Генерация исключений

Генерация исключения выполняется с помощью инструкции throw. Имеются два формата этой инструкции.

Первый формат:

```
throw<α>, где
throw – зарезервированное слово;
α     - выражение.
```

Второй формат:

```
throw
```

Основным является первый формат. Второй формат применяется для так называемого повторного выброса исключений.

Назначение первого формата инструкции throw состоит в создании объекта исключения. Основная информация для обработчика исключительной ситуации состоит в типе выброшенного объекта.

Объект исключения может иметь любой тип:

- встроенный.
- пользовательский.

Предпочтительней выбрасывать объекты пользовательского типа. Стандартным является использование анонимного объекта пользовательского типа. Предпочтительнее использовать либо один из классов стандартной библиотеки C++, предназначенных для работы с исключениями, либо порождать от этих классов собственный класс. Не рекомендуется использовать в инструкции **throw** строковые литералы. Это усложняет реализацию обработчика.

Примеры хороших и плохих решений.

Хорошие решения:

- `throw std :: invalid_argumtnt("Invalid _ argument in my_sqrt\n"); // используется библиотечный класс`
- `throw Bad_index();// используется пользовательский // класс`

Плохие решения (используются строковые литералы):

- `throw "Bad index\n";`
- `throw "Invalid argument\n";`
- `throw "Foo\n";`
- `throw "Help\n";`

Объект исключения

Объект исключения автоматически создается компилятором. Он строится на основе того объекта, который «выбрасывает» инструкция **throw**. Например:

```
throw std :: length_error("Недопустимо большой размер "
                          "объекта \n");
```

После того как отработает инструкция **throw**, временный анонимный объект будет уничтожен. Но перед его уничтожением компилятор снимет копию с этого временного объекта. Эта копия будет сохранена компилятором в надежном месте. Указанная копия и называется объектом исключения. Объект исключения доступен любому обработчику **catch**, который способен обрабатывать данный тип исключений.

После создания объекта исключения управление должно покинуть тело той функции, в которой сгенерировано исключение. Однако до этого события компьютер должен выполнить так называемую раскрутку стека (**stack unwinding**).

Раскрутка стека

В процессе раскрутки стека происходит уничтожение локальных объектов. Для локальных объектов, имеющих пользовательский тип, автоматически вызываются деструкторы. Для удаления локальных объектов, имеющих встроенный тип, компилятор ничего не делает. Для объектов, созданных с помощью оператора new, компилятор не будет автоматически вызывать оператор delete с целью принудительного освобождения памяти.

Перейдем теперь к рассмотрению второй обязанности разработчика функции, которая состоит в том, что необходимо уведомить клиента о том какие исключения он должен обрабатывать. Одна из специальных возможностей предусмотрена стандартом языка C++. Эта возможность носит название спецификации исключения.

Спецификации исключений

Это необязательная конструкция. Программист может ее записывать в объявлении и определении функции. Для определенности будем рассматривать объявление (прототип) функции, которое с учетом спецификаций исключения будет иметь следующую структуру.

<α>[β] ;

Здесь α – заголовок функции, β – необязательная спецификация исключения. Отметим, что имеются два формата спецификаций исключения.

- **throw**(γ), где γ – список типов исключений.
- **throw().**

Второй формат спецификаций исключений используется в тех случаях, когда разработчик функции предполагает, что функция не будет выбрасывать исключения.

Приведем пример.

```
class A
{
};

class B
{
};

class C
{
};

void f() throw(A, B);
void h() throw();
```

В программном коде этого примера вначале имеются определения трех классов исключений с именами `A`, `B` и `C`. Затем приводятся прототипы двух функций, которые содержат спецификации исключений. Предполагается, что функция `f()` может выбросить исключения типа `A` и `B`. Функция `h()` не должна выбрасывать исключений.

Теперь казалось бы все хорошо. Клиент знает, что ему необходимо создать два обработчика **catch**. Один из них должен обрабатывать исключения типа A, а второй обработчик – для типа B.

Но что произойдет в том случае, когда функция выбросит исключение класса C. Это может, например, произойти так:

```
void f() throw(A, B)
{
    //..
    g();
}

void g()
{
    //..
    throw C();
}
```

Эта ситуация известна как выброс непредвиденного исключения. Возможность появления такой ситуации усложняет использование спецификаций исключения.

Работа с обработчиками

Перейдем к рассмотрению того, что должен предусмотреть клиент в своем коде. Если кратко сформулировать его действия, то это будет выглядеть следующим образом: блоки **try**.

Формат блока **try**.

```
try
<α>
<β>
```

Здесь α – составная инструкция, а β – список обработчиков (блок обработчиков).

Пример.

```
try
{
   f1();
   f2();
   // …
}
// Блок обработчиков. Показан только один обработчик
catch(//..)
{
   //..
}
```

Формат обработчика

```
catch(<α>)
 <β>
```

Здесь α – своего рода единственный формальный параметр, а β – тело обработчика. Если объект внутри тела не используется, то в качестве параметра обработчика может использоваться тип обработчика. Рассмотрим следующий пример.

Вариант 1.

```
catch(std :: out_of range err)
{
   //..
}
```

В этом варианте объект исключения передается обработчику **catch** по значению. Это приведет к созданию временного объекта. Для устранения такого нежелательного явления предпочтительнее принимать объект по ссылке. Это и делается во втором варианте обработчика.

Вариант 2

```
catch(std :: out_of rang& err)
{
   //..
}
```

Формальный параметр обработчика может быть так называемым абстрактным объявителем, т. е. информацию только о типе исключения. При этом информация о имени параметра не указывается. Например:

```
// Код разработчика
class MyError
{
   //..
};

// Код клиента
catch(MyError)
{
   //..
}
```

Допустима в обработчике исключения так называемая эллиптическая сигнатура. В этом случае обработчик оказывается совместимым с любым типом исключений. Например:

```
catch(…)
{
   //..
};
```

Такой обработчик может реагировать на исключения любого типа. Он должен располагаться последним обработчиком в блоке.

Пример обработки исключений

```
// Код разработчика
class Array
```

```
{
  public:
    enum {MAXSIZE = 20};
    struct BadIndex
    {
       int index;
       BadIndex(int index) : index_(index){}
    };
    Array(int capacity = MAXSIZE)
    {
        //..
    }
    double& operator[](int index) throw(BadIndex)
    {
       if(index >= 0 && index < capacity_)
              return ptr_[index];
       else
              throw BadIndex(index);
    }
   private:
     int capacity_;
     double *ptr_;
}

// Код клиента
int main()
{
   Array ar;
   try
   {
      ar[20] = 10;
      // ..
   }
   catch(Array :: BadIndex& err)
   {
      cout << "Ошибка BadIndex для значения индекса ="
           << err.index << endl;
   }
}
```

Современная точка зрения на спецификации исключения

Ряд авторов не рекомендуют использовать спецификации исключений. К их числу относятся Г. Саттер и А. Александреску, авторы книги «Стандарты программирования на языке C++». В книге содержится совет с номером 75 «Избегайте спецификаций исключений».

Альтернативой использованию спецификаций исключений может служить написание комментария с указанием возможных исключений.

Стандартные классы исключений

В стандартной библиотеке языка C++ предусмотрена система классов, предназначенных для работы с исключениями. Базовым классом этой системы является класс exception. Этот класс объявлен в заголовке exception.

Интерфейс класса exception имеет следующий вид

```
namespace std
{
   class exception
   {
      public:
         exception()throw()
         exception(const exception&)throw();
         exception& operator=(const exception&)throw();
         virtual ~exception()throw();
         virtual const char* what() const throw();


   };
}
```

Стандартные классы исключений подразделяются на три категории:

- языковой поддержки.
- стандартной библиотеки C++.
- внешних ошибок.

Классы языковой поддержки

Исключения этой категории используются на уровне языка C++. В связи с этим их логичнее было бы отнести к базовому языку, чем к библиотеке. Исключения этой категории выбрасываются при неудачном завершении выполнения некоторых операций. К таким исключениям относятся следующие классы исключений:

- bad_alloc. Исключения этого класса выбрасываются при неудачном выполнении глобального оператора new (кроме версии с запретом исключений).
- bad_cast. Исключения этого класса могут быть выброшены при выполнении оператора dynamic_cast, если преобразование типа по ссылке во время завершения операции закончится неудачей.
- bad_typeid. Это исключение может быть выброшено при выполнении оператора typeid. Указанный оператор предназначен для идентификации типа во время выполнения. Если аргументом оператора typeid ноль или нулевой указатель, то генерируется исключение.
- bad_exception. Это исключение выбрасывается при работе с непредвиденными исключениями.

Классы исключений стандартной библиотеки

Эти классы обычно являются производными от класса logic_ error

ШАБЛОНЫ ФУНКЦИЙ

Шаблон функции представляет собой обобщенное описание семейства функций. Это своего рода план, используя который компилятор может создавать определения конкретных функций.

Процесс создания определений функций на основе их шаблона называется конкретизацией (инстанцированием) шаблона. Результат инстанцирования называют экземпляром (специализацией) Существуют два вида инстанцирования: явное и неявное.

Основным видом инстанцирования функций является неявное инстанцирование. Важным преимуществом неявного способа инстанцирования является то положение, что использование шаблона функции оказывается похожим на использование обычной функции.

Отметим, что определение шаблона функции следует располагать в заголовочном файле.

Объявление и определение шаблона функции

Объявление и определения шаблона функции должны в качестве своего рода префикса содержать новый структурный элемент следующего вида:

```
template<α>
```

Здесь **template** – зарезервированное слово, а α – список параметров шаблона.

Таким образом, к обычным параметрам функции добавляются еще параметры шаблона. Параметры шаблона можно разделить на три категории:

- Параметры обобщенные типы.
- Параметры константы.
- Параметры-шаблоны, являющиеся шаблонами класса.

С помощью параметров первой группы передается информация о конкретных типах данных, для которых компилятор должен инстанцировать определение функции.

С помощью параметров второй группы в определение инстанцируемой функции передаются константы.

Элементы списка параметры-шаблоны применяются в тех случаях, когда в теле функции используются шаблоны классов.

Элементы списка параметров шаблона имеют следующий формат:

<α><β>

В таблице, приводимой ниже, содержатся пояснения для компонентов этого формата.

Вид параметра	α	β
Обобщенный тип	class или typename	Идентификатор обобщенного типа
Параметр-константа	Спецификатор типа	Идентификатор
Параметр-шаблон	template<список-шаблона> class	Идентификатор

Приведем примеры префиксов шаблона функций

Пример 1.

```
template <class T>  или
template <typename T>
```

Пример 2.

```
template <typename T1, typename T2>
```

Пример 3.

```
template <typename T, int size>
```

Пример 4.

```
template <typename T, template <typename ElementType>
          class Cont>
```

Зарезервированное слово typename в языке С++ появилось сравнительно не давно. В префиксе шаблона функции можно использовать как зарезервированное слово class, так и зарезервированное слово typename/ Предпочтение лучше отдать зарезервированному слову typename/

Примеры объявлений и определений шаблонов функций

Пример 1. Шаблон функции нахождения максимального значения двух величин

```
// Объявление функции
template <typename T>  // Префикс шаблона функции
T max(T a, T b);       // Объявление функции
                       // T – имя обобщенного типа

// Определение функции
template <typename T>  // Префикс шаблона функции
T max(T a, T b)        // Заголовок функции
{                      // Тело функции
     return (a > b)? a : b;
}
```

Пример 2.Обмен значений двух величин

```
// Объявление функции
template <typename T>  // Префикс шаблона функции
void swap(T& a, Tb);   // Объявление функции
                       // T – имя обобщенного типа

// Определение функции
template <typename T>  // Префикс шаблона функции
T max(T a, T b)        // Заголовок функции
{                      // Тело функции
     T temp = a;
     a      = b
     b      = temp;
}
```

Инстанцирование шаблона функции

Инстанцирование шаблона функции состоит в построении определения конкретной функции. Различают явное и неявное инстанцирование шаблона функции. Наиболее часто используется неявное инстанцирование шаблона функции.

Неявное инстанцирование

Неявное инстанцирование выполняется на основе анализа компилятором вызова функции. Компилятор анализирует типы аргументов вызова функции. На основе этого анализа определяются аргументы шаблона. Вообще говоря, этот процесс не всегда завершается успехом. Такие случаи будут рассмотрены. Однако, если такой процесс оказался успешным, то аргументы шаблона в его определении заменят параметры шаблона и компилятор построит определение конкретной функции. Отметим, что инстанцирование выполнять не будет (в нем нет необходимости) в том случае, когда определение конкретной функции уже было ранее построено.

Пример. Инстанцирование шаблона функции для определения максимального значения двух величин

```
#include <iostream>
using std :: cout;
using std :: endl;

template <typename T>
T Max(T a, Tb);

int main()
{
   int i = 2;
   int j = 3;
   cout << Max(i, j) << endl;

   double d1 = 1.0;
   double d2 = 5.0;
   cout << Max(d1, d2) << endl;

   double d3 = 4.0;
   double d4 = 8.0;
   cout << Max(d3, d4) << endl;

   // cout << Max(i, d3) << endl;
   return 0;
}

template <typename T>
T Max(T a, Tb)
{
    return (a > b) ? a : b;
}
```

В приведенном выше программном коде шаблон функции Max инстанцируется дважды:

- Для вызова Max(i, j)/
- Для вызова Max(d1, d2)

Необходимость в инстанцировании шаблона при компиляции вызова Max(d3, d4).отсутствует. При попытке компиляции вызова Max(i, d3) появляется сообщение об ошибке (error):

```
Could not find a match for Max<>(int, double)
```

При компиляции первого вызова (Max(I, j)) компилятор устанавливает, что в шаблоне функции необходимо сделать подстановку T == int. Аналогично при компиляции второго вызова (Max(d1, d2)) компилятор выявит необходимость в подстановке T == double.

При компиляции четвертого вызова (Max(I, d3)) возникает противоречие, состоящее в следующем. Если исходить из первого аргумента вызова функции (i), то необходимо выполнять подстановку T == int. В то же время, исходя из второго параметра вызова (d3), необходимо выполнить другую подстановку: T == double. Возникающий конфликт компилятор разрешить не может. Заменим, что автоматические преобразования типов при инстанцировании шаблона не допускаются.

В последних версиях компилятора рассматриваемая трудность может быть преодолена с помощью явного аргумента шаблона. Для этого вызов Max2(i, d3) следует заменить вызовом Max<double>(I, d3). Это позволит выполнить программу со снятым комментарием

Явное инстанцирование конкретной функции

Эта разновидность инстанцирования выполняется с помощью директивы явного инстанцирования. Директива явного инстанцирования начинается зарезервированным словом **`template`**. За этим словом должно следовать объявление той специализации, которую необходимо сгенерировать. Например, при использовании директивы явного инстанцирования :

```
template void Swap<double>(double&, double&);
```

будет сгенерировано определение конкретной функции из семейства `Swap`, для которой `T` == **`double`**.

Директива явного инстанцирования создает так называемую точку явного инстанцирования `POI` (`point of instantiation`). `POI` – это позиция в исходном коде, в которой шаблон мысленно разворачивается путем подстановки аргументов шаблона вместо параметров.

Структура использования шаблона функции с явным инстанцированием

Организуются три файла, из которых два файла имеют расширение `.h`, а третий - расширение `.cpp`. Например.

```
// Файл Max.h
// Содержит только объявлении шаблона функции Max
template <typename T>
T Max(T a, T b);

// Файл MaxDef.h
#include "Max.h"
template <typename T>
T Max(T a, T b)
{
    return (a > b) ? a : b;
}

// Файл инстанцирования MaxInstances.cpp
#include MaxDef.h
// Директивы явного инстанцирования. Например:
template float Max<float>(float, float);
```

Теперь в код клиента необходимо будет включать только файл `Max.h`

```
// Код клиента
#include "Max.h"
// …
```

Перегрузка шаблона функции

Шаблоны функций можно перегружать. Наряду с шаблоном `Max`, который был приведен ранее, можно написать и другие шаблоны с тем же самым именем `Max`. Ниже в качестве примера приводится определение шаблона `Max`, предназначенного для определения значения наибольшего элемента массива.

```
template <typename T>
T Max(const T* arr, int size)
{
    T   m = arr[0];
    for(int i = 1; i < size; i++)
       if(arr[i] > m)
           m = arr[i];
    return m;
}
```

Явная специализация шаблона функции

Иногда обобщенный алгоритм не может быть использован для некоторого типа данных. Обратимся к уже рассмотренному шаблону функции Max для нахождения максимального значения двух величин. Таким шаблоном нельзя воспользоваться для типа const char *. Для типа, выпадающего из обобщенного алгоритма, можно построить так называемую явную специализацию.

Пусть имеется шаблон функции

```
template <typename T>
T Max2(T a, Tb)
```

```
{
   return (a > b) ? a : b;
}
```

Напишем для этого шаблона функции явную специализацию, представляющую собой законченное определение функции, когда T == const char*. Объявление и определение явной специализации должны начинаться с префикса template<>.

```
// Явная специализация шаблона функции Max
template<>
const char* Max(const char* s1, const char* s2)
{
    return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

Кроме явной специализации можно написать обычную функцию, имя которой совпадает с именем шаблона. Преимуществом явной специализации по отношению к обычной функции является то положение, что в явной специализации не допускаются автоматические преобразования типа.

ШАБЛОНЫ КЛАССОВ

Шаблон класса представляет собой обобщенное определение семейства классов. Шаблон класса определяет поля и функции порождаемых от него классов. Шаблоны классов следует применять в том случае, когда функции класса не зависят от типа данных. Наибольшее применение шаблоны классов нашли при разработке контейнеров.

Объявление и определение шаблона класса отличаются от объявлений и определений обычных классов наличием префикса, который строится по тем же правилам, что и префикс шаблона функции.

Здесь так же, как и при работе с шаблонами функций следует различать три категории параметров шаблонов:

- Параметры-обобщенные типы.
- Параметры-константы.
- Параметры-шаблоны классов.

Параметры шаблона классов в целом имеют то же назначение, что и параметры шаблона функции.

Приведем пример определения шаблона класса, в котором будут присутствовать параметры шаблона двух видов: параметры-обобщенные типы и параметры-константы. При этом будут учитываться следующие положения.

1. Все определение шаблона, включая определения вынесенных функций, будет располагаться в заголовочном файле.
2. В соответствии с правилами хорошего стиля программирования в теле шаблона будем размещать только прототипы его функций.
3. При размещении определения функции вне тела шаблона его программный код следует предварять префиксом шаблона.
4. Имя функции шаблона при записи ее определения вне тела шаблона следует уточнять с помощью оператора расширения

области видимости «::». В состав уточняющего квалификатора кроме имени шаблона должен входить список параметров шаблона.

```
// Шаблон стека. Файл Stack.h
template <typename T, int size>
class Stack
{
    public:
      Stack();
      bool isFull();
      bool isEmpty();
      bool push(const T& item);
      bool pop(T& item);
    private:
      T items[size];
      int top;
};

template <typename T, int size>
Stack<T, int size> :: Stack() : top(-1)
{
}

template <typename T, int size>
bool Stack<T, int size> :: isEmpty()
{
     return top == -1;
}

template <typename T, int size>
bool Stack<T, size> :: push(const T& item)
{
    if(isFull())
      return false;
    items[++top] = item;
    return true;
}
```

Заметим, что при использовании шаблона функции в качестве имени типа выступает не имя класса, а имя класса, дополненное списком имен параметров шаблона, которые должны быть заключены в угловые скобки.

В данном примере стек строится на основе массива, располагаемого в стеке. Список параметров шаблона содержит два элемента. Первый из них T является обобщенным типом. Он определяет тип элементов, которые будут храниться в стеке. а второй параметр size относится к категории параметров-констант. Он будет определять размер памяти массива.

Использование шаблона класса

Шаблоны класса отличаются от шаблонов функций способом использования. Аргументы шаблона функции обычно выводятся неявным

образом. Напротив конкретные классы строятся на основе определений объекта, в котором в явном виде указываются значения аргументов шаблона. Например.

```
Stack<int, 25> st1;
Stack<double, 15> st2;
```

При компиляции программного кода, приведенного выше, компилятор должен построить определения двух конкретных классов ()двух стеков. В первом из них используется массив, предназначенный для хранения до 20 элементах типа int, а во втором – массив, рассчитанный на хранение 15 элементов типа double.

Определение шаблона может содержать значения, принимаемые по умолчанию. Например.

```
template <typename T = int, int size = 20>
class Stack2
{
   // ...
};
```

Использование шаблона Stack2.

```
Stack2<> st3;  // Альтернативное определение:
               // Stack2<int,20>
Stack2<double> // Альтернативное определение
               // Stack2<double, 20>
```

Наследование и шаблоны

Возможны следующие разновидности наследования:

1. Шаблон класса может быть порожден от шаблонного класса.
2. Шаблон класса может быть порожен от нешаблонного класса.
3. Нешаблонный класс может быть порожден от шаблонного класса.

Пример 1. Наследование шаблона класса от шаблона класса.

```
// базовый класс
template <typename T>
class Array
{
   // ...
   private:
     T* p;
     int size_;
};

// Порожденный класс
template <typename T>
class GrowArray : public Array<T>
{
    // ...
};
```

Пример 2. Шаблон класса порождается от нешаблонного класса

Рассматриваемый пример представляет интерес по нескольким причинам:

1. Отсутствует «разбухания» кода.
2. Демонстрируется целесообразность использования private-наследования.
3. Используется вложенный класс.

```
// Базовый класс Файл BaseStack.h
// Назначение класса состоит в реализации функциональности
// стека
class BaseStack
{
   protected:
     BaseStack();
     ~BaseStack();
     void push(void* Object);
     void* pop();
     bool isEmpty()const;

   private:
     struct Node    // Вложенный класс: Узел
     {
         void* data_;
         Node* next_;
         Node(void* data, Node* next) : data_(data),
              next_(next)
         {
         }
     };

     Node* top;
     BaseStack(const BaseStack& rhs);
     BaseStack& operator=(const BaseStack& rhs);
};

// Реализация базового класса BaseStack. Файл BaseStack.cpp
// Программный код не приводится

// Порожденный с помощью private-наследования шаблон класса
// Stack<T>. Назначение шаблона состоит в реализации
// контроля типов. Файл Stack.h
#include "BaseStack.h"
template <typename T>
class Stack : public BaseStack
{
     public:
       void push(T* ObjectPtr);
       {
           BaseStack :: push(ObjectPtr);
       }

       T* pop()
       {
           return static_cast<T">(BaseStack :: pop());
       }
```

```
        bool isEmpty()const
         {
              return BaseStack :: isEmpty();
         }
};
```

Обсуждение.

Реализация стека состоит из трех классов:

1. BaseStack – нешаблонный класс, реализующий функциональность стека;
2. Node – нешаблонный класс, вложенный в класс BaseStack.
3. Stack – шаблонный класс, порожденный с помощью private-наследования от класса BaseStack.

Обращает на себя внимание отсутствие у класса BaseStack интерфейса пользователя. Интерфейс, который предоставляется этим классом, может быть назван интерфейсом наследования. Его элементы находятся в protected-секции. В частности объекты типа BaseStack могут создавать только функции-члены классов, порожденные от рассматриваемого класса. Для этой цели порожденные классы могут воспользоваться защищенным конструктором умолчания.

В private-секции рассматриваемого класса содержится объявление класса Node. Этот класс можно рассматривать как элемент реализации класса BaseStack. Дело в том, что объекты класса Node должны создаваться только функциями-членами класса BaseStack.

Интерфейс пользователя, обеспечивающий контроль типов, создается шаблоном класса, который порождается от класса BaseStack с помощью private-наследования. Следует отметить, что функции этого класса чрезвычайно просты. Поэтому они могут быть реализованы встроенными.

Пример 3. Порождение нешаблонного класса от шаблона класса.

```
template <typename T>
class B
{
    // ...
};

class D : public B<int>
{
    // ...
};
```

Шаблоны классов и отношение включения

Шаблон класса может использоваться в качестве компонента другого шаблона. В реализации шаблона Stack в качестве его компонента используется шаблон массива.

```
// Компонент (часть)
template<typename T>
class Array
{
```

```
    // ...
};

// Композит (общий)
template <typename T>
class Stack
{
   public:
      // ...
   private:
      Array<T> ar;
      // ...
};
```

Рекурсивное использование шаблонов классов

Шаблоны классов можно применять рекурсивно. Пусть имеется следующий шаблон массива.

```
template <typename T, int size>
class Array
{
   // ...
};
```

Располагая таким определением шаблона, его можно использовать следующим образом:

```
Array<Array<int, 5>, 10> matrix;
```

Это определение создает массив, состоящий из 10 элементов, каждый из которых в свою очередь является массивом из 5 элементов типа int. Таким образом, получаем двумерный массив, который может хранить до 10 строк и 5 столбцов.

Если в шаблоне Array перегружен оператор operator[], то к элементам matrix можно обращаться следующим образом matrix[i][j].

Друзья и шаблоны классов

При использовании «друзей» в шаблонах классов возникают определенные трудности. Ограничимся рассмотрением этого вопроса применительно к «дружественным» функциям. Более того, будем рассматривать конкретный пример, в котором для шаблона Array необходимо перегрузить оператор поместить в поток «<<».

Рассмотрим два варианта такой перегрузки.

1. Легко реализуемый вариант, когда определение перегружающей функции располагается в теле шаблона: недостаток – несоответствие стилю хорошего программирования.
2. Более сложно реализуемый вариант, но отвечающий хорошему стилю программирования, когда определение перегружающей функции находится вне тела шаблона.

Вариант 1.

```
template <typename T>
class Array
```

```
{
    public:
     friend ostream7 operator<<(ostream& os,
                                const Array<T>& rhs)
     {
         // ...
     }
};
```

Вариант 2.

Здесь необходимы следующие дополнительные компоненты:

- Опережающее объявление для перегружающей шаблонной функции.
- Указание в теле шаблонного класса на то, что дружественная функция является шаблоном.

```
// Опережающее объявление для шаблонного класса Array,
// необходимо для последующего опережающего объявления
// дружественной функции
template <typename T>
class Array;

// Опережающее объявление дружественной функции
template<typename T>
ostream& operator<< (ostream& os, const Array<T>& rhs);

template <typename T>
class Array
{
    public:
       friend ostream& operator<< <>(ostream& os,
                                     const Array<T> rhs);
       // ...
};

template <typename T>
ostream& operator<<(ostream& os, const Array<T>& rhs)
{
    // ...
}
```

Явная и частичная специализация шаблона класса

Иногда конкретный класс, получаемый из шаблона класса, по тем или иным причинам оказывается неприемлимым. В этом случае можно использовать либо явную, либо частичную специализацию шаблона класса.

Частичная специализация – это альтернативное определение шаблона класса, в котором некоторые из параметров шаблона заменены аргументами. Явная специализация имеет место, когда в частичной специализации выполнена замена всех параметров шаблона.

Здесь могут быть полезными следующие понятия:

- Первичный шаблон.
- Вторичный шаблон.

Первичный шаблон – это исходный шаблон, а вторичный шаблон – его специализация.

Пример.

```
// Первичный шаблон
template <typename T1, typename T2>
class Pair
{
   // ...
};

// Частичная специализация.
// Зафиксирован второй параметр шаблона
template <typename T1>
class Pair<T1, int>
{
   // ...
};

// Явная специализация.
// Зафиксированы все параметры шаблона
template <>
class Pair<int, int>
{
   // ...
};
```

Обсуждение.

Основной принцип использования состоит в том, что компилятор выбирает наиболее специализированный шаблон.

Пусть имеется следующий клиентский программный код

```
Pair <double, double> pr1;
```

В этом случае ни один из вторичных шаблонов не подходит. Поэтому будет использоваться первичный шаблон.

Пусть имеется другой клиентский программный код

```
Pair <double, int> pr2;
```

В этом случае будет использоваться частичная специализация.

Наконец для клиентского кода

```
1. Pair <int, int> pr3;
```

будет использована явная специализация.

Алгоритмы

Это существенная часть библиотеки STL. Алгоритмы более всего похожи на обычную библиотеку. Основное отличие их от традиционной библиотеки они (алгоритмы) реализованы в виде шаблонов функций.

Алгоритмы – это внешние функции, работающие с итераторами. Преимущество такого подхода состоит в следующем. Вместо того, чтобы разрабатывать такой алгоритм для каждого вида контейнера, достаточно реализовать его один раз для обобщенного контейнера.

Следует отметить один важный факт. В основе использования алгоритмов лежит скорее парадигма процедурного программирования, а не

парадигма объектно-ориентированного программирования. При работе с алгоритмами данные разделяются на части, взаимодействующие через некоторый интерфейс.

Один из недостатков такого подхода состоит в том, что некоторые комбинации данных и алгоритмов могут оказаться недопустимыми и еще хуже, допустимыми, но мало эффективными.

Алгоритм for_each

Формат.

```
template <typename InIter, typename UnaryFunction>
void for_each(InIter first, InIter last,
              UnaryFunction f);
```

Алгоритм for_each вызывает функцию f для каждого элемента последовательности, задаваемой двумя итераторами [first, last). Здесь f так называемая унарная функция. Унарная функция – это функция, принимающая один аргумент.

Пример 1. Имеется контейнер типа list<int>. Требуется организовать вывод элементов контейнера на дисплея с помощью алгоритма for_each.

```
#include<ostream>
#include<list>
#include<algoritm>
using namespace std;

void print(int x)
{
   cout << x << '\t';
}

int main()
{
    int ar[] = {1, 3, 5, 7};
    list<int> lst(ar, ar + n);
    for_each(lst.begin(), lst.end(), print);
    cout << endl;
}

// Вывод не экран дисплея
1    3    5    7
```

Проблема. Каким образом в случае необходимости передавать информацию в функцию f. Например, каким образом организовать вывод на экран дисплея не всех элементов контейнера, а только тех, которые удовлетворяют некоторому условию (например, только положительные элементы).

Выход из положения состоит в применение так называемых функциональных объектов.

Функциональные объекты

Функциональный объект – это экземпляр класса, в котором перегружен оператор функция: operator().

Пример 2. Дан контейнер типа list<int>. Вывести на экран дисплея элементы контейнера, значения которых не превышают величины a.

```
// директивы препроцессора
// Класс, экземпляры которого являются объект-функциями
class print_if
{
    public:
      print_if(int a) : a_(a)
      {
      }

      void operator()(int value)
      {
         if(value <= a)
           cout << value << '\t';
      }
    private:
      int a_;
};
// Клиентский код
int main()
{
   int a;
   // Ввод a
   list<int>  lst;
   // Работа с контейнером lst

   // Вывод с использованием алгоритма for_each
   for_each(lst.begin(), lst.end(), prinf_if(a));
   // ...
}
```

В третьем параметре алгоритма for_each создается анонимный (без имени) объект, а ожидается вызов функции. Особенностью класса print_if является наличие перегруженного оператора функция: operator(). Наличие этого перегруженного оператора и принимает во внимание компилятор. При выполнении алгоритма for_each в рассматриваемом примере для каждого элемента контейнера lst из диапазона, задаваемого первым и вторым параметрами алгоритма, будет выполняться перегруженный оператор функция.

Алгоритм copy

Существуют два способа копирования:

- Замещения.
- Вставки

По умолчанию используется способ замещения. Способ вставки требует использования так называемого адаптера итератора.

Объявления алгоритма copy

```
template <typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last, OutIter dest);
```

Алгоритм copy работает с двумя контейнерами:

- Контейнером источником.
- Контейнером приемником.

Каждый из контейнеров представлен итераторами. Первый из контейнеров представлен двумя итераторами, а второй только одним.

Параметры, относящиеся к контейнеру источнику, задают диапазон копируемых значений, а единственный параметр контейнера приемника задает позицию, начиная с которой будут размещаться копируемые элементы.

Элементы последовательности [first, last) копируется в последовательность, которая начинается с dest. Причем элемент

- `*first` замещает `*dest`
- `*(first + 1)` замещает `*(dest + 1)`
- и т. д.

Алгоритм возвращает итератор dest + (last - first).

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

void Print(int x)
{
    cout << x << '\t';
}

int main()
{
    int ar1[] = {-2, 5, 4, 3};
    vector<int> vct(ar1, ar1 + 4);
    for_each(vct.begin(), vct.end(), Print);
    cout << endl;

    int ar2[] = {1, 2, 3, 4, 5};
    list<int> lst(ar2, ar2 + 5);
    for_each(lst.begin(), lst.end(), Print);
    cout << endl;

    copy(vct.begin(), vct.end(), lst.begin());
    for_each(lst.begin(), lst.end(), Print);
    cout << endl;

    copy_backward(vct.begin(), vct.end(), lst.end());
    for_each(lst.begin(), lst.end(), Print);
    cout << endl;
```

```
    vector<int> vct2(ar1, ar1 + 4);
    for_each(vct2.begin(), vct2.end(), Print);
    cout << endl;

    list<int> lst2(ar2, ar2 + 5);
    for_each(lst2.begin(), lst2.end(), Print);
    cout << endl;

    copy(lst2.begin(), --lst2.end(), inserter(vct2, vct2.begin()
+ 2));
    for_each(vct2.begin(), vct2.end(), Print);
    cout << endl;

    cout << "Hello world!" << endl;
    return 0;
}
```

алгоритм sort

Сортировка диапазона на месте. Имеются две разновидности этого алгоритма. Объявление первого из них имеет следующий вид

```
template<typename RandIter>
void sort(RandIter first, RandIter last);
```

Объявление второй разновидности

```
template<typename RandIter, typename Compare>
void sort(RandIter first, RandIter last, Compare comp);
```

Шаблон sort сортирует диапазон [first, last) на месте. Сортировка не стабильная, поэтому равные элементы не сохраняют свой исходный порядок.

Первая версия сравнивает при помощи оператора >. Вторая версия использует функцию двух переменных для проверки истинности выражения comp().

Пример

```
int tar[6] = {5, 4, 3, 2, 1, 0};
sort(ar, ar + 6);
```

В результате выполнения этого вызова алгоритма sort получим

```
ar = {0, 1, 2, 3, 4, 5}
```

Сортировка по убыванию

В качестве функции, определяющей критерий сортировки, будет использоваться стандартный объект-функция greater. Пример

Имеется контейнер v типа vector<int>. Вызов алгоритма следующего вида

```
sort(v. begin(), v.end(), greater<int>());
```

обеспечивает сортировку в порядке убывания.

Сортировка данных пользовательского типа

Один из возможных способов использования состоит в перегрузке операторов < и >. Пример

```
struct Person
{
   string name_;
   int    age_;
   double salary;
   // ctor
   bool operator<(const Person& person)const
   {
      return age_ < person.age_;
   }

   bool operator>(const Person& person)const
   {
      return age_ > person.age_;
   }
};
```

```
bool cmp_salary(const Employee& e1, const Employee& e2)
{
  return e1.salary < e2.salaty;
}
```

Клиентский код

```
int main()
{
   vector<Person> ar;
   // …
   // Сортировка по возрастанию
   sort(v.begin(), v.end{});
   // Сортировка по убыванию
   sort(v.begin(), v.end(), greater<Person>());
   // Сортировка с использованием функции cmp_salary()
   std :: sort(vec.begin(), vec.end(), cmp_salary);
}
```

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

1. Абстрактный класс в языке C++ – это класс, в котором имеется хотя бы одна чисто виртуальная функция. Особенностью абстрактного класса является невозможность создания объектов.
2. Абстрактный тип данных.
    a. Определение, которое приводит Коплиен Дж.[14, стр. 25]. Абстрактные типы данных (АТД) определяют интерфейс абстракции данных без указания подробностей реализации.

b. Определение, приведенное в книге[17, стр. 130]. Абстрактный тип данных (abstract data type). Тип, представление которого скрыто. Используя абстрактный тип данных, достаточно знать только то, какие операции он использует.
c. Определение Мейера Б. [22, стр.1179]Абстрактный тип данных – это множество математических элементов, характеризуемое перечислением функций, применимым ко всем элементам, и формальными свойствами этих функций.
d. Определение Френка Каррано и Джанет Причард [13, стр. 806]. Абстрактный тип данных (АТД) – совокупность данных и точно определенных операций над ними.
e. Определение Роберта Седжвика. АДТ – это тип данных (набор значений и совокупность операций для этих значений), доступ к которым осуществляется только через интерфейс. … Ключевым отличие, делающее тип данных абстрактным, характеризуются словом только.

3. Артефакт (artifact) называется любой результат работы: код, графическое изображение для Web, схема базы данных, текстовые данные, диаграммы, модели и т.д.[16, стр. 64].
4. Зацепление (coupling) – степень, с которой различные программные компоненты связаны между собой. Обычно носит негативный оттенок. Зацеплению противопоставляется связность. Зацепление описывает отношения между модулями, а связность – внутри модуля [3, стр. 430].
5. Инвариант класса. Дадим два варианта определения этого термина:
   - утверждение, которое должно выполняться при создании каждого экземпляра класса и сохраняться каждой экспортируемой подпрограммой класса, так что оно будет всегда выполняться для всех экземпляров класса при внешнем наблюдении за ними [22, стр. 1181].
   - правила, определяющие, каким образом используются переменные – члены для представления значения [26, стр. 135]
6. Инкапсуляция (сокрытие информации). В объектно-ориентированном программировании инкапсуляция предусматривает выполнение трех операций:
   a. Объединение данных и обрабатывающих их подпрограмм в одной синтаксической конструкции (классе).
   b. Отделение интерфейса от реализации
   c. Сокрытие реализации для клиента класса.
7. Инстанцирование (Instaniation) – создание экземпляров класса [16, стр. 706].
8. Интерфейс класса.
   - Открытый интерфейс класса. Определяется совокупностью функций, объявленных в разделах public класса.

- Интерфейс внешнего клиента. Включает: а) открытый интерфейс, б) интерфейс, определяемый дружественными компонентами класса, в) интерфейс, определяемый внешними (глобальными функциями).
- Защищенный интерфейс (интерфейс наследования). Определяется функциями, объявленными в разделе protected.
- Закрытый интерфейс. Это интерфейс разработчика. Определяется совокупностью функций, объявленных в разделе private/

9. Класс. В объектно-ориентированном программировании класс является средством частичной или полной реализации абстрактного типа данных.
10. Конкретный класс (Concrete class) – класс, который может иметь экземпляры [16, стр. 706].
11. Конструктор. Специальная функция класса, используемая для целей инициализации объекта.
12. Конструктор умолчания. Конструктор, не имеющий параметров.
13. Модель (Model) – описание статических или динамических характеристик системы, представленное с различных точек зрения (обычно в текстовом виде или в виде диаграмм) [16, стр. 707].
14. Модуль. Под модулем будем понимать единицу декомпозиции программы. В процедурном программировании такой единицей является подпрограмма, а в объектно-ориентированном программировании - класс.
15. Объект. Объектом называется экземпляр класса. В языке C++ переменная классового типа является объектом.
16. Объектно-ориентированный язык программирования (Object-oriented programming language) язык программирования? поддерживающий принципы инкапсуляции, наследования и полиморфизма. Возможность определять полиморфные операции [16, стр. 707].
17. Открытое наследование (public - наследование) - языковой механизм для реализации отношения между классами, известного как общее / частное.
18. Полиморфизм (Polymorphism) – возможность разной реакции различных классов или объектов на одно и то же сообщение, основанная на использовании полиморфных операций [16, стр. 709].
19. Полиморфная операции (Polymorphic operation) – операция, которая по-разному реализуется различными классами [16, стр. 709].
20. Пользовательский тип данных. Этот термин иногда используют в качестве синонима абстрактному типу данных. Например, в [31, стр. 68] читаем : «C++ стремится решить задачу, позволяя пользователю непосредственно определять типы, которые ведут себя (почти) также, как и встроенные. Я предпочитаю термин тип, определяемый пользователем (пользовательский тип). Достаточно точное

абстрактных типов данных потребовало бы «абстрактной» формулировки.»

21. Принцип LSP (принцип подстановки Лисков). Этот принцип является критерием, позволяющим определить возможность использования public-наследования. В соответствии с этим принципом public-наследование допустимо в том случае, когда порожденный класс является разновидностью базового класса.
22. Постусловие (Post-condition) – ограничение, которое должно соблюдаться после выполнения операции [16, стр. 709].
23. Предусловие (Pre-condition) – ограничение, которое должно соблюдаться до выполнения операции [16, стр. 709].
24. Проектирование по контракту – это метод конструирования ПО, в котором проектируемые компоненты системы взаимодействуют друг с другом точно определенных контактов [22, стр. 1188].
25. Связность (cohesion) – степень связности компонентов единой программной системы (таких, как элементы одного класса). Связность противопоставляется зацеплению (зацепление описывает отношения между модулями, а связность – внутри модуля) [3, стр. 438].
26. Технология программирования – дисциплина, изучающая технологические процессы программирования и порядок их прохождения.

.ЛИТЕРАТУРА

1. Аммерааль Л. STL для программистов на C++. – М. ДМК, 2000. – 240с.
2. Ахо, Альфред, Хопкрофт, Джон, Ульман, Джеффри, Д. Структуры данных и алгоритмы.: Пер. с англ.: уч. Пос. – М.: Издательский дом «Вильямс», 2000. – 384 с.
3. Бадд Т. Объектно-ориентированное программирование в действии/Пер. с англ. - СПб., Питер. -464с.
4. Бланшет Жасмин, Саммерфилд Марк. QT4: программирование GUI на C++. Пер. с англ. – М.: КУДИЦ-ПРЕСС, 2007. – 648с.
5. Вандевурд Д., Джосаттис Н., Шаблоны C++: справочник разработчика. : Пер. с англ. - М. : Издательский дом "Вильямс", 2003. - 544с.
6. Влассидес, Джон. Применение шаблонов проектирования. Дополнительные штрихи.: Пер. с англ. – М.: Издательский дом «Видьямс», 2003.- 144с.
7. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориетированного проектирования. Паттерны проектирования. – СПб.: - Питер, 2001. – 368 с.
8. Гради Буч и др. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2008. – 720с
9. Джосьютис Н., C++ Стандартная библиотека. Для профессионалов – СПб.: Питер, 2004. – 730с.

10. Дьюхэрст Стефан, Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ. – М.: ДМК Пресс, 2006. – 264с.
11. Дьюхерст Стивен. C++. Священные знания. – Пер. с англ. – СПб.: Символ-Плюс, 2007, - 240с.
12. Земсков Ю. В. Qt 4 на примерах. – СПб.: БХВ-Петербург, 2008. – 608с.
13. Каррано Ф. М., Причард Д. Д. Абстракция данных и решение задач на C++. Стены и зеркала. - М.: Вильямс, 2003. - 847с.
14. Коплиен Дж. Программирование на C++. Классика CS. - СПб.: Питер,2005.-479с.
15. Коплиен Дж. Мультипарадигменное проектирование для C++. Библиотека программиста. - СПб. : Питер, - 2005. - 253 с
16. Ларсен, Крег. Применение UML 2.0 и шаблонов проектирования. Практическое руководство. 3-е издание.: Пер. с англ. – М.: ООО «И.Д. Вильямс» 2009. – 736 с.
17. Липпман, Стенли Б., Лажойе, Жоли, Му Барбара Э. Язык программирования C++. Вводный курс, 4-е изд. : Пер. с англ. – М. : ООО «И. Д. Вильямс», 2007. - 896с.
18. Лисков Б. Гатег Дж. Использование абстракций и спецификаций при разработке программ: Пер. с англ. – М.: - М.: Мир, 1989. – 424с.
19. Лишнер Р. C++. Справочник. – СПб.: Питер, 2005. – 907с.
20. Макконнел С. Совершенный класс. Мастер-класс / Пер. с англ. – М.: Издательско-торговый дои «Русская редакция»; СПб.: Питер, 2005. – 806 стр.
21. Мартин, Роберт С. Быстрая разработка программ: принципы, паттерны и практика.: Пер. с англ. – М: Издательский дом «Вильямс», 2004. – 752 с.
22. Мейер Бертран. Объектно-ориентированное конструирование программных систем / Пер. с англ. - М.: Издательско-торговый дом "Русская редакция", 2005. - 1232с.
23. Мейерс С. Наиболее эффективное использование C++. М.: ДМК, 2000. - 296с.
24. Мэйерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ.- М.: ДМК Пресс, 2006. – 300с.
25. Мейерс С. Эффективное использование STL. Библиотека программиста. – СПб.: Питер, 2002. – 224с.
26. Мэйн М, Савитч У. Структуры данных и другие объекты в C++: Пер. с англ. -Издательский дом «Вильямс», 2002 – 832с.
27. Остерн М. Г. Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки C++. Пер. с англ. - СПб.: Невский Диалект, 2004. - 544с.
28. Плаугер П., Степанов А., Ли М., Массер Д. STL – стандартная библиотека шаблонов C++: Пер. с англ. – СПб.: БХВ-Петербург, 2004. – 656с.

29. Прата Стивен. Язык программирования C++. Лекции и упражнения, 5-е изд. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2007. – 1184с.
30. Пышкин Е. В. Основные концепции и механизмы объектно-ориентированного программирования. – СПб.: БХВ-Петербург, 2005. – 640с.
31. Сатер Герб, Александреску Андрей. Стандарты программирования на C++. : Пер. с англ. - М. : Издательский дом "Вильямс", 2005. - 224с.
32. Сатер Герб. Новые сложные задачи на C++. : Пер. с англ. - М. : Издательский дом "Вильямс", 2005. - 272с.
33. Д. Р. Стефенс, К. Диггенс, Д. Турканс, Д. Когсуэлл. C++. Сборник рецептов. Пер. с англ. – М.: КУДИС-ПРЕСС, 2007. – 624с.
34. Страуструп Б. Язык программирования C++, третье издание. - СПб.: БИНОМ, 1999. - 991с.
35. Уайс Марк Ален. Организация структур данных и решение задач на C++.: Пер. с англ. – М: ЭКОМ Паблишерз, 2008. – 896с.
36. Уилсон М, C++: практический подход к решению проблем программирования / Пер. с англ. М.: КУДИЦ-ОБРАЗ, 2006, - 736с.
37. Уилсон М. Расширение библиотеки STL для C++. Наборы и итераторы: Пер. с англ. Слинкина А. А. – М.: ДМК Пресс, СПб. БХВ-Петербург, 2008. – 608с.
38. Чеботарев А. В. Библиотека Qt 4. Создание прикладных приложений в среде Linux. Профессиональная работа. – М.: Издательский дом «Вильямс», 2006, - 256с.
39. Шилдт Г. Искусство программирования на C++. – СПб.: БХВ-Петербург, 2005. – 496с.
40. Шилдт, Герберт. C++: методики программирования Шилдта.: Пер. с англ. – М. : ООО «И.Д. Вильямс», 2009. – 480с
41. Шлее М. Профессиональное программирование на C++.- СПб.: БХВ-Петербург, 2007, 880с.
42. Брюс Эккель, Философия C++, 2-ое изд., Введение в стандартный C++, Питер, 2004, 572с.
43. Брюс Эккель, Чак Эллисон, Философия C++, Практическое программирование, Питер, 2004, 608с.
44. Эллис М., Страуструп Б., Справочное руководство по языку программирования C++ с комментариями, -М.: Мир, 1992, 445с.
45. Эпиенс А., Принципы объектно-ориентированной разработки программ. 2-е изд. : Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 496с.

Замечания

Инстанцирование (англ. *instantiation*) — создание экземпляра класса. В отличие от слова «создание», применяется не к объекту, а к классу. То есть, говорят: *(в виртуальной среде) создать экземпляр класса* или, другими словами, *инстанцировать класс*. Порождающие шаблоны используют полиморфное инстанцирование.