МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет
им. Н.И. Лобачевского»

С.Н. Карпенко

Основы объектно-ориентированного программирования на языке C++

Учебно-методическое пособие

Рекомендовано методической комиссией института ИТММ
для студентов ННГУ, обучающихся по направлениям подготовки
09.03.04. «Программная инженерия» и 02.03.02 «Фундаментальная
информатика и информационные технологии»

Нижний Новгород
2018

УДК 004:655.4/.5(075)
ББК 681.3:Ч617(075)

К 89Карпенко С.Н.. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++: учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2018. – 104с.

Рецензент: д.т.н., профессор **В.Е. Турлапов**

Предлагаемое учебно-методическое пособиеявляется второй частью учебно-методического пособия автора «Основы программирования на языке С», и посвящено основам объектно-ориентированного программирования (ООП) на языке С++. Дается представление о парадигмах ООП с иллюстрацией на конкретном примере, конструкторах, деструкторе и типах конструкторов, механизмах и особенностях перегрузки операций, наследовании и иерархии классов, виртуальных методах и абстрактных классах, шаблонах функций и классов. Материал пособия иллюстрируется большим количеством примеров.

Для студентов первого курса, обучающихся понаправлениям подготовки09.03.04. «Программная инженерия» и 02.03.02 «Фундаментальная информатика и информационные технологии».

УДК 004:655.4/.5(075)
ББК 681.3:Ч617(075)

© Нижегородский государственный университет им. Н.И. Лобачевского, 2018

ВВЕДЕНИЕ

В настоящее время технология объектно-ориентированного программирования является наиболее распростаненной технологией разработки программных систем разных типов. Технология объектно-ориентированного программирования поддерживается практически всеми используемыми языками программирования. Не будет большим преувеличением сказать, что знание и владение этой технологией является обязательными для современного программиста.

Предлагаемое пособие предназначено для студентов первого курса ИТ специальностей и посвящено основам объектно-ориентиованного программирования. Материал пособия излагается на примере языка С++, первого широко распростаненного языка программирования, в котором была использована технология объектно-ориентиованного программирования. В пособии рассматриваются следующие темы: ведение в ООП, классы и объекты, конструкторы и деструктор класса, обработка исключений, перегрузка операций, наследование, виртуальные методы и абстрактные классы, введение в шаблоны.

Пособие составлено на основе материалов лекций, читаемых автором в течение ряда лет.

1. ВВЕДЕНИЕ В ООП

1.1. Парадигмы ООП

1.1.1. Парадигмы программирования

Сначала дадим некоторые определения.

Парадигма (пример, модель, образец) — совокупность фундаментальных научных установок, представлений и терминов, принимаемая и разделяемая научным сообществом и объединяющая большинство его членов [1].

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером [2].

Общая парадигма программирования звучит примерно так: программы состоят из набора программных компонент и набора данных, которыми они (компоненты) оперируют. Такое общее определение парадигмы программирования вытекает из определения программы как формы представления алгоритма, который есть однозначно понимаемая последовательность действий над однозначно понимаемыми объектами (данными). Алгоритм и данные допускают декомпозицию на подалгоритмы и «подданные».

А как организованы эти наборы программных компонент и наборы данных? Как они взаимодействуют? Эти вопросы могут решаться с позицией различных парадигм программирования, часть из которых представлена на рис.1.

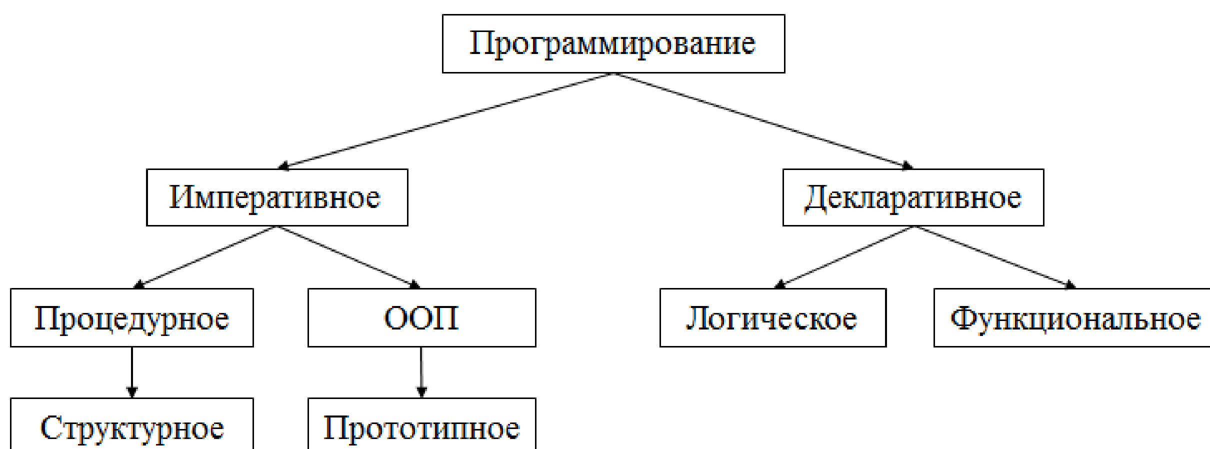

Рис.1. Парадигмы программирования.

Императивное программирование — это парадигма программирования, которая, в отличие от декларативного программирования, описывает процесс вычисления в виде инструкций, изменяющих состояние программы. Императивная программа очень похожа на приказы, выражаемые повелительным наклонением в естественных языках, то есть это последовательность команд, которые должен выполнить компьютер [2].

Процедурное программирование — это парадигма программирования, основанная на концепции вызова процедуры. Процедуры, также известны как подпрограммы, методы или функции (это не математические функции, но функции, подобные тем, которые используются в функциональном программировании). Процедуры просто содержат последовательность шагов для выполнения. В ходе выполнения программы любая процедура может быть вызвана из любой точки, включая саму данную процедуру.

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. В соответствии с данной методологией любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:

- последовательное исполнение — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
- ветвление — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
- цикл — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

Подробнее см. [2].

1.1.2. Парадигмы ООП

Общее определение парадигм ООП

В основе объектно-ориентированного программирования лежат следующие парадигмы:

1. Инкапсуляция:

   a. данные объединяются с алгоритмами в объекты: члены-данные и члены-методы;
   b. данные (поля, свойства) характеризуют свойства объекта, алгоритмы (методы) — его поведение;
   c. сокрытие деталей — свойства и методы делятся на открытые (интерфейс объекта) и скрытые;
2. Наследование
   a. объекты могут наследоваться от других объектов;
   b. при наследовании потомок: сохраняет свойства и методы предка, добавляет новые свойства и методы, может менять старые;
3. Полиморфизм (многообразие форм) — определение поведения объекта контекстом его использования.

Рассмотрим эти парадигмы немого подробнее.

Инкапсуляция

В ООП объекты являются объектами определенных классов. Классы играют роль типов, а объекты — переменных этих типов. Классы (и объекты) объединяют члены-данные (поля) и члены-функции (методы).

Пример:

```
class A          // Класс A
{
int i1;    // Член-данное
void f1();      // Член-метод
void f2();      // Член-метод
};
int main()
{
  A a, b;
  a.f1();
}
```

В приведенном примере:

1. В классе A объявлены член-данное i1 и два метода: F1 и f2.
2. В функции main объявлены два объекта класса A: a и b; выполняется вызов метода f1 объекта a.

Детали реализации класса (поля и методы) могут быть скрыты от клиента класса – функции объявляющей и использующей объекты этого класса.

Сокрытие деталей выполняется с помощью объявления секций private (скрытый) и public (открытый).

Пример:

```
class A
{
private:
int i1;
public:
void f1();
void f2();
};
int main()
{
  A a, b;
  a.f1();
  a.i1 = 10; // ошибка!!
}
```

В приведенном примере:

1. В классе A объявлены член-данное i1 объявлено в секции private (закрытое), а методы: f1 и f2 – как отрытые.
2. Соответственно в клиенте (функции main) вызываемый через a метод f1 доступен, а поле i1 недоступно.

Наследование

Классы могут объявляться как наследники других, ранее объявленных классов. При наследовании потомок: сохраняет свойства и методы предка, может добавлять новые свойства и методы, может менять старые.

Пример:

```
class A
{
int i1;
void f1();
void f2();
};
class B : public A // Наследник
{
int i2;
void f1();      // Замещение
void f3();
};
int main()
{
  A a;
  B b;
```

```
  a.i1 = 1;
  a.f1();
  a.f2();
  b.i1 = 2;
  b.i2 = 3;
b.f1();
  b.f2();
  b.f3();
}
```

В приведенном примере:

1. Класс B объявлен как наследник класса A.
2. Класс B:
    a. наследует поле i1 и метод f2 класса A:
    b. в нем объявлены «свои» поле i2 и метод f3;
    c. в нем перекрыт (замещен) метод f1 класса A.
3. В клиенте классов A и B объявлены объекты этих классов, которым доступны поля и методы, описанные в примере. При этом, вызов a.f1() означает вызов метода f1 класса A, а вызов b.f1() – вызов метода f1 класса B.

Полиморфизм

Полиморфизм (зависимость поведения объекта от контекста его использования) проиллюстрируем на следующем примере:

```
class A
{
void f1();
virtualvoid f4();
//...
};

class B : public A
{
void f1();
virtualvoid f4();
//...
};
void f(A& obj)
{
  obj.f1(); obj.f4();
}
int main()
{
  B b;
```

```
  f(a);
  f(b);
}
```

В приведенном примере:

1. Класс B объявлен как наследник класса A.
2. В классе B перекрыты оба объявленных в классе A метода f1 и f4. Метод f1 перекрыт обычным образом, метод f4 – как виртуальный (см. ниже).
3. Функция f получает ссылку на объект класса A и обращается к методам f1 и f4 этого объекта.
4. В функции main объявляются объекты a и b и выполняется вызов функции f с передачей ей объекта a и потом – объекта b.
5. При первом вызове функции f будут вызваны методы f1 и f4 класса A.
6. При втором вызове функции f, когда ей вместо объекта A «подсунули» объект B, будут вызваны: метод f1 класса A и метод f4 класса B.

1.2.Как работают парадигмы ООП

1.2.1. Постановка задачи

Работу парадигм ООП более детально рассмотрим на примере следующей задачи.

Написать программу - графический редактор, которая умеет:

1. Составлять рисунок из фигур: точек, прямых линий. треугольников, …
2. Добавлять, удалять, перемещать, изменять размеры фигур рисунка.
3. Отрисовывать (показывать) рисунок.

Для упрощения ситуации будем рассматривать только три отмеченные типа фигур и только функцию показа рисунка.

Для решения поставленной задачи нам надо:

1. Выбрать представление рисунка как набора геометрических фигур.
2. Написать функцию рисования рисунка.

1.2.2. Процедурное решение задачи

Процедурное решение задачи может выглядеть следующим образом.

Представление рисунка

Представление рисунка включает:

1. Объявления геометрических фигур.
2. Описание обобщенной фигуры.
3. Описание рисунка как массива обобщенных фигур.

Объявления геометрических фигур

Объявления геометрических фигур может выглядеть так:

```
struct Tpoint
{
int x, y;   // Координатыточки
};
struct Tline
{
int x1, y1; // Координаты 1-ойточки
int x2, y2; // Координаты 2-ой точки
};
struct Ttriangle
{
int x1, y1; // Координаты 1-ой вершины
int x2, y2; // Координаты 2-ой вершины
int x3, y3; // Координаты 3-ой вершины
};
```

Надеемся, что в комментариях это не нуждается.

Описание обобщенной фигуры

Для описания обобщенной фигуры выберем конструкцию:

```
enum Tfigtype{point, line, triangle};// Типфигуры
struct Tfigure      // Фигура
{
  Tfigtype figtype; // типфигуры
void *figure;      // указатель на описание фигуры
};
```

Здесь объявляется перечисленный тип `Tfigtype`, который принимает три значения (точка, линия, треугольник) и структура `Tfigure`, представляющая обобщенную фигуру. В этой структуре объявляется:

- поле `figtype`, которое в каждом конкретном случае будет принимать значение точка, линия или треугольник;
- поле `figure`, которое в каждом конкретном случае будет принимать значение указателя на объект соответствующей структуры геометрической фигуры;

Описание рисунка как массива обобщенных фигур

Для описания всего рисунка выберем конструкцию:

```
const FIGMAX = 500;       // максим. к-во фигур
int figcount;  // реальное к-во фигур
Tfigure figlist[FIGMAX]; // массив фигур
```

Схема представления рисунка представлена на рис.2.

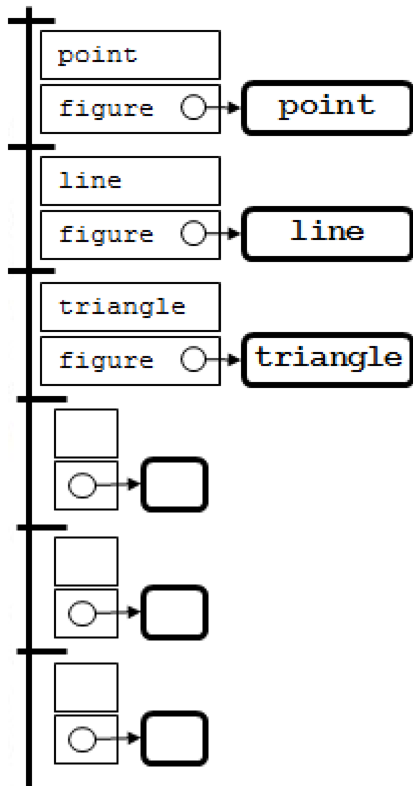

Рис.2. Схема представления рисунка.

Отрисовка рисунка

Для отрисовки рисунка нам надо будет написать функции рисования геометрических фигур, прототипы которых будут выглядеть так:

```
// Процедурырисованияфигур ----------------------
void showpoint    (int x , int y);
void showline     (int x1, int y1, int x2, int y2);
void showtriangle(int x1, int y1, int x2, int y2, int x3, int y3);
```

Детали работы этих функций нас сейчас не интересуют.

И наконец, общая функция рисования рисунка будет выглядеть примерно так:

```
void showpicture(Tfigure figlist[] , int figcount)
{
  Tpoint    *point;     // рабочиепеременные
  Tline     *line;
  Ttriangle *triangle;
for ( int i = 0; i < figcount; i++ )
  {
switch (figlist[i].figtype)
     {
case point: point = (Tpoint*)(figlist[i].figure);
                  showpoint(point->x, point->y);   break;
case line:  line = (Tline*)(figlist[i].figure);
                  showline(line->x1, line->y1, line->x2, line-
>y2);
break;
case triangle: triangle = (Ttriangle*)(figlist[i].figure);
                     showtriangle(triangle->x1, triangle->y1,
                                  triangle->x2, triangle->y2,
                                  triangle->x3, triangle->y3)
break;
    }
}
```

Общая схема работы этой функции такова:

1. Функция принимает массив фигур `figlist` и количество элементов этого массива `figcount`.
2. Объявляются три рабочих указателя `point,line, triangle`.
3. Далее в цикле для каждого элемента массива фигур, в зависимости от его типа указатель на фигуру приводится к нужному рабочему указателю и вызывается нужная функция отрисовки фигуры соответствующего типа.

1.2.3. Решение в рамках ООП

Далее мы поэтапно рассмотрим, что дает применение парадигм ООП для упрощения решения рассматриваемой задачи.

Инкапсуляция

Представление рисунка

В представлении рисунка у нас изменится объявления геометрических фигур:

```
class Cpoint
{
```

```
int x, y;   // Координаты точки
void show();
}
class Cline
{
int x1, y1; // Координаты 1-ойточки
int x2, y2; // Координаты 2-ой точки
void show();
}
class Ctriangle
{
int x1, y1; // Координаты 1-ой вершины
int x2, y2; // Координаты 2-ой вершины
int x3, y3; // Координаты 3-ой вершины
void show();
}
```

Главное в этом новом варианте состоит в том, что объект каждого класса содержит координаты своей фигуры и метод ее рисования, которому уже не надо передавать координаты, которые он «знает» сам – детали рисования (координаты) скрыты в объекте.

Остальные элементы представления рисунка остаются пока без изменения.

Отрисовка рисунка

В отрисовке рисунка функций рисования отдельных фигур уже нет – они ушли в определения классов.

Общая функция отрисовки рисунка будет выглядеть уже так:

```
void showpicture(Tfigure figlist[] , int figcount)
{
  for ( int i = 0; i < figcount; i++ )
  {
switch (figlist[i].figtype)
    {
case point:    (Cpoint*)    (figlist[i].figure)->show();   break;
case line:     (Cline*)     (figlist[i].figure)->show();   break;
case triangle: (Ctriangle*) (figlist[i].figure)->show();   break;
}
  }
}
```

Наследование и полиморфизм

Использование наследования и полиморфизма позволить существенно упростить как представление рисунка, так и функцию его рисования.

Представление рисунка

Объявления фигур будет выглядеть так:

```
// Объявленияфигур -------------------------
class Cfigure  {
virtualvoid show() = 0;
}
class Cpoint : public Cfigure {
int x, y;   // Координатыточки
virtualvoid show();
}
class Cline : public Cpoint  {
int x2, y2; // Координаты 2-ой точки
virtualvoid show();
protected:
void showline(int x1, int y1, int x2, int y2);
}
class Ctriangle : public Cline {
int x3, y3; // Координаты 3-ойвершины
virtualvoid show();
}
```

Здесь сначала объявляется класс `Cfigure` , в котором объявлен виртуальный метод `show`. Все остальные фигуры последовательно наследуются друг от друга с виртуальным перекрытием метода `show`.

Единый предок всех классов позволяет упростить общее представление рисунка:

```
const FIGMAX = 500;       // максим. к-во фигур
int figcount;   // реальное к-во фигур
Cfigure* figlist[FIGMAX]; // массив фигур
```

Здесь это массив указателей на общего предка всех фигур.

Отрисовка рисунка

Ну и наконец, совсем просто будет выглядеть функция отрисовки рисунка:

```
void showpicture(Tfigure figlist[] , int figcount)
{
for ( int i = 0; i < figcount; i++ )
{
     figlist[i].figure->show();
  }
}
```

Вся сложность управления выбором типа рисунка здесь спрятана в механизм полиморфизма: выбора виртуального метода нужного класса.

2. КЛАССЫ И ОБЪЕКТЫ

2.1. Класс: объявление и описание

2.1.1. Объявление класса

Итак, в соответствии с парадигмой инкапсуляции класс объединяет данные класса и функции класса. Элементы данных класса принято называть полями или членами-данными. Функции класса принято называть методами или членами-функциями.

Синтаксис:

```
class<имя_класса>
{
<объявления полей>
<объявления методов>
}; // заканчивается ';' !!!!
```

где:

- `<имя_класса>` - имя класса;
- `<объявления полей>` - объявления полей класса;
- `<объявления методов>` - объявления методов класса.

Поля объявляются аналогично объявлению переменных; методы – в виде прототипа функций.

Пример:

```
class Ccomplex
{
// Поля: действительная и мнимая части комплексного числа
double re, im;
// методы:
  Ccomplex Add(const Ccomplex& cmp); // Сложение комплексных чисел
  Ccomplex Mlt(const Ccomplex& cmp); // Умножение комплексных 
чисел
  . . . . . . . . . . . . . . . . . .
};
```

В примере приведен фрагмент объявления класса `Ccomplex` - комплексное число. В качестве полей объявлены действительная и мнимая части комплексного числа. В качестве методов объявлены прототипы функций сложения и умножения комплексных чисел.

При объявлении класса действуют следующие правила:

1. Поля класса могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на объект этого класса).
2. Инициализация полей класса при их объявлении не допускается.

2.1.2. Описание класса

Описание класса состоит из описаний методов класса. Синтаксис описания метода класса имеет вид:

```
<тип><имя_класса>::<имя_метода>(<параметры>)
{
<описание алгоритма>
}
```

где:

- <тип> - тип возвращаемых методом данных;
- <имя_класса> - имя класса метода;
- <имя_метода> - имя класса метода;
- <параметры> - список формальных параметров метода;
- <описание алгоритма> - описание алгоритма метода;

Нетрудно видеть, что описание метода отличается от описания функции только указанием на то, какому классу принадлежит описываемый метод.

Пример:

```
Ccomplex Ccomplex::Add(constCcomplex& cmp)
{
Ccomplex rez;
  rez.re = re + cmp.re;
  rez.im = im + cmp.im;
return rez;
}
```

В примере представлено описание метода сложения комплексных чисел класса Ccomplex. Правила описания методов класса будут представлены ниже.

Хотя объявление и описание класса могут находиться в одном файле, настоятельно рекомендуется:

1. Объявление класса записывать в заголовочный файл; описание – в исполняемый.
2. Каждый класс описывать в своих заголовочном и исполняемом файлах, имя которого совпадает с именем класса

ccomlpex.h

```
class Ccomplex
{
  // поля:
  double re, im;
  // методы:
  Ccomplex Add(Ccomplex cmp);
  Ccomplex Mlt(Ccomplex cmp);
  . . . . . . . . . . . . . . .
};
```

ccomlpex.cpp

```
#include "Ccomplex.h"
Ccomplex Ccomplex::Add(const Ccomplex& cmp)
{
  Ccomplex rez;
  res.re = re + cmp.re;
  res.im = im + cmp.im;
  return rez;
}
. . . . . . . . . . . . . . . . .
```

Рис.3. Заголовочный и исполняемый файлы класса Ccomlex.

Метод может быть описан в объявлении класса:

```
classCcomplex
{
double re, im;
void setre(double _re) { re = _re;  }
double getre()         { return re; }
    // . . . . . . . . . . . . . .
};
```

Злоупотреблять этим, однако, не следует:

- при больших размерах тела метода ухудшается читаемость кода;
- методы, описанные в объявлении класса по умолчанию имеют спецификатор online.

2.2. Объект: объявление и использование

2.2.1. Объявления объектов

Класс является типом данных. Объявление и описание класса это объявление нового типа данных.

Переменные типа «класс» называются объектами класса или экземплярами класса. Объявление объекта аналогично объявлению переменной базового типа, где вместо имени типа указывается имя класса:

```
<имя_класса><имя_объекта>;
```

Объявление указателя на объект и динамическое создание объекта:

```
<имя_класса>* <имя_указателя_на_объект> = new <имя_класса>;
```

Пример:

```
Ccomplex A, B;
Ccomplex* D = new Complex;
```

Доступ к полям и методам – как в структурах:

```
A.re = 3.56;
A = B.Add(*D);
D->im = -2.0;
B = D->Add(A)
```

При объявлении объектов их поля могут быть инициированы аналогично инициированию структур:

```
Ccomplex A = {1.5, 2.8}, B = {-0.7, 9.2};
```

2.2.2. Взаимодействие объектов и методов

Создание объектов при объявлении

Пусть клиентом класса CComplex являтся следующая функция main:

```
int main() {
  Ccomplex A = {1.5, 0.3 };
static Ccomplex B;
  Ccomplex* D = new Complex;
  B.re = 3.14;
D->im = -2.;
  . . . . . . . . . . . .
}
```

Размещение объектов примера в памяти компьютера представлено на рис.4. Вне зависимости от количества и типов памяти объявленных объектов, размещенные в программном сегменте методы класса представлены в единственном экземпляре.

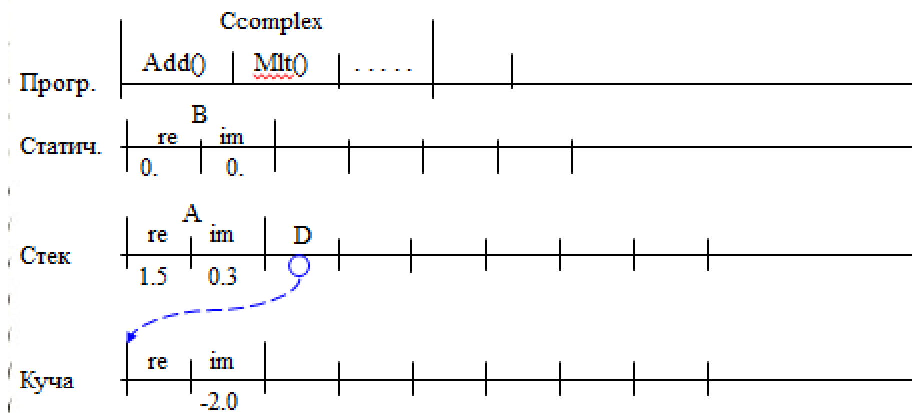

Рис.4. Размещение в памяти объектов примера.

Взаимодействие объектов и методов

В такой ситуации возникает вопрос: как объекты видят «свои» методы? В ООП действуют следующие правила взаимодействия объектов и методов.

1. Доступ к методам класса возможен только через объекты класса:

```
Ccomplex A, B;
A.Add(B);
Add(B);   // Ошибка
```

2. Каждый метод класса неявно получает параметр this – указатель на тот объект класса, через который он был вызван:

```
Ccomplex Ccomplex::Add(Ccomplex cmp[, Ccomplex* this])
{
Ccomplex rez;
  res.re = this->re + cmp.re;
res.im = im + cmp.im;
return rez;
}
```

3. При описании метода в ссылках на поля объекта, через который был вызван метод косвенную адресацию `this->` можно не указывать.

2.3. Сокрытие членов класса

2.3.1. Секции private и public

При объявлении класса его члены (поля, методы) могут быть скрыты или открыты с помощью спецификаторов доступа private и public:

```
class<имя_класса>
{
public:
<объявления открытых полей и методов>
private:
<объявления скрытых полей и методов >
};
```

При этом действуют следующие правила.

1. Если спецификатор доступа не указан, то по умолчанию считается private.
2. Секций public и private может быть несколько и они могут чередоваться в произвольном порядке.
3. Скрытые члены класса (private) не доступны через объекты этого класса. Они доступны только методам класса.
4. Открытые члены (public) составляют интерфейс класса и «видны» его объектам.
5. Хотя бы один член класса должен быть открыт!!!

Пример:

```
class Ccomplex
{
private:
// поля: действительная и мнимая части
double re, im;
public:
// методы:
    Ccomplex Add(const Ccomplex& cmp); // Сложение
    Ccomplex Mlt(const Ccomplex& cmp); // Умножение
  . . . . . . . . . . . . . . . . . .
};
int main() {
  Ccomplex A, B;
  Ccomplex* D = new Complex;
  B.re = 3.14; // Ошибка
A = D->Add(B);
  . . . . . . . . . . .
}
```

В приведенном примере закрытыми объявлены поля класса Ccomplex, открытыми – методы. Соответственно в функции main ссылка на поле re объекта B является ошибкой.

2.3.2. Что и как скрывать в объявлении класса

Сокрытие методов класса

В отношении сокрытия методов класса целесообразны следующие рекомендации:

1. Скрывать надо методы класса не существенные для клиента и описывающие детали реализации функциональности класса.
2. Открытыми оставляют «функциональные» методы, описывающие внешнее поведение класса.
3. Открытых методов должно быть не более 5-10 (за исключением некоторый специальных методов). Если их больше, то надо попробовать разбить класс на два класса.

Сокрытие полей класса

Поля класса принято скрывать для их защита полей от случайной «порчи» клиентом класса. В случае сокрытия полей класса для доступа к полям класса пишут специальные методы.

Пример:

```
classCcomplex
{
private:
double re, im;
public:
void setre(double _re);
double getre();
void setim(double _im);
double getim();

double& Re();
double& Im();

. . . . . . . . . . . . . . . . . .
};
voidCcomplex::setre(double _re) { re = _re;  }
doubleCcomplex::getre()          { return re; }
voidCcomplex::setim(double _im) { im = _im;  }
doubleCcomplex::getim()          { return im; }
double&Ccomplex::Re()             { return re; }
double&Ccomplex::Im()             { return im; }

int main() {
  Ccomplex A, B;
B.re = 3.14;    // Ошибка !!
```

```
  B.setre(3.14); // Правильно
double dd = B.Re();
  B.Im() = 5.9;
}
```

В приведенном примере:

1. Поля re и im класса Ccomplex объявлены как скрытые, открытыми объявлены специальные методы доступа к полям.
2. Методы доступа к полям делятся на две группы:
   a. методы типа set<имя_поля> и get<имя_поля>;
   b. методы типа <имя_поля>.
3. Метод типа set<имя_поля> получает значение поля и присваивает его полю, метод типа get<имя_поля> возвращает значение поля.
4. Метод типа <имя_поля> (Re, Im) возвращает ссылку на поле, которая может быть использована как в правой, так и в левой части операции присваивания.
5. На практике могут использоваться как обе группы методов доступа к полям, так и одна из них.

2.3.3. Описание методов в объявлении класса

Методы класса можно описывать в объявлении класса:

```
classCcomplex
{
private:
double re, im;
public:
void setre(double _re) { re = _re;   }
double getre()         { return re;  }
void setim(double _im) { im = _im;   }
double getim()         { return im;  }
double& Re()         { return re; }
double& Im()         { return im; }

//. . . . . . . . . . . . . . . . . .
};
```

Преимущество такого описания методов состоит в том, что такие методы компилируются как inline функции, т.е. результат их компиляции «вкладывается» в место вызова.

2.3.4. Классы и структуры

В языке C++ структура (struct) также является классом, в котором наряду с полями-данными могут быть объявлены и описаны поля-методы.

Единственное отличие состоит в том, что спецификаций доступа по умолчанию в классе является private, а в структуре public.

Кроме того:

1. При объявлении объекты класса (и структуры) могут инициироваться только если все поля данные открыты.
2. Для класса операция объект = объект срабатывает как для структур (копирование полей данных).

2.4. Примеры

2.4.1. Класс Ccomplex

Немного более подробное объявление класса Ccomplex приведено ниже.

```
Файл Ccomplex.h ------------------------------------------------
classCcomplex
{
private:
double re, im;
public:
void setre(double _re) { re = _re;   }
double getre()         { return re; }
void setim(double _im) { im = _im;   }
double getim()         { return im; }

double& Re()   { return re; }
double& Im()   { return im; }

    Ccomplex Add(const Ccomplex& cmp);    // a  = b + c
    Ccomplex AddEq(const Ccomplex& cmp); // a += b
//. . . . . . . . . . . . . . . . . .
};
```

Здесь добавлены два метода Add и AddEq, являющиеся аналогами операций сложения двух комплексных чисел и увеличения комплексного числа.

Описание этих методов выглядит так:

```
Файл Ccomplex.cpp ----------------------------------------------
#include"Ccomplex.h"
```

```
// Аналог a = b + c ------------------------------
Ccomplex Ccomplex::Add(const Ccomplex& cmp)
{
  Ccomplex rez;
  rez.re = re + cmp.re;
  rez.im = im + cmp.im;
return rez;
}
// Аналог a += b ----------------------------------
Ccomplex Ccomplex:: AddEq(const Ccomplex& cmp)
{
  re += cmp.re;
  im += cmp.im;
return *this;
}
```

Вметоде Add:

1. Объявляется rez – результат выполнения метода.
2. Формируется действительная и мнимая части rez.
3. Возвращается значение rez.

В методе AddEq:

4. Действительная и мнимая части «себя» увеличиваются на действительную и мнимую части cmp.
5. Возвращается значение «себя».

Клиент приведенного примера класса Ccomplex может быть таким:

```
Файл test.cpp -------------------------------------------------
#include"Ccomplex.h"
int _tmain(int argc, _TCHAR* argv[])
{
  Ccomplex A, B, C;
  Ccomplex* D = new Ccomplex;
  A.setre(1.);   A.setim(2.);
  B.Re() = 3.;   B.Re() = 4.;
  B.Im() = A.Re() + 3.;
  D->Re() = 5.;  D->Im() = 6.;

B = A.AddEq(*D); // А это как?
  C = A.Add(B);    // И это как?
  A = B;
return 0;
}
```

2.4.2. Класс Cmystring

Мы будем рассматривать следующее объявление класса `Cmystring`:

```
ФайлCmystring.h ---------------------------------------------
class Cmystring
{
private:
char* str;   // строка
int size;    // размер (max)
int leng;    // длина реальная
public:
int StrLen() { return size; }   // запрос длины строки
void Create(int _size);         // создание строки
voidCreate(constchar* cstr); // созданиестроки
voidDestroy();                 // удалениестроки
voidStrCopy(constchar*);      // копированиестроки
voidStrCopy(constCmystring&); // копированиестроки
void Print();                    // вывод строки на экран
Cmystring AddSt(const Cmystring&);// объединениестрок
private:
void Rebuffer(int newSize, int sizeDelta = 10);
void Resize(int newSize, int sizeDelta = 10);
void _strcopy(char* s1, char* s2);
};
```

В этом объявлении

1. Объект строка представлена тремя полями:
    a. `str` – указатель на массив типа char, в котором будет храниться строка;
    b. `size` – размер этого массива (размер буфера);
    c. `leng` – длина строки (количество «правильных» символов).
2. Поля представления строки скрыты, но стандартных методов типа set, get не предусмотрено потому, что в этом случае клиент класса не должен иметь доступа к этим полям потому, что:
    a. всю необходимую функциональность он получит через методы класса;
    b. полями класса управляет сам класс, обеспечивая скрытый от клиента контроль переполнения буфера.
3. Объявлен и описан метод `StrLen` запроса длины строки.
4. Объявлены два метода Create, создающие строку по заданной длине и по заданному содержанию строки. Эти методы заказывают в куче память нужной длины и формируют значения полей str, size и leng.

5. Объявлен метод `Destroy`, который будет освобождать занятую объектом память в куче.
6. Объявлены два метода копирования строки: из указателя на char и из другого объекта `Cmystring`.
7. В дополнение объявлены методы вывода строки на консоль и объединения двух строк.
8. И наконец, объявлены три скрытых метода, смысл которых объясним ниже.

Исполняемый файл примера выглядит так:

```
ФайлCmystring.cpp --------------------------------------------
#include"Cmystring.h"
// созданиестроки -----------------------------
void Cmystring::Create(int _size)
{
  str = newchar[size]; str[0] = 0;
size = _size;
  leng = 0;
}
// созданиестроки -----------------------------
void Cmystring::Create(constchar* cstr)
{
  leng = 0;
while ( cstr[leng++] );
  size = leng + 1;
  str = newchar[size];
int i = 0;
while (str[i] = cstr[i]) i++;
}
// удалениестроки ----------------------------
void Cmystring::Destroy()
{
if ( size )
delete[] str;
  str = 0;
  leng = 0;
  size = 0;
}
// копирование --------------------------------
void Cmystring::StrCopy(constchar* cstr)
{
int len = 0;
while (cstr[len]) len++;
if ( len > size )
    Rebuffer(len);
  _strcopy(str, cstr);
```

```
  leng = len;
}
// копирование ---------------------------------
void Cmystring::StrCopy(const Cmystring _str)
{
if (_str.Leng() > leng )
    Rebuffer(_str.Leng());
  _strcopy(str, _str.str);
  leng = _str.Leng();
}
// объединениестрок --------------------------
Cmystring Cmystring::AddSt(const Cmystring& _str)
{
  Cmystring tmp;
  tmp.Create(leng + _str.leng + 1);
  _strcopy(tmp.str,str); // копируемсебя
  _strcopy(tmp.str + leng,_str.str); // копируем _str
  tmp.leng = leng + _str.leng;
  tmp.size = leng + _str.leng + 1;
return tmp;
}
// увеличениедлиныбуфера
void Cmystring::Rebuffer(int newSize, int sizeDelta = 10)
{
delete[] str;
  str = newchar[newSize + sizeDelta];
  size = newSize + sizeDelta;
  str[0] = '\0';
  leng = 0;
}
// увеличениедлиныстроки
void Cmystring::Resize(int newSize, int sizeDelta = 10)
{
char* tmp = newchar[newSize + sizeDelta];
  _strcopy(tmp, str);

  size = newSize + sizeDelta;
delete[] str;
}
// копированиестроки
void Cmystring::_strcopy(char* s1, constchar* s2)
{
do
  {
    *s1++ = *s2;
  }
while ( *s2++ );
}
```

В этом файле:

1. В методе `Create(int _size)` в куче заказывается память под буфер строки и формируются значения полей класса.
2. В методе `Create(constchar* cstr)` определяется длина строки cstr, в куче заказывается память под буфер строки класса, формируются значения полей класса и строка cstr копируется в буфер.
3. В методе освобождается память, занимаемая буфером в куче и обнуляются поля класса.
4. Вметоде`StrCopy(constchar* cstr)`:
    a. определяется len - длина строки, которую надо скопировать в объект;
    b. выполняется проверка, не больше ли она размера буфера объекта, и если больше, то вызывается скрытый метод `Rebuffer` , который выполняет увеличение размера буфера до указанной длины.
    c. выполняется копирование строки cstr в буфер. Поскольку копирование строе будет выполняться из многих мест кода, целесообразно ввести скрытую функцию `_strcopy` , выполняющую это копирование.
5. Метод `StrCopy(const Cmystring _str)` работает аналогично методу `StrCopy(constchar* cstr)` с учетом различий в источнике копируемой строки.
6. Метод `AddSt` выполняет объединение двух строк: «себя» и строки, полученной в качестве параметра. Для этого:
    a. объявляется рабочий объект tmp, в которай будут «сливаться» объединяемые строки;
    b. в этом объекте создается буфер нужной длины;
    c. с помощью функции `_strcopy` в него копируется «своя» строка, а затем - `_str`.
    d. формируются значения полей объекта tmp;
    e. объект tmp возвращается в качестве результата.

3. КОНСТРУКТОРЫ И ДЕСТРУКТОР КЛАССА

3.1. Введение

В объявлении класса его поля не могут быть инициализированы. Поля должны инициализироваться при объявлении объектов.

Особую роль играют поля – указатели, под которые надо заказывать динамическую память. При удалении объекта эту память надо освобождать.

Инициализацию полей объекта (и заказ памяти под динамические поля) при объявлении объекта выполняет специальный метод класса конструктор. Освобождение памяти динамических полей объекта при его удалении – деструктор.

3.2. Что такое конструктор?

Конструктор - специальный метод класса, имеющий следующие особенности:

1. Имя метода-конструктора совпадает с именем класса.
2. Конструктор ничего не возвращает, даже типа void.
3. Параметры конструктора могут иметь любой тип. Можно задавать значения параметров по умолчанию.
4. Конструкторы класса вызываются неявно (автоматически) при объявлении объектов класса и в ряде других случаев.
5. Конструкторов может быть несколько с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).

Функции конструктора: инициализация полей создаваемого объекта и запрос памяти под динамические поля объекта.

Синтаксис объявления конструктора:

<имя_класса>(<список_параметров>);

Конструкторы:

- не наследуются;
- не могут вызваться явно;
- не могут быть объявлены с модификаторами const, static и virtual;
- нельзя получить указатель на конструктор;

Пример:

```
class Cmystring
{
private:
     char* str;   // строка
     int size;    // размер (max)
     int leng;    // длина реальная
public:
       . . . . . . . . . . . . . . . . . . . .
       Cmystring(int _leng, char fill);
       . . . . . . . . . . . . . . . . . . . .
};

Cmystring::Cmystring(int _leng, char fill)
{
     leng = _leng;  size = _leng + 1;
     str = newchar[size];
     for ( int i = 0; i < leng; i++ ) str[i] = fill;
     str[leng] = 0;
}

int main()
{
     Cmystring s1(10, ' '), s2(15, '#');
     . . . . . . . . . . . . . . . . .
     return 0;
}
```

В приведенном примере в классе Cmystring объявлен и описан конструктор, получающий длину строки leng и символ ее начального заполнения fill. При объявлении объектов s1 и s2 в функции main будет автоматически вызван этот конструктор.

3.3. Типы конструкторов

Каждый класс может иметь несколько конструкторов. Все конструкторы класса относятся к одному из следующих четырех типов:

1. Конструктор по умолчанию – без параметров. Конструктор по умолчанию используется для создания «пустого» объекта.
2. Конструктор копирования – имеет один параметр типа константная ссылка на объект того же класса. Конструктор копирования используется для создания копии объекта. Конструктор копирования необходим для вызова параметра-объекта по значению.
3. Конструктор преобразования типа – имеет один параметр любого типа. Конструктор преобразования типа используется для формирования объекта класса из объекта другого типа. Конструктор преобразования типа описывает задаваемое пользователем преобразование типа и будет

неявно использоваться при вызове перегруженных функций (третье правило сигнатуры).

4. Конструктор инициализатор – имеет более одного параметра. Конструктор инициализатор используется для формирования объекта класса по задаваемым конструктору параметрам. Конструктор Cmystring(int _leng, char fill); описанный в приведенном выше примере является конструктором инициализатором.

Для конструкторов класса справедливы следующие правила:

1. Каждый класс может иметь по одному конструктору по умолчанию и копирования и по несколько конструкторов инициализаторов и преобразования типа.
2. Все конструкторы класса, не являющиеся конструкторами первых трех типов являются конструкторами инициализаторами.
3. Все конструкторы класса, имеющие один параметр, отличный от ссылки на объект того же класса, являются конструкторами преобразования типа.
4. В некоторых случаях возникает необходимость написания конструкторов инициализаторов с одним параметром. Формально такой конструктор является конструктором преобразования типа и может быть неявно использован компилятором в приведении типов при вызове перегруженных функций. Для предотвращения такой возможности конструктор с одним параметром может быть объявлен со спецификатором explicit

Пример:

```
class Cmystring
{
private:
      char* str;   // строка
      int size;    // размер (max)
      int leng;    // длина реальная
public:
      // Конструкторы ---------------------------------
        Cmystring(void);                         // По умолчанию
      Cmystring(constCmystring&cms);         // Копирования
      Cmystring(constchar* cstr); // Преобразованиятипа
      Cmystring(double num);        // Преобразованиятипа
        Cmystring(int _leng, char fill = ' ');  // Инициализатор 1
      explicit Cmystring(int _leng);           // Инициализатор 2
        . . . . . . . . . . . . . . . . . . . . 
};

// Конструкторы ---------------------------------
Cmystring::Cmystring(void) // Поумолчанию
{
      leng = 0;  size = 0;  str  = 0;
}
Cmystring::Cmystring(const Cmystring& cms) // Копирования
{
```

```
	size = cms.size;
	leng = cms.leng;
	str = newchar[size];
	for (int i = 0; i < size; i++)
		str[i] = cms.str[i];
}
Cmystring::Cmystring(constchar* cstr)    // Преобразованиятипа
{
	leng = 0;
	while ( cstr[leng] ) leng++;
	size = leng + 1;
	str = newchar[size];
	for (int i = 0; i < size; i++)
		str[i] = cstr[i];
}
Cmystring::Cmystring(double num)  // Преобразованиятипа
{
	// преобразование num в str ------------------------
	. . . . . . . . . . . . . . . . . . .
}
Cmystring::Cmystring(int _leng, char fill) // Инициализатор 1
{
	leng = _leng;  size = _leng + 1;
	str = newchar[size];
	for ( int i = 0; i < leng; i++ ) str[i] = fill;
	str[leng] = 0;
}
Cmystring::Cmystring(int _leng) // Инициализатор 2
{
	leng = _leng;  size = _leng + 1;
	str = newchar[size];
	str[leng] = 0;
}

intmain()
{
	Cmystrings1;            // конструкторпоумолчанию
	Cmystring s2("Москва"); // конструктор преобразования типа
	Cmystring s4(9.14);     // конструктор преобразования типа
	Cmystring s5(10, '#');  // конструктор инициализатор 1
	Cmystring s6(15);       // конструктор инициализатор 2
	Cmystring s7(s2);       // конструктор копирования
	. . . . . . . . . . . . . . . . .
	return 0;
}
```

В приведенном примере в классе Cmystring объявлены и описаны два конструктора инициализатора. Первый – с заполнителем строки fill, второй – без заполнителя. Второй конструктор формально является конструктором преобразования типа, и это его свойство блокировано спецификатором explicit в объявлении этого конструктора.

3.4. Деструктор класса

Деструктор - специальный метод класса, имеющий следующие особенности:

1. Имя метода-деструктора: ~<имя класса>.
2. Деструктор не имеет параметров и ничего не возвращает, даже типа void.
3. Деструктор класса вызываются неявно (автоматически) при удалении объектов класса.

Функции деструктора:

- освобождение памяти, занимаемой динамическими полями объекта;
- любые завершающие действия, которые необходимо выполнить вместе с удалением объекта (например, скрытие геометрической фигуры на экране).
- Синтаксис объявления деструктора:

  ~<имя_класса>(void);

Деструкторы:

- не наследуются;
- не рекомендуется вызвать явно;
- не могут быть объявлены с модификаторами const, static (virtual - могут);
- нельзя получить указатель на деструктор.

Пример:

```
class Cmystring
{
private:
      char* str;  // строка
      int size;   // размер (max)
      int leng;   // длина реальная
public:
      // Конструкторы --------------------------------
        . . . . . . . . . . . . . . . . . . . . . .
      // Деструктор ----------------------------------
        ~Cmystring(void);
      . . . . . . . . . . . . . . . . . . . . . .
};

// Деструктор -----------------------------------
Cmystring::~Cmystring(void)
{
      if ( size )
            delete[] str;
      leng = 0;
      size = 0;
      str  = 0;
}
```

3.5. Когда нужны конструкторы и деструктор

При написании класса можно явно не описывать его конструкторы и деструктор. Если в классе отсутствуют явно описанные конструкторы и деструктор, то компилятор неявно (автоматически) создает в этом классе конструктор по умолчанию, конструктор копирования и деструктор. При этом:

- созданные неявно конструктор по умолчанию и деструктор ничего не делают (имеют пустое тело);
- созданный неявно конструктор копирования просто присваивает полям создаваемого объекта значения соответствующих полей копируемого объекта.
- Использование неявно созданных конструкторов и деструктора нежелательно вообще и просто недопустимо в случае, когда класс имеет динамические поля.

3.6. Когда и как вызываются конструкторы и деструктор

К сожалению, часто встречающимся заблуждением являются мнения о том, что конструкторы вызываются для выделения памяти под объект, а деструктор – для освобождения памяти, занимаемой объектом. Конструктор должен выделять, а деструктор освобождать <u>память, занимаемую динамическими полями объекта</u> (а не самого объекта). Вызов конструкторов и деструктора определяются следующими правилами:

Конструктор вызывается неявно (автоматически) <u>в начале времени жизни объекта</u> (сразу после выделения памяти под объект). Тип вызываемого конструктора определяется контекстом объявления объекта.

4. Деструктор вызывается неявно (автоматически) <u>в момент прекращения времени жизни объекта</u> (перед освобождением памяти объекта).
5. Каждому вызову конструктора объекта соответствует вызов деструктора этого объекта

В соответствии с различными механизмами выделения памяти можно привести следующие случаи вызова и взаимодействия конструкторов и деструкторов:

1. При объявлении локального объекта. Конструкторы вызываются при входе в блок, деструкторы – при завершении блока.

Пример:

```
{
 Complex C;          // конструктор по умолчанию
 Complex C1(3.,2.); // конструктор инициализатор
 Complex C2(5.);     // конструктор преобразования типа
```

```
 Complex C4(C1);    // конструктор копирования
 . . . . . . . . . . . . . . . . . . . .
} // Деструктор для каждого созданного объекта
```

2. При объявлении глобального объекта. Конструкторы вызываются до начала работы функции main, деструкторы – после завершения работы main.

Пример:

```
Complex C;          // конструктор по умолчанию
Complex C1(3.,2.); // конструктор инициализатор
void fun()
{
static Complex C2(5.); // конструктор преобразования типа
   . . . . . . . . .
}
```

3. При создании и удалении динамического объекта. Конструкторы вызываются при выполнении new, деструкторы – при выполнении delete.

Пример:

```
Complex* C1 = new Complex();  // конструктор по умолчанию
Complex* C2 = newComplex (3.,2.); // констр.инициализатор
. . . . . . . . . . . . . . .
delete C1; // деструктор для C1
delete C2; // деструктор для C2
```

4. При копировании принимаемых и возвращаемых функцией значений:

Пример:

```
Complex fun(Complex _c) // конструктор копирования для _c
{
  Complex tmp; // конструктор по умолчанию для tmp
  . . . . . . . . .
return tmp;  // конструктор копирования для
// возвращаемого значения
} // деструкторы для _c и tmp
```

4. ОБРАБОТКА ИСКЛЮЧЕНИЙ

4.1. Ошибки и их обработка

При написании программного кода часто возникают ситуации выявления ошибок в исходных данных, поведении среды. …, которые надо как-то обработать.

Такие ситуации будем рассматривать на примере функции перевода строкового представления целого числа в целочисленный тип:

```
int StrToInt(constchar str[])
{
int num = 0;
int i = 0;
while(str[i])
if (str[i]>='0'&& str[i]<='9')
num = num * 10 + str[i++] - '0';
else
//???????????
return num;
}
```

В этой функции выполняется проверка, является ли очередной символ полученной строки цифровым. Если да, то процесс перевода идет далее. Если нет, то надо как-то сообщить вызывающей функции, что в исходных данных вызова была ошибка. Получив такое сообщение, вызывающая функция сможет его обработать: либо завершить выполнение программы с соответствующей диагностикой, либо повторить вызов функции с правильными исходными данными.

Как это сделать? Рассмотрим несколько способов.

4.1.1. Код возврата

Использование кода возврата выглядит следующим образом:

```
bool StrToInt(constchar str[], int& num)
{
bool rez = true;
  num = 0;
int i = 0;
while(str[i])
if (str[i]>='0'&& str[i]<='9')
num = num * 10 + str[i++] - '0';
     else
```

```
      rez = false;
return rez;
}
```

Здесь функция `StrToInt` результат `num` передает через формальный параметр, а в качестве возвращаемого значения передает сообщение удачно или нет был выполнен перевод.

Вызов такой функции будет иметь вид:

```
if ( !StrToInt("234", cn) )
{
// обработка ???????
}
```

4.1.2. Код завершения

Использование кода завершения для контроля и обработки ошибок имеет вид:

```
int _ERCODE;

int StrToInt(constchar str[])
{
  _ERCODE = 0; // OK
int num = 0;
int i = 0;
while(str[i])
if (str[i]>='0'&& str[i]<='9')
num = num * 10 + str[i++] - '0';
     else
       _ERCODE = 1; // Not numeric
return num;
}
```

Здесь объявлена глобальная переменная `_ERCODE` (код завершения) и функция `StrToInt` устанавливает его в ноль (нормальное завершение) или в единицу (нечисловой символ).

Вызов такой функции будет выглядеть так:

```
int cn = StrToInt("234");
if (_ERCODE != 0 )
{
// обработка ???????
}
```

Следует отметить, что общим недостатком приведенных способов контроля и обработки ошибок является то, что контроль и обработку надо вставлять после каждого обращения к функции , что значительное усложняет код и ухудшает его читаемость.

Для контроля и обработки ошибок в C++ встроен механизм обработки исключений, который в значительной степени свободен от этих недостатков.

4.1.3. Обработка исключений

Использование механизма обработки исключений в нашем случае имеет вид:

```
bool StrToInt(constchar str[], int& num)
{
bool rez = true;
  num = 0;
int i = 0;
while(str[i])
if (str[i]>='0'&& str[i]<='9')
num = num * 10 + str[i++] - '0';
else
  {
int exception = 1; // Not numeric
throw exception;
}
return rez;
}
```

Здесь в случае выявления ошибки в функции возбуждается исключение типа int (`throw exception;`) .

Код, использующий функцию `StrToInt` выглядит так:

```
try {
// контролируемый код
  . . . . . . . . .
  cn = StrToInt("234");
  . . . . . . . . .

}
catch ( int excep ) {
// обработка ???????
}
```

Здесь контролируемый код заключен в try-блок, после которого обработчик исключений (конструкция catch). Если при выполнении контролируемого кода будет возбуждено исключение, то управление перейдет

на обработку (catch). Если исключение не будет возбуждено, то по завершению блока try управление перейдет за конструкцию catch.

Основное преимущество такого способа в том, что контролируемый в блоке try может быть достаточно большим, а не только содержать одно обращение к функции `StrToInt`.

Разберемся в механизме обработки исключений подробнее.

4.2. Возбуждение и обработка исключений

4.2.1. Возбуждение исключения

Исключения возбуждаются с помощью операции throw, общий вид которой

```
throw<выражение_типа>;
```

где: выражение_типа – константа или переменная (объект) любого типа. Тип этого выражения называется типом возбуждаемого исключения.

Значение выражения будет передано обработчику исключения и может включать информацию, необходимую для правильной обработки исключения.

Пример 1.

```
throw 8;
```

Здесь возбуждается исключение типа int со значением 8.

Пример 2.

Объявляется структура `TMyExeption`, которая будет созданным типом исключения:

```
struct TMyExeption
{
int exeptionCode;
int errorPosition;
   TMyExeption(int _exeptionCode, int _errorPosition)
   {
      exeptionCode = _exeptionCode;
      errorPosition = _errorPosition;
   }
}
```

Вэтойструктуре:

- объявлены поля `exeptionCode` и `errorPosition` , которые будут нести информацию о конкретном исключении;
- описан конструктор инициализатор.

Возбуждение исключения типа `TMyExeption` может иметь вид:

```
throw TMyExeption(2, 24);
```

Здесь возбуждается исключение типа `TMyExeption` с передачей обработчику двух значений: 2 и 24.

4.2.2. Обработка исключений

Обработка исключений выполняется в конструкции try-catch, имеющей следующий вид:

```
try {
// контролируемый код
  . . . . . . . . . . .
}
catch ( <тип1> excep ) {
// обработка исключений
// типа <тип1>
}
catch ( <тип2> excep ) {
// обработка исключений
// типа <тип2>
}
. . . . . . . . . . . .
```

Действуют следующие правила обработки исключений:

1. Конструкция try-catch состоит из блока контролируемого кода try и следующих за ним обработчиков исключений catch.
2. Включенный в блок try контролируемый код может содержать возбуждение исключений различных типов; каждый обработчик catch обрабатывает исключение своего типа.
3. Если в блоке try

- исключения не возбуждались, то по завершению блока try управление передается за последний обработчик конструкции;
- возбудилось исключение типа <типN>, то управление передается в обработчик типа <типN>; по завершению обработчика управление передается за последний обработчик конструкции.

4. При передаче управления в обработчик для восстановления правильной работы программы выполняется «откат» стека вызова функций в блоке try.

5. Блок catch может принимать параметр по значению, по ссылке, по константной ссылке и по указателю.
6. Обработчики могут возбуждать исключения.
7. Конструкции try - catch могут вкладываться друг в друга. При этом:

- при возбуждении исключения типа <тип> ищется ближайший из охватывающих try – catch, который содержит обработчик типа <тип>;
- если такового нет, то обработку исключения выполняет операционная система, которая выдает стандартное диагностическое сообщение и завершает задачу.

4.2.3. Выход из обработчика catch

Существует несколько способов выхода из обработчика catch:

1. Простое завершение – выполнение обработчика до конца и переход за последний catch конструкции try – catch.
2. Безусловный переход (goto) на любую видимую метку вне конструкции try – catch.
3. Выход из функции (return), включающей конструкцию try-catch.
4. Возбуждение исключения:

- без параметра как возбуждение исключения того же типа с передачей его обработки на более высокий уровень вложенности конструкций try-catch;
- с параметром (throw <параметр>) – возбуждение исключения другого типа.

4.3. Пример

В качестве примера рассмотрим обработку исключения в уже известной нам функции `StrToInt`.

Прежде всего объявим тип исключения для нашей функции:

```
enum TStrEXType { strNOTNUMERIC, strOUTOGRANGE };
struct TStrException
{
  TStrEXType extype;  // вид исключения
int bspos;           // позиция нецифрового символа
TStrException(TStrEXType _extype, int _bspos):
extype(_extype), bspos(_bspos) {}
};
```

В перечисленном типе `TStrEXType` объявляем два вида исключений:

- `strNOTNUMERIC` - нечисловой символ;
- `strOUTOGRANGE` - выход за границу строки (в примере не используется).

Тип исключения структура содержит два поля: вид исключения и позиция нецифрового символа и конструктор инициализатор.

Сама функция `StrToInt` с возбуждением исключения выглядит так:

```
int StrToInt(constchar str[])
{
int num = 0;
int i = 0;
while(str[i])
if (str[i]>='0'&& str[i]<='9')
num = num * 10 + str[i++] - '0';
     else
throw TStrException(strNOTNUMERIC, i); // Not numeric
return num;
}
```

В качестве обработчика можно рассматривать следующую функцию:

```
int main()
{
  setlocale(LC_CTYPE, "Russian");
char str[50];
int num;
met:
try {
    cout <<"Число = ";
    cin >> str;
    num = StrToInt(str);
  }
catch ( TStrException& except ) {
    cout <<"символ '"<< str[except.bspos]
<<"' не числовой"<< endl;
goto met;
}
  cout << str <<" = "<< num << endl;

return 0;
}
```

Здесь в блок try вставлен ввод числа с клавиатуры и обращение к функции `StrToInt`. В обработчике выводится на консоль номер нечислового символа в строке и управление передается на повторение блока try.

5. ПЕРЕГРУЗКА ОПЕРАЦИЙ

5.1. Общие правила перегрузки операций

5.1.1. Перегрузка операции - функция

Перегрузка операций — специальный синтаксический механизм C++, который позволяет использовать стандартные операции языка (+, -, *, [], …) не только для переменных базовых типов, но и для объектов классов, для которых эти операции перегружены.

В языке С операция рассматривается как функция, принимающая параметры (операнды операции) и возвращающая значение – результат операции. Перегрузка операций в С++ это написание специальных функций (методов класса), описывающих выполнение той или иной операции.

Синтаксис такой функции:

```
<тип>operator<знак_операции>(<параметры>);
```

где: `operator<знак_операции>` - имя функции, перегружающей операцию `<знак_операции>`

Примеры:

```
// Перегрузка операции + для комплексных чисел
Ccomplexoperator+(constCcomplex&c1, constCcomplex&c2);

// Перегрузкаоперацииприсваиваниядлястроки
Cmystring&operator=(constCmystring& _str);
// Перегрузка операции индексации для строки
char&operator[](int index);
```

5.1.2. Два способа перегрузки операций

Операция может быть перегружена (1) как метод класса и (2) как обычная (внешняя по отношению к классу) функция.

Перегружающая операцию обычная функция получает свои операнды через формальные параметры, т.е. имеет два параметра для бинарной операции и один параметр для унарной операции.

Перегружающий операцию метод класса первый (левый) операнд получает как объект, через который вызывается метод, а второй – как формальный параметр метода. Т.е. перегружающий операцию метод имеет один параметр для бинарной операции и не имеет параметров для унарной операции.

Пример:

```
class Ccomplex
{
private:
```

```
double re, im;     // Действительная и мнимая части
public:
      . . . . . . . . . . . . . . . . . . .
      Ccomplex operator+(const Ccomplex& c2);
      . . . . . . . . . . . . . . . . . . .
};
Ccomplex Ccomplex::operator+(const Ccomplex& c2)
{
    Ccomplex tmp;
    tmp.re = re + c2.re; // т.е tmp.re = this->re + c2.re;
    tmp.im = im + c2.im;
return tmp;
}
Ccomplex operator-(const Ccomplex& c1, const Ccomplex& c2)
{
    Ccomplex tmp;
    tmp.re = c1.re - c2.re;
    tmp.im = c1.im - c2.im;
return tmp;
}
int main()
{
    Ccomplex a(1.0, 2.0), b(3.0, 4.0), c;
    c = a + b; // означает: c = a.operator+(b)
c = a - b; // означает: c =   operator-(a, b)
return 0;
}
```

В приведенном примере операция «+» перегружена как метод класса, а операция «-» - как обычная функция.

5.1.3. Общие правила и ограничения перегрузки операций

1. Для перегруженных операций при вычислении выражений сохраняются стандартные приоритеты операций и правила ассоциации (справа налево или слева направо).
2. Нельзя перегружать операции: «.», «::», «**sizeof**» и операции для стандартных (встроенных) типов данных
3. Функции и методы, перегружающие операции не могут иметь аргументов по умолчанию
4. Методы, перегружающие операции не могут объявляться как static (как const и virtual могут).
5. Операции: присваивания (=), вызова функции (), индексации ([ ]) и доступа по указателю (->) можно перегружать только как методы класса.
6. Одна и та же операция может быть перегружена несколько раз для различных типов операндов. Пример:

```
class Ccomplex
{
      . . . . . . . . . . . . . . . . . . .
```

```
      Ccomplex operator+(const Ccomplex& c2);
      Ccomplex operator+(double d);
      . . . . . . . . . . . . . . . . . .
};
Ccomplex Ccomplex::operator+(const Ccomplex& c2)
{
    Ccomplex tmp;
    tmp.re = re + c2.re;
    tmp.im = im + c2.im;
return tmp;
}
Ccomplex Ccomplex::operator+(double d)
{
    Ccomplex tmp;
    tmp.re = re + d;
    tmp.im = im;
return tmp;
}
int main()
{
    Ccomplex a(1.0, 2.0), b(3.0, 4.0), c;
    c = a + b;
c = a + 3.8;
return 0;
}
```

В приведенном примере операция «+» перегружена дважды: для случая, когда правый операнд является комплексным числом и для случая, когда правый операнд – действительное число.

7. Операция, левый операнд которой не является объектом класса должна перегружаться как обычная функция. Если функция, перегружающая такую операцию должна иметь доступ к закрытым полям класса, то она должна быть объявлена в классе как дружественная (friend). Пример:

```
class Ccomplex
{
      . . . . . . . . . . . . . . . . . .
             Ccomplex operator+(const Ccomplex& c2);
      friend Ccomplex operator+(double d, const Ccomplex& c2);
      . . . . . . . . . . . . . . . . . .
};
Ccomplex Ccomplex::operator+(const Ccomplex& c2)
{
    Ccomplex tmp;
    tmp.re = re + c2.re;
    tmp.im = im + c2.im;
return tmp;
}
Ccomplex operator+(double d, const Ccomplex& c2)
{
    Ccomplex tmp;
    tmp.re = d + c2.re;
    tmp.im =     c2.im;
return tmp;
}
int main()
{
    Ccomplex a(1.0, 2.0), b(3.0, 4.0), c;
    c = a + b;
```

```
c = 3.8 + a;
return 0;
}
```

5.2. Правила перегрузки отдельных типов операций

Приведенные ниже правила перегрузки операций обеспечивают корректное выполнение операций, вычисление выражений, передачу выражений в качестве параметров и т.д. При необходимости отдельные позиции правил могут быть нарушены, но при этом программист должен брать на себя ответственность за последствия таких нарушений.

5.2.1. Перегрузка операции «=» (присваивания)

Метод (операция присваивания может быть перегружена только как метод класса), перегружающий операцию присваивания должен:

1. Иметь возвращаемый тип – ссылка на объект.
2. Перед выполнением присваивания проверять, не делается ли попытка присвоить «себя себе».
3. Присваивать полученное значение себе
4. Возвращать себя: return*this;

Пример:

```
Ccomplex& Ccomplex::operator=(const Ccomplex& C)// возвращаемыйтип Ccomplex&
{
      if ( this != &C ) // проверка (указатель на меня) != (адрес C)
      {
            re = C.re;
            im = C.im;
      }
return *this;       // возвратсебя
}
```

5.2.2. Перегрузка операций типа «+»

К операциям типа «+» относятся бинарные операции, тип возврата которых совпадает с типом операндов. Это арифметические операции: «+»,«-»,«*»,«/»,«%»; операции сдвига, побитовые операции «&»,«|»,«^».

Метод или функция, перегружающий операцию этого типа должен:

1. Иметь возвращаемый тип – объект.
2. Объявлять рабочий локальный объект tmp для результата операции

3. Выполнить операцию и присвоить результат рабочему локальному объекту tmp
4. Возвращать рабочий локальный объект: return tmp;

Пример:

```
Ccomplex Ccomplex::operator+(const Ccomplex& c2)// Возвращаемыйтип Ccomplex
{
    Ccomplex tmp;          // Локальный объект для результата операции
    tmp.re = re + c2.re; // Результат операции присвоить tmp
tmp.im = im + c2.im;
return tmp;            // Возврат tmp
}
```

5.2.3. Перегрузка операций типа «+=»

К операциям типа «+=» относятся все операции типа «сделать с собой». Эти операции являются комбинацией операций описанных выше типов.

Метод или функция, перегружающий операцию этого типа должен:

1. Иметь возвращаемый тип – ссылка на объект.
2. Выполнить операцию «над собой»
3. Возвращать себя: return *this;

Пример:

```
Ccomplex& Ccomplex::operator+=(const Ccomplex& C)// Возвращ.тип Ccomplex&
{
    re += C.re;     // Операция
    im += C.im;     // "над собой"
return *this;   // Возврат себя
}
```

5.2.4. Перегрузка операций сравнения (отношения)

К операциям сравнения относятся операции «==»,«!=»,«<=»,«>=»,«<»,«>», возвращающие тип bool.

Метод или функция, перегружающий операцию сравнения должен:

1. Иметь возвращаемый тип – bool.
2. Метод должен быть константным.
3. Выполнить сравнение.
4. Возвращать результат сравнения.

Пример:

```
booloperator==(const Ccomplex& C) const
{
      return (re == C.re && im == C.im);
```

```
}
```

5.2.5. Перегрузка операции «[]» (индексации)

Перегрузка операции индексации обычно применяется к объекту со свойствами массива (имеющему поле-массив). Операция индексации должна быть перегружена в двух вариантах: обычном и константном. Обычная перегрузка обеспечивает использование операции индексации в левой и правой части операции присваивания. Константная - только в правой. Константный вариант перегрузки необходим для использования операции в константных методах или для константных объектов.

Метод (операция индексации может быть перегружена только как метод класса), перегружающий операцию индексации в обычном варианте должен:

1. Иметь возвращаемый тип – ссылка на тип элемента индексируемого массива.
2. Перед выполнением операции проверять, не выходит ли запрашиваемое значение индекса за границы массива и возбуждать при необходимости исключение OUTOFRANGE.
3. Возвращать запрашиваемый элемент массива.

Метод, перегружающий операцию индексации в константном варианте, отличается от обычного только тем, что является константным методом класса и должен иметь возвращаемый тип – константная ссылка на тип элемента индексируемого массива.

Пример:

```
enum TCmystringExeption { strINDOUTOFRANGE }; // тип исключения
class Cmystring
{
private:
char* str;   // строка
int size;    // размер (max)
int leng;    // длина реальная
public:
  . . . . . . . . . . . . . . . . . . . . . . .
char&operator[](int index);          // операцияиндексации
constchar&operator[](int index) const;     // операцияиндексации
  Cmystring&operator=(const Cmystring& _str) // операцияприсваивания
  . . . . . . . . . . . . . . . . . . . . . . .
};

char& Cmystring::operator[](int index)     // операцияиндексации
{
   if ( index < 0 || index >= leng ) {
      TCmystringExeption expt = strINDOUTOFRANGE;
      throw expt;
   }
   return str[index];
}
```

```
constchar& Cmystring::operator[](int index) const     // операцияиндексации
{
   if ( index < 0 || index >= leng ) {
      TCmystringExeption expt = strINDOUTOFRANGE;
      throw expt;
   }
   return str[index];
}

Cmystring& Cmystring::operator=(const Cmystring& _str)
{
if ( this != &_str )
   {
if ( size < _str.size )
      {
delete[] str;
         size = _str.size;
         leng = _str.leng;
         str = newchar[size];
      }
for (int i = o; i < leng; i++)
         (*this)[i] = _str[i];
}
return *this;
}
```

В приведенном примере в методе, перегружающем операцию присваивания, используется ранее перегруженная операция индексации: (*this)[i] = _str[i];. Причем, в левой части используется обычная перегрузка, а в правой – константная. Почему?

Примечание. Операция индексации может быть перегружена не только для параметра целочисленный индекс, но и для любого типа параметра. В частности, операцию индексации перегружают для объекта типа контейнер, имеющего набор (список) объектов некоторого класса. Операция индексации в этом случае выполняет поиск объекта набора по одному из его признаков. Подробнее – см. тему Контейнер.

5.2.6. Перегрузка унарных операций

Список параметров методов, перегружающих унарные операции (`++ -- ~ ! - & * new delete`) пуст

Для постфиксного инкремента и декремента необходимо включить в список параметров 1 фиктивный параметр типа int.

Пример:

```
class Ccomplex
{
private:
double re, im;     // Действительная и мнимая части
public:
   . . . . . . . . . . . . . . . . . . . . . . . . . .
    Ccomplex operator++();     // префиксный ++
```

```
    Ccomplex operator++(int); // постфиксный ++
    Ccomplex operator-();     // унарный -
    . . . . . . . . . . . . . . . . . . . . .
};

Ccomplex Ccomplex::operator++(int) // постфиксный ++
{
      Ccomplex tmp(*this);
      re = re + 1.0;
      im = im + 1.0;
      return tmp;
}
Ccomplex Ccomplex::operator++() // префиксный ++
{
      re = re + 1.0;
      im = im + 1.0;
      return *this;
}
Ccomplex Ccomplex::operator-()
{
return Ccomplex(-re, -im);
}
```

5.2.7. Перегрузка операций обмена с потоком

Операции << и >> (операции сдвига) уже перегружены в классах потока. Для «своего» класса их надо перегружать как операции потока

Общее правило перегрузки операций << и >>:

1. Операции перегружаются как внешние функции (при необходимости – как дружественные)
2. Синтаксис прототипов:

```
ostream&operator<<(ostream&, const MyClass&);
istream&operator>>(istream&,       MyClass&);
```

3. Функции должны возвращать поток

Пример:

```
class Cmystring
{
private:
char* str;   // строка
int size;    // размер (max)
int leng;    // длина реальная
public:
// . . . . . . . . . . . . . . . . . . . . .
friend ostream&operator<<(ostream& stream, const Cmystring& _str);
friend istream&operator>>(istream& stream,       Cmystring& _str);
};

// -----------------------------------------------------------------
ostream&operator<<(ostream& stream, const Cmystring& _str)
{
    stream << _str.str << endl;
return stream;
}
```

```
// ---------------------------------------------------------------
istream&operator>>(istream& stream,       Cmystring& _str)
{
char ss[255];
    stream.getline(ss, 255);
_str = ss;
return stream;
}
```

6. НАСЛЕДОВАНИЕ

6.1. Наследование классов

6.1.1. Что такое наследование классов

Классы могут объявляться как наследники других, ранее объявленных классов. Синтаксис объявления класса-наследника имеет вид:

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

Здесь класс B объявляется как наследник ранее объявленного класса A. В разных литературных источниках можно встретить различные варианты терминологии:

- предок – потомок;
- суперкласс – подкласс;
- базовый класс – подчиненный класс.

Иногда говорят, что класс B унаследован (наследует) от класса A.

При наследовании в классе-потомке:

- сохраняет свойства (поля) и методы класса-предка;
- могут быть добавлены новые свойства и методы;
- могут быть заменены (замещены) некоторые методы класса-предка.

Пример:

```
class A
{
int i1;
void f1();
void f2();
};
class B : public A // Наследник
{
int i2;
void f1();      // Замещение
void f3();
};
```

В приведенном примере:

1. Класс B объявлен как наследник класса A.
2. Класс B:
   a. наследует поле i1 и метод f2 класса A:
   b. в нем объявлены «свои» поле i2 и метод f3;
   c. в нем перекрыт (замещен) метод f1 класса A.

6.1.2. Перегрузка и перекрытие методов

С наследованием классов связан механизм перекрытия, или замещения (overriding) методов предка методом потомка. Отличие механизма перегрузки методов класса (overloading) от механизма перекрытия продемонстрируем на следующих примерах.

Пример 1.

```
class A
{
int i1;
void f1(int a);
void f1(double d); // Перегрузкав A
};
class B : A          // Наследник
{
int i2;
void f3(int a);
};
int main()
{
  A a;
  B b;
  b.f1(1); // A::f1(int)
  b.f1(1.);// A::f1(double)
b.f3(2); // B::f3(int)
}
```

В приведенном примере:

1. В классе A объявлен и перегружен метод f1.
2. Класс B наследует оба метода f1 класса A и в нем объявлен новый метод f3.
3. В клиенте классов A и B (функции main) объявлены объекты этих классов, которым доступны поля и методы, описанные в примере. При этом, вызов b.f1(1) означает вызов метода f1(int) класса A; вызов b.f1(1.) – вызов метода f1(double) класса A; вызов b.f3(2) – вызов метода f3(int) класса B.

Пример 2.

```
class A
{
int i1;
void f1(int a);
void f1(double d); // Перегрузкав A
};
class B : A           // Наследник
{
int i2;
void f3(int a);
void f1(int a);     // Перекрытие
};
int main()
{
  A a;
  B b;
  b.f1(1);     // B::f1(int)
  b.f1(1.);    // Ошибка: в B нет f1(double)
  b.A::f1(1); // A::f1(int)
  b.A::f1(1.);// A::f1(double)
}
```

Приведенный пример отличается от предыдущего только тем, что в классе B перекрыт метод f1(int) класса A. В этом случае в клиенте:

- вызов b.f1(1) будет означать вызов метода f1(int) класса B;
- вызов b.f1(1.) будет ошибкой, т.к. перекрытие метода f1 в B означает перекрытие всех методов f1 класса предка и класс B уже не наследует метод f1(double);
- вызов b.A::f1(1) будет означать вызов метода f1(int) класса A;
- вызов b.A::f1(1.) будет означать вызов метода f1(double) класса A.

Таким образом:

- перегрузка (overloading) методов означает объявление в классе насколько методов с одним именем и различными сигнатурами (списками формальных параметров); перегрузка работает «внутри» класса и не связана с механизмом наследования;
- перекрытие (overriding) методов означает объявление в классе-потомке метода одноименного с одним из методов класса-предка; перекрытие одного метода класса-предка перекрывает все перегруженные методы класса-предка с несовпадающей и неприводимой сигнатурой.

6.1.3. Спецификатор доступа protected

Применительно к механизму наследования наряду со спецификаторами доступа public и private применяется спецификатор доступа protected. Объявленные в разделе protected поля и методы класса не видны клиенту класса, но видны методам потомков этого класса.

Пример.

```
class A
{
private:      // не виден потомку; не виден клиенту
int a;
public:       // виден потомку; виден клиенту
int b;
protected:    // виден потомку; не виден клиенту
int c;
};
class B : A
{
void f();
}
void B::f()
{
   a = 0; // ошибка доступа
   b = 0; // доступен
   c = 0; // доступен
}
int main()
{
   A aObject;
   aObject.a = 1;    // ошибкадоступа
   aObject.b = 1;    // доступен
   aObject.c = 1;    // ошибкадоступа
}
```

В приведенном примере в методе f класса B поле a класса-предка A недоступно; поля b и c доступны. В клиенте класса A (функции main) доступно только поле b.

6.2. Спецификаторы доступа при наследовании

6.2.1. Доступ при наследовании

При объявлении класса-наследника в языке C++ должен быть указан спецификатор доступа при наследовании:

```
class B : <спецификатор_доступа> A
```

```
{
      // . . . . . . . . . . .
}
```

где в качестве спецификатора доступа может быть указано: private, protected или public (по умолчанию – private).

Взаимодействие спецификаторов доступа в классе-предке со спецификаторами наследования представлено в таблице 1.

Доступ в классе-предке	**Спецификатор наследования**	**Доступ в классе-потомке**	**Доступ в клиенте класса-потомка**
private	private	не видим	cant access private
protected		private	cant access private
public		private	cant access private
private	protected	не видим	cant access protected
protected		protected	cant access protected
public		protected	cant access protected
private	public	не видим	because uses private
protected		protected	because uses protected
public		public	Видим

Таблица 1. Взаимодействие спецификаторов доступа при наследовании.

Из приведенной таблицы видно:

1. При всех видах наследования закрытые (private) поля предка не видны в классе –потомке (и в клиентах класса-потомка).
2. При закрытом наследовании (private) защищенные (protected) и открытые (public) поля предка наследуются в потомке как закрытые.

3. При защищенном наследовании (protected) защищенные (protected) и открытые (public) поля предка наследуются в потомке как защищенные.
4. При открытом наследовании (public) защищенные (protected) и открытые (public) поля предка сохраняют свой тип защиты.

6.2.2. Взаимодействие спецификаторов доступа при наследовании

Схему взаимодействия спецификаторов доступа иллюстрирует также рис.5:

- при открытом наследовании (класс B) поля i1 и i2 сохраняют свои уровни защиты;
- при защищенном наследовании (класс C) они наследуются как защищенные;
- при закрытом наследовании (класс D) они наследуются как закрытые.

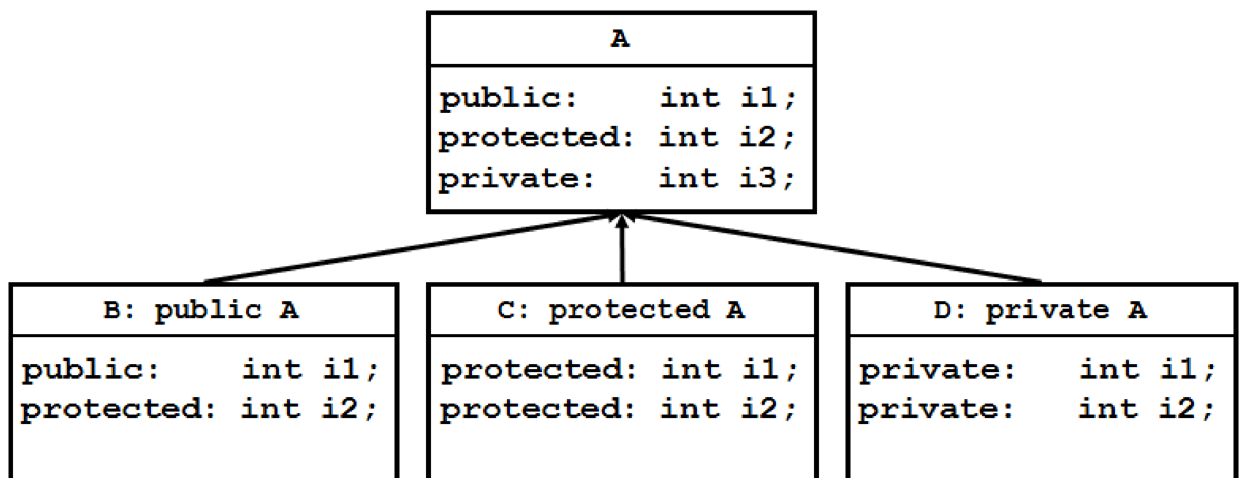

Рис. 5. Взаимодействие спецификаторов доступа при наследовании.

Из этого следует, что клиенты классов C и D вообще не видят класс A. Также класс A не видят потомки класса D. Зачем может быть нужно защищенное и закрытое наследования?

Закрытое наследование обычно используется для создания клона класса с ограниченной функциональностью.

Пример.

```
class A
{
public:
```

```
void fa1();
void fa2();
};
class AClone : private A
{
public:
void fa1() { A::fa1(); } // сохранение функциональности
void faclone();          // новая функциональность
// у объектов класса AClone функциональности fa2 нет
};
```

В приведенном примере:

- объявлен класс A, имеющий два открытых метода: fa1 и fa2;
- класс AClone закрыто наследуется от класса A;
- в классе AClone открыто перекрыт метод fa1 предка с явным вызовом метода fa1 предка – функциональность fa1 в классе AClone сохраняется; функциональность fa2 – нет;
- также в классе AClone добавлен новый метод faclone.

Защищенное наследование может быт использовано для иерархического клонирования классов.

6.3. Конструирование и деструктурирование потомков

6.3.1. Конструирование потомков

Схему конструирования объектов при наследовании рассмотрит на следующем примере.

```
class A
{
private:
int* a;
int na;
public:
    A(int _na)
    {
      na = _na; a = newint[na];
}
};
class B : public A
{
private:
char* b;
int nb;
```

```
public:
B(int _na, int _nb) : A(_na)
    {
      nb = _nb; b = newchar[nb];
}
};
class C : public B
{
private:
double* c;
int nc;
public:
    C(int _nc, int _na, int _nb)
    {
      B(_na, nb);
      nc = _nc; c = newdouble[nc];
    }
};
int main()
{
  C cObj(10, 15, 20);
  // . . . . . . . . .
  return 0;
}
```

В приведенном примере:

- объявлен класс A, от которого открыто унаследован класс B, от которого открыто унаследован класс C;
- в каждом классе закрыто объявлены динамические массивы: указатели на массив и длина этого массива;
- в каждом классе открыто объявлена и описаны конструкторы инициализаторы, которые создают в куче «свои» динамические массивы;
- выполнение каждого конструктора потомка начинается с явного вызова конструктора прямого предка;
- в результате при объявлении в клиенте класса C (функция main) объекта cObj этот объект будет конструироваться поэтапно сверху-вниз (рис.6.): на первом этапе будет сконструирована часть A объекта, на втором – часть B и на третьем – часть C.

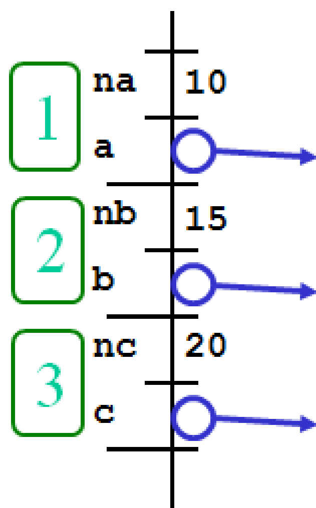

Рис. 6. Схема конструирования объектов при наследовании

Конструирование потомков подчиняется следующим общим правилам:

1. Конструкторы не наследуются, поэтому класс-потомок должен иметь собственные конструкторы.
2. Конструктор класса-потомка должен вызывать конструктор прямого предка. При этом:

- при отсутствии явного вызова конструктора предка неявно (автоматически) вызывается конструктор предка по умолчанию (конструктор без параметров);
- при необходимости конструктор потомка может явно вызвать конструктор предка любого типа (но только один!);
- конструктор предка должен быть вызван в начале работы конструктора потомка (лучше – в списке инициализации).

3. Многоуровневый объект конструируется «сверху-вниз» от предков к потомкам.
4. Конструкторы не могут быть виртуальными (также как не могут быть объявлены со спецификаторами static или const).

6.3.2. Деструктурирование потомков

Схему деструктурирования объектов при наследовании рассмотрит на следующем примере, являющемся продолжением предыдущего примера.

```
class A
{
protected:
int* a;
```

```
int na;
public:
    ~A()
    {
delete[] a; a = 0; na = 0;
}
};
class B : public A
{
protected:
char* b;
int nb;
public:
    ~B()
    {
delete[] b; b = 0; nb = 0;
    }
};
class C : public B
{
protected:
double* c;
int nc;
public:
    ~C()
    {
delete[] c; c = 0; nc = 0;
}
};
```

В приведенном примере:

- в каждом классе открыто объявлена и описаны деструкторы, которые освобождают в куче «свои» динамические массивы;
- деструкторы потомков не содержат явных вызовов деструкторов прямых предков, деструкторы прямых предков вызываются неявно (автоматически);
- деструкторы прямых предков вызываются в конце работы деструкторов потомков;
- в результате при завершении работы клиента класса C (функция main) при неявном вызове деструктора объекта cObj этот объект будет деструктурироваться поэтапно снизу-вверх (рис.7.): на первом этапе будет деструктурирована часть C объекта, на втором – часть B и на третьем – часть A.

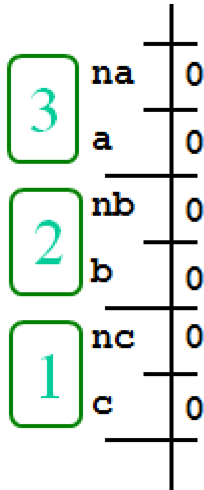

Рис. 7. Схема деструктурирования объектов при наследовании.

Деструктурирование потомков подчиняется следующим общим правилам:

1. Деструкторы не наследуются, поэтому класс-потомок должен иметь собственный деструктор. При этом:

- если у класса-потомка нет явного деструктора, то созданный по умолчанию деструктор вызывает деструктор прямого предка;
- в явно описанном деструкторе класса-потомка деструктор предка явно вызывать не надо – деструктор предка также будет вызван автоматически;
- деструктор предка автоматически вызывается после завершения работы деструктора потомка.

2. Деструкторы многоуровневого объекта вызываются в порядке, обратном порядку вызова конструкторов («снизу вверх»).
3. Деструкторы могут быть виртуальными (но не static или const).

6.4. Пример

6.4.1. Класс Triad

Постановка задачи

Создать класс Triad (тройка чисел); определить методы увеличения полей на целое значение.

Определить класс-наследник Time с полями: час, минута, секунда. Перекрыть методы увеличения полей предка, считая, что первое число – секунды, второе число – минуты, третье число – часы.

Класс Triad

```
class Triad
{
 protected:
int n1, n2, n3;
 public:
Triad();
   Triad(int _n1, int _n2, int _n3);
void addn1(int d) { n1 += d; }
void addn2(int d) { n2 += d; }
void addn3(int d) { n3 += d; }
};
// ---------------------------------------------------------------
--
Triad::Triad(): n1(0), n2(0), n3(0) {}
// ---------------------------------------------------------------
--
Triad::Triad(int _n1, int _n2, int _n3): n1(_n1), n2(_n2),
n3(_n3){}
```

В классе Triad:

- объявлены три целочисленных поля: n1, n2 и n3; поля объявлены как защищенные потому, что должны быть доступны наследникам класса и не доступны клинетам;
- объявлен конструктор по умолчанию и конструктор инициализатор (в рамках рассматриваемой задачи достаточно);
- объявлены и описаны в объявлении класса методы addn1, addn2 и addn3 увеличения полей класса на заданное значение;
- описаны конструктор по умолчанию и конструктор инициализатор.

6.4.2. Класс Time

```
class Time : public Triad
{
 public:
   Time();
   Time(int _sec, int _min, int _hou);
void addn1(int sec = 1);
void addn2(int min = 1);
```

```
void addn3(int hou = 1);
};
// ---------------------------------------------------------------
--
Time::Time(): Triad() { }
// ---------------------------------------------------------------
--
Time::Time(int _sec, int _min, int _hou) : Triad(_sec, _min, _hou)
{ }
// ---------------------------------------------------------------
--
void Time::addn1(int sec)        {
n1 += sec;
if ( n1 > 59 )
   {
      addn2(n1/60);   n1 %= 60;
}
}
// ---------------------------------------------------------------
--
void Time::addn2(int min)        {
   Triad::addn2(min);
if ( n2 > 59 )
   {
      addn3(n2/60);   n2 %= 60;
   }
}
// ---------------------------------------------------------------
--
void Time::addn3(int hou)        {
n3 += hou;
if ( n3 > 24 )
      n3 %= 24;
}
```

Вклассе Time:

- поля n1, n2 и n3 унаследованы от класса Triad и интерпретируются как секунды, минуты и часы соответственно;
- объявлен конструктор по умолчанию и конструктор инициализатор, принимающий количества секунд, минут и часов;
- перекрыты методы addn1, addn2 и addn3;
- в описании конструкторы класса Time реализуются через соответствующие конструкторы предка;
- в описании перекрытых методов addn1, addn2 и addn3 выполняется контроль «переполнения» заданного количества секунд, минут и часов с учетом перехода «лишних» секунд в минуты, а «лишних» минут в часы.

Следует отметить, что приведенный выше конструктор инициализатор класса Time будет работать правильно только при задании правильных количеств секунд, минут и часов. Если при обращении к этому конструктору будут указаны «лишние» секунды, минуты или часы, то это приведет к ошибке, избежать которой можно описав конструктор следующим образом:

```
Time::Time(int _sec, int _min, int _hou)
{
   n1 = 0;
   n2 = 0;
   n3 = 0;
   addn3(_hou);
addn2(_min);
   addn1(_sec);
}
```

7. ВИРТУАЛЬНЫЕ МЕТОДЫ И АБСТРАКТНЫЕ КЛАССЫ

7.1. Понятие виртуального метода

7.1.1. Что такое виртуальный метод?

Понятие виртуального метода поясним на следующем примере:

```
class A
{
private:
int a;
public:
   A(int _a): a(_a)    {}
long f1 (double d)    { cout <<"A::f1 "<< endl; return 0; }
};
class B: public A
{
private:
int b;
public:
   B(int _a, int _b): A(_a), b(_b)    {}
long    f1 (double d)    { cout <<"B::f1 "<< endl; return 0; }
};
void fa(A& ao)
{
   ao.f1(4);
}

int main()
{
   A a(1);
   fa(a);
B b(1,2);
   fa(b);
}
```

В приведенном примере:

1. Объявлен класс A, в котором объявлено поле a и описаны конструктор инициализатор и метод f1/
2. Класс B объявлен как открытый наследник класса A, в котором объявлено поле b, описан конструктор инициализатор и перекрыт метод f1.
3. Функция fa получает ссылку на объект класса A и обращается к методу f1этого объекта.

4. В функции main объявляются объекты a (класса A) и b (класса B) и выполняется вызов функции fa с передачей ей объекта a и потом – объекта b.
5. При первом вызове функции fa с параметром a будет вызван метод f1класса A.
6. При втором вызове функции fa с параметром b будет также вызван метод f1класса A. Здесь функция fa считает, что получила объект класса A и «не заметит», что ей передали объект класса B.

Но если, при объявлении метода f1 в классе A добавить спецификатор virtual:

```
virtuallong f1 (double d)   { cout <<"A::f1 "<< endl; return 0; }
```

то при втором вызове функции fa с параметром b будет вызван метод f1класса B. Здесь функция fa хотя и считает, что получила объект класса A, но «заметит», что ей передали объект класса B.

Таким образом:

Виртуальными (виртуально перекрытыми) методами называется цепочка перекрытых в потомках методов, для которых первое объявление метода в классе предке сделано со спецификатором virtual.

При вызове виртуального метода через ссылку (или указатель) на объект, тип вызываемого виртуального метода определяется не типом ссылки (или указателя), через который он вызывается, а типом объекта, «подставленного» под эту ссылку (или указатель).

7.1.2. Особенности виртуального перекрытия методов

Виртуальное перекрытие методов имеет следующие особенности:

1. Виртуально перекрываются только точно перекрытые методы предка. Точное перекрытие метода в классе потомке означает:

- точное совпадение количества и типов параметров перекрывающего и перекрываемого методов без учета приведения типов;
- точное совпадение или ковариантное соответствие типов возвращаемых значений.

2. Типы возвращаемых значений методов ковариантно соответствуют, если:

- оба являются указателями или ссылками на объекты классов;
- класс указателя или ссылки, возвращаемого перегружающим методом прямо или косвенно выведен из класса указателя или ссылки, возвращаемого перегружаемым методом.

3. Методы, получающие в качестве параметра объект «своего» класса не могут быть виртуально перекрыты.
4. Методы, получающие в качестве параметра ссылку или указатель на объект «своего» класса могут быть виртуально перекрыты, но при этом:

- виртуально перекрываемые методы потомков получают в качестве параметров указатели или ссылки на объекты не «своего» класса, а на объекты базового класса, которые надо явно приводить к указателям или ссылкам на объекты «своего» класса с помощью операции static_cast<…>;
- применение операции явного приведения типов блокирует синтаксический контроль типов и перекладывает на программиста ответственность за правильный вызов виртуально перекрытых методов и правильное их кодирование.

Кроме того:

1. Виртуальным достаточно объявить только метод базового класса в иерархии наследования. Точно перегруженные методы потомков автоматически становятся виртуальными
2. Виртуальные методы:

- наследуются (как обычно)
- предка доступны в потомке через операцию разрешения видимости (как обычно):

```
<имя_предка>::<вызов_метода>;
```

3. Виртуальный метод:

- не может объявляться с модификатором static, friend (const – может)
- не срабатывает через объект (только через указатель и ссылку)

4. Конструктор класса не может быть виртуальным
5. Деструктор – может (и должен!!)

7.1.3. Механизм позднего связывания

Ранее и позднее связывание

В этом разделе мы дадим ответ на вопрос, как работают виртуальные методы? Т.е. как система «узнает», какой из виртуально перекрытых методов ей надо вызывать?

При вызове функций могут работать два механизма: механизм раннего связывания и механизм позднего связывания:

- при раннем связывании адрес передачи управления на вызываемую функцию определяется на шаге линковки программы и во время работы программы не меняется;
- при позднем связывании адрес передачи управления на вызываемую функцию определяется на выполнения программы и определяется контекстом ее вызова в том или ином случае.

Виртуальные методы вызываются с помощью механизма позднего связывания. Механизм позднего связывания виртуальных методов реализуется с помощью таблиц виртуальных методов.

Таблицы виртуальных методов

Работа механизма позднего связывания выполняется с помощью создаваемых компилятором таблиц виртуальных методов (vtbl). Работа механизма позднего связывания на основе этих таблиц в случае программы вида:

```
class A {
int f1();
virtualint f2();
virtualint f3();
};
class B : A {
int f1();
int f2();
int f3();
};
int main () {
  A a;
  B b;
}
```

представлена на рис.8.

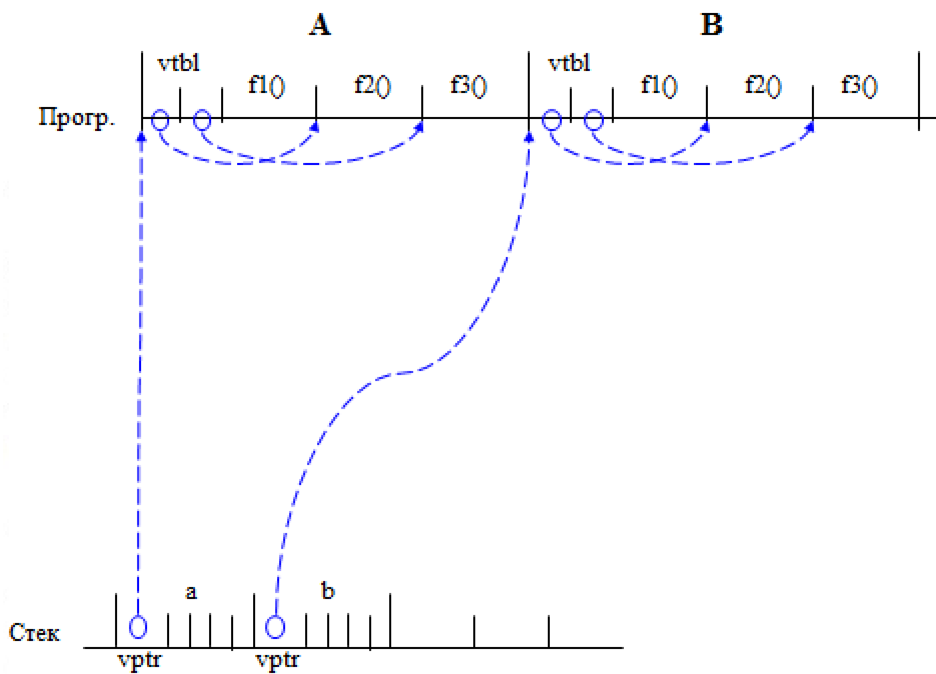

Рис. 8. Схема работы механизма позднего связывания.

Механизм позднего связывания работает следующим образом:

1. Для каждого класса, содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (vtbl). Таблица виртуальных методов:

- размещается в программном сегменте перед методами класса (см. рис.8);
- для каждого виртуального метода содержит его адрес в программном сегменте;
- адреса методов содержатся в таблице в порядке их описания в классах.

2. Каждый объект класса с виртуальными методами содержит скрытое дополнительное поле vptr (виртуальный табличный указатель) указатель на таблицу виртуальных методов класса. Оно формируется конструктором класса при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).
3. На этапе компиляции вызовы виртуальных методов заменяются на обращения к vtbl через vptr объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы.

7.1.4. Что дает использование виртуальных методов?

Вызов виртуальных методов, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы.

Это несколько замедляет выполнение программы. Виртуальность отрицательно сказывается на быстродействии программы, поэтому методы имеет смысл делать виртуальными тогда и только тогда, когда это действительно необходимо.

С другой стороны, использование виртуальных методов позволяет существенно упросить исходный код программы, что мы сейчас покажем на следующем примере.

Вспомним задачу о графическом редакторе темы «Введение в ООП». В одном из первых вариантов ее решения объявления фигур (точка, линия, треугольник) выглядят так:

```
// Объявления фигур -------------------------
class Cpoint
{
int x, y;   // Координаты точки
void show();
};
class Cline : public Cpoint
{
int x2, y2; // Координаты 2-ой точки
void show();
};
class Ctriangle : public Cline
{
int x3, y3; // Координаты 3-ой вершины
void show();
};
```

Описание каждой фигуры в виде структуры Tfigure включало тип фигуры и родовой указатель на фигуру:

```
// Описание фигуры ---------------------------
enum Tfigtype{point, line, triangle};// Тип фигуры
struct Tfigure  {   // Фигура
  Tfigtype figtype; // типфигуры
void *figure;     // указатель на описание фигуры
};
```

И, наконец, описание рисунка задавалось в виде массива описаний фигур:

```
// Описание рисунка (массива фигур) -----------
constint FIGMAX = 500;  // максим. к-во фигур
int figcount;            // реальное к-во фигур
Tfigure figlist[FIGMAX]; // массив фигур
```

В этом варианте функция отрисовки фигур рисунка выглядела так:

```
void showpicture(Tfigure figlist[] , int figcount)
{
```

```
for ( int i = 0; i < figcount; i++ )
   {
switch (figlist[i].figtype)
      {
case point:    (Cpoint*)   (figlist[i].figure)->show();   break;
case line:     (Cline*)    (figlist[i].figure)->show();   break;
case triangle: (Ctriangle*)(figlist[i].figure)->show();   break;
//. . . . . . . . . . . . . . . . . . . . . .
switch (??) {
case rr: ... break;
          . . . . . . . . .
switch (??) {
case tt: ... break;
            . . . . . . . . .
          }
          . . . . . . . . .
        }
      }
   }
}
```

Если же метод show класса Cpoint объявить виртуальным, а описание рисунка представить в виде:

```
const FIGMAX = 500;       // максим. к-во фигур
int figcount;             // реальное к-во фигур
Cpoint* figlist[FIGMAX]; // массив фигур
```

то функция отрисовки фигур рисунка будет выглядеть так:

```
void showpicture(Tfigure figlist[] , int figcount)
{
for ( int i = 0; i < figcount; i++ )
{
     figlist[i].figure->show();
  }
}
```

Возникает вопрос: а куда же делась вся сложность первого варианта функции отрисовки рисунка? Ответ простой: «ушла» в сложности работы с таблицами виртуальных методов.

7.2. Абстрактные методы и классы

7.2.1. Понятие абстрактного метода и класса

Вернемся к примеру предыдущего раздела. Там при объявлении фигур мы линию наследовали от точки, а треугольник от линии. Вообще это не очень удобно. Взглянув, например, «свежим взглядом» на объявление линии:

```
class Cline : public Cpoint
{
int x2, y2; // Координаты 2-ойточки
void show();
};
```

не сразу становится понятным, почему линия задается одной точкой; надо вспоминать, что линия выведена из точки и …

Более естественным выглядит следующее «независимое» объявления фигур:

```
class Cpoint
{
int x, y;}
void show();
}
class Cline
{
int x1, y1;
int x2, y2;
void show();
}
class Ctriangle
{
int x1, y1;
int x2, y2;
int x3, y3;
void show();
}
```

Но здесь уже нет наследования, и мы уже не можем объявить метод show виртуальным. Как быть?

В этом примере мы столкнулись с достаточно часто встречающейся ситуацией, когда есть несколько независимых классов, которые имеют одинаковое поведение: отрисовать себя, которое надо реализовать в виде виртуального метода. А для этого все эти классы должны быть унаследованы от какого-то одного класса, где этот метод объявлен как виртуальный. Интересно то, что этот общий класс предок ничего больше не должен делать, только объявить виртуальный метод show, который будет виртуально перекрываться в его потомках.

Для такой ситуации в языке C++ предусмотрен механизм абстрактных методов и абстрактных классов.

Абстрактный (чисто виртуальный) метод это виртуальный метод, который:

- объявлен со спецификатором = 0:

```
virtual<тип><имя>(<параметры>) = 0;
```

- не описан в классе, в котором объявлен.

Абстрактный метод это «фиктивный» метод, переопределяемый далее в потомках класса, в котором он объявлен.

Абстрактный класс это класс, который содержит хотя бы один абстрактный метод.

В нашем примере использование абстрактного класса Cshown будет выглядеть так:

```
class Cshown
{
virtualvoid show() = 0;
};
class Cpoint : Cshown
{
int x, y;}
void show();
};
class Cline : Cshown
{
int x1, y1;
int x2, y2;
void show();
};
class Ctriangle : Cshown
{
int x1, y1;
int x2, y2;
int x3, y3;
voidshow();
};
```

А описание рисунка – как массив указателей на Cshown:

```
const FIGMAX = 500; // максим. к-во фигур
int figcount;       // реальное к-во фигур
Cshown* figlist[FIGMAX]; // массивфигур
```

7.2.2. Свойства абстрактных методов и классов

1. Абстрактным является класс, имеющий хотя бы один абстрактный метод. Абстрактные классы реализуют вершину в иерархии наследования.
2. Правила и ограничения:

- нельзя объявлять объекты абстрактного класса;
- можно объявлять указатели и ссылки на объекты абстрактного класса, которым можно присваивать значения адресов объектов классов-наследников;
- функция не может принимать и возвращать объект абстрактного класса (может ссылки и указатели);
- абстрактный класс нельзя использовать для явного приведения типа;
- если класс, производный от абстрактного, не определяет все абстрактные методы предка, он также является абстрактным.

3. Абстрактный класс может иметь свою функциональность (поля, методы, конструкторы, деструктор). Чисто абстрактным называют класс, имеющий только объявления абстрактных методов.

Абстрактные классы появляются в результате объектно-ориентированного анализа и проектирования по следующей схеме:

1. Шаг 1. Абстрагирование (описание объектов). На основе анализа предметной области задачи:

- выявление объектов задачи;
- составление списка объектов с указанием их свойств (полей) и поведений (методов).

2. Шаг 2. Классификация (формирование классов):

- анализ списка объектов, выявление общих свойств и поведений;
- формирование классов и связей наследования между ними.

Если на шаге классификации выявляется некоторое поведение, общее для некоторой группы объектов, то его целесообразно представить абстрактным классом, из которого будут выводиться классы этих объектов.

7.2.3. Пример классификации

В результате анализа может быть выявлено, что общими поведениями геометрических фигур помимо умения отрисовать себя могут быть: умение

перемещаться в заданную точку рисунка, умение сохранять себя в файле и читать из файла и т.д.

Соответственно, выделяются три абстрактных класса:

```
class CShown
{
virtualvoid Show() = 0;
};
class CMoved
{
virtualvoid Move(int x, int y) = 0;
};
class CFiled
{
virtualvoid SaveToFile(char* filename) = 0;
virtualvoid LoadFromFile(char* filename)= 0;
};
```

Далеевозникаетнесколькововопросов.

Вопрос 1. От кого наследовать класс фигуры, у которой должно быть два поведения? Например, отрисовать себя и перемещаться?

Одно из решений объявить абстрактный класс, унаследованный от `CShown` и `CMoved`:

```
class CShownMoved : CShown, CMoved
{
public:
void Show() = 0;
void Move(int x, int y) = 0;
};
```

от которого наследовать класс фигуры.

Вопрос 2. Как описать массив фигур? Т.е. какого типа должны быть элементы этого массива, чтобы можно было объявленными виртуальными методами?

Здесь решение состоит в том, что:

1. Объявить класс «общий предок». Этот класс может быть пустым:

```
class CCommon  {  };
```

2. Все абстрактные классы унаследовать от этого класса.

Тогда, массив фигур можно объявить как массив указателей на `CCommon`:

```
const FIGMAX = 500; // максим. к-во фигур
int figcount;       // реальное к-во фигур
CCommon* figlist[FIGMAX]; // массив фигур
```

а отрисовку или перемещение очередной фигуры выполнять так:

```
((CShown*)(figlist[i]))->Show();
((CMoved*)(figlist[i]))->Move();
```

Следует отметить, что в нашем случае все фигуры должны иметь отмеченные поведения, и можно было бы сделать один абстрактный класс, включающий четыре абстрактных метода. На практике это бывает далеко не всегда так.

8. ВВЕДЕНИЕ В ШАБЛОНЫ

8.1. Шаблоны функций

8.1.1. Перегрузка функций: параметрический полиморфизм

Как отмечалось ранее, в языке C++ работает механизм перегрузки функций, когда можно описать несколько функций с одинаковым именем, но с разным составом формальных параметров.

Вспомним механизм перегрузки функций на следующем примере.

```
int mymin(int a, int b)
{
if ( a < b ) return a; elsereturn b;
}
double mymin(double a, double b)
{
if ( a < b ) return a; elsereturn b;
}
char mymin(char a, char b)
{
if ( a < b ) return a; elsereturn b;
}
//-------------------------------------------------
int main()
{
int i1 = 4, i2 = 7, i3;
   i3 = mymin(i1, i2);

double d1 = 4.67, d2 = 1.89, d3;
   d3 = mymin(d1, d2);

char c1 = 5, c2 = 8, c3;
   c3 = mymin(c1, c2);
return 0;
}
```

В приведенном примере:

- описаны три функции mymin, определяющих минимальное значение из двух значений полученных параметров;
- функции отличаются типом получаемых параметров и типом возвращаемого значения;

- в функции main описан вызов трех функций mymin: в первом случае будет вызываться mymin(int, int); во втором - mymin(double, double); в третьем - mymin(char, char).

При перегрузке функций компилятор определяет, какую из одноименных функций надо вызвать в том или ином случае, руководствуясь **правилами сигнатуры**:

1. Выбор среди перегруженных функций функции, у которой типы формальных параметров точно соответствуют типам фактических параметров вызова.
2. Если нет, то выбор среди перегруженных функций функции, у которой типы формальных параметров могут быть приведены к типам фактических параметров вызова с учетом встроенных преобразований типов.
3. Если нет, то выбор среди перегруженных функций функции, у которой типы формальных параметров могут быть приведены к типам фактических параметров вызова с учетом преобразований типов, описанных программистом (конструкторов преобразования типа)..
4. Тип возвращаемого функцией значения не учитывается (за исключением виртуальной перегрузки методов класса).

Перегрузка функций иногда называют параметрическим полиморфизмом.

8.1.2. Шаблоны функций: полиметрический полиморфизм

В приведенном выше примере все три функции mymin имеют одинаковое тело и отличаются только типами формальных параметров и возвращаемого значения. В такой ситуации функцию mymin можно было описать один раз, используя механизм шаблонов функций или функции с шаблоном.

При использовании функций с шаблонами приведенный выше пример будет выглядеть так:

```
template<class Type>
Type mymin(const Type& a, const Type& b)
{
if ( a < b )
return a;
else
     return b;
}
//--------------------------------------------------
int main()
{
```

```
int i1 = 4, i2 = 7, i3;
   i3 = mymin(i1, i2);

double d1 = 4.67, d2 = 1.89, d3;
   d3 = mymin(d1, d2);

char c1 = 5, c2 = 8, c3;
   c3 = mymin(c1, c2);
return 0;
}
```

В приведенном примере

1. Функция mymin описана с предзаголовком `template<class Type>`, в котором объявляется, что далее идет описание функции с шаблоном типа Type.
2. При компиляции функции main примера, компилятор, встретив первое обращение к функции mymin,
   a. определяет, что в этом месте функция вызывается в варианте: `int mymin(int, int);`
   b. «берет» описание функции с шаблоном Type, заменяет в тексте этого описания Type на int и компилирует полученную функцию. Эта операция называется **конкретизацией**, или **инстанцированием** шаблона функции.
3. При компиляции второго и третьего обращения к функции mymin компилятор поступает аналогичным образом.

Использование функций (и классов) с шаблонами называют полиметрическим полиморфизмом.

8.1.3. Шаблоны функций. Подробнее

Синтаксис объявления шаблона функции

В общем виде синтаксис объявления шаблона функции имеет вид:

```
template<class<тип_шаблон>>
<возвращаемый_тип><имя_функ>(<параметры>)
{<тело_функ>}
```

где   - имя «формального» типа, которое может использоваться в возвращаемом функцией типе, в объявлении формальных параметров и в теле функции.

В объявлении шаблона функции вместо ключевого слова class можно указывать ключевое слово typename:

```
template<typename<тип_шаблон>>
<возвращаемый_тип><имя_функ>(<параметры>)
{<тело_функ>}
```

Параметры шаблона функции

Объявление шаблона перекрывает глобально объявленный тип.

Пример:

```
typedefdouble Type;
template<class Type>
    Type mymin(const Type& a, const Type& b)    {
if ( a < b )
return a;
else
return b;
    }
```

В приведенном примере глобально объявленный тип Type перекрывается в функции шаблоне mymin, где под Type будет пониматься не double, а параметр шаблона.

Шаблон функции может иметь несколько параметров.

Пример:

```
template<class T1, typename T2>
T1 mymin(const T1& a, const T2& b)
{
if ( a < b )
return a;
else
return b;
}
//--------------------------------------------------
int main()
{
int i1 = 8, i2 = 7, i3;
double d1 = 4.67, d2 = 1.89, d3;
   i3 = mymin(i1, d1);
   d3 = mymin(d1, i2);

return 0;
}
```

В приведенном примере:

1. Шаблон функция mymin объявлена с двумя параметрами: T1 и T2.
2. При первой конкретизации в функции main она будет конкретизирована как

```
int mymin(constint&, constdouble&)
```

привтором – как:

```
doublemymin(constdouble&, constint&)
```

При вызове шаблона функции можно явно указывать типы аргументов в угловых скобках после имени функции. Функция main предыдущего примера могла выглядеть так:

```
int main()
{
int i1 = 8, i2 = 7, i3;
double d1 = 4.67, d2 = 1.89, d3;
   i3 = mymin<int, double>(i1, d1);
   d3 = mymin<double, int>(d1, i2);

return 0;
}
```

Шаблон функции может иметь в качестве параметра не только тип данных, но и обычные параметры.

Пример.

```
template<class T, int size>
T mymin(const T *a)
{
  T rez = a[0];
for (int i = 1; i < size; i++ )
if ( rez > a[i] ) rez = a[i];
return rez;
}
int main()
{
constint na = 5;
int* a = newint[na];
for ( int i = 0; i < na; ++i)
     a[i] = i;
constint nm = 8;
double* m = newdouble[nm];
for ( int i = 0; i < nm; ++i)
     m[i] = i*0.1;
int d = mymin<int, na>(a);
double dd= mymin<double, nm>(m);
return 0;
}
```

В приведенном примере шаблон функции mymin имеет два параметра: тип массива T и его длину size. При вызове такой функции (см. функцию main) надо явно указывать значения параметров шаблона.

Класс как параметр шаблона

Если в качестве параметра шаблона передается класс, то в нем должны быть перегружены операции, применяющиеся в шаблоне функции.

Пример:

```
class Complex
{
private:
double re, im;
public:
    Complex(double _re = 0.0, double _im = 0.0);
booloperator<(const Complex& c) const;
};
Complex::Complex(double _re, double _im)
{
    re = _re;
    im = _im;
}
bool Complex::operator<(const Complex& c) const
{
return (re*re + im*im) < (c.re*c.re + c.im*c.im);
}
template<class Type>
Type mymin(const Type& a, const Type& b)
//Type mymin(Type& a, const Type& b)
{
if ( a < b )
return a;
else
return b;
}
//-----------------------------------------------
int main()
{
int i1 = 4, i2 = 7, i3;
    i3 = mymin(i1, i2);
double d1 = 4.67, d2 = 1.89, d3;
    d3 = mymin(d1, d2);
    Complex c1(3.4, 1.1), c2(1.2, 0.9), c3;
    c3 = mymin(c1, c2);

return 0;
}
```

В приведенном примере функция шаблон mymin применяется для объектов класса Complex, в котором перегружена использующаяся в функции шаблоне операция сравнения.

Перегрузка шаблонов функций

Шаблоны функций могут быть перегружены как при помощи обычных функций, так и при помощи шаблонов.

Пример:

```
template<class T>
T mymin(const T& a, const T& b)
{
if ( a < b ) return a; elsereturn b;
}
template<class T1, class T2>
T1  mymin(const T1& a, const T2& b)
{
if ( a < b ) return a; elsereturn b;
}
int mymin(int a, int b)
{
if ( a < b ) return a; elsereturn b;
}
```

В приведенном примере шаблон функции mymin для одного параметра перегружен шаблоном функции с двумя параметрами и обычной функцией . При наличии перегрузки функции шаблона обычной функцией предпочтение отдается этой функции.

Специализация шаблона функции

При необходимости шаблон функции может быть специализирован. Специализация шаблона функции в описании шаблона функции для конкретного типа данных. Необходимость в специализации шаблона возникает в случае, когда для некоторого типа данных необходимы изменения в описании алгоритма шаблона функции.

Рассмотрим эту ситуацию на следующем примере:

```
template<class T>
T mymax( T p1, T p2)
{
return p1 > p2 ? p1 : p2;
}

int main()
{
int c = mymax(3, 4);
char s2[] = "Валя";
char s1[] = "Вова";
char* s = mymax(s1, s2);
```

```
}
```

В приведенном примере описан шаблон функции mymax, возвращающей максимальное из двух полученных параметров. В функции main примера первое обращение к функции приведет к правильному результату. Второе обращение к функции даст неверный результат потому, что тип передаваемых параметров (s1 и s2) – указатели на char; соответственно тип формальных параметров (p1и p2) конкретизированной функции mymax будет тоже указатели на char. В результате сравнение p1 > p2 будет не сравнением строк, а сравнением значений указателей (т.е. адресов строк).

Т.е. для случая, когда шаблону функции mymax будут передаваться строки (указатели на char) алгоритм этой функции должен быть другим. Это может быть задано дополнительным описанием специализации шаблона для этого типа данных:

```
typedefchar* CCP;
template<>
CCP mymax<CCP>( CCP s1, CCP s2)
{
return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

Комментарии:

1. Заголовок специализации шаблона имеет вид:

```
template<>
<тип> mymax<тип>( <параметры_функции> )
```

где тип – это конкретный тип, под который описывается специализация шаблона.

2. Тип специализации шаблона должен быть простым (не составным). Поэтому в приведенном примере для указателя на char объявлен синоним типа CPP.

8.1.4. Модели компиляции шаблонов

Как отмечалось, шаблон функции конкретизируется и компилируется в тот момент, когда компилятор встречает вызов шаблона функции с конкретными типами параметров шаблона. Это значит, что в момент встречи вызова компилятор должен видеть описание шаблона.

В C++ используется две модели компиляции шаблонов: компиляция с включением и компиляция с разделением.

Компиляция с включением

При использовании модели компиляции с включением описания шаблонов включается в заголовочные файлы. Подключение такого заголовочного файла к исполняемому файлу, в котором используются описанные в нем шаблоны функций обеспечивает видимость компилятором этих описаний.

Компиляция с включением имеет два существенных недостатка.

Первый недостаток состоит в неоправданном росте размеров заголовочных файлов, в которые принято включать только объявления. Избавиться от этого недостатка можно так:

- в заголовочном файле (filename.h) оставить только объявления шаблонов функций;
- описания шаблонов функций поместить в файл с именем filename.cpp;
- в конец заголовочного файла добавить директиву #include " filename.cpp ".

Второй недостаток компиляции с включением связан с тем, что конкретизация и компиляция шаблонов функции выполняется независимо для каждого файла программы. Если вызов шаблона функции для одной и той же конкретизации будет встречаться в нескольких файлах программы, то такая конкретизация и компиляция шаблона будет выполняться несколько раз.

Избежать такого дублирования можно выполнив в одном (и только в одном) из файлов **явное объявление конкретизации** шаблона:

```
templateint min<int>(int a, int b);
```

В этом случае объявленная конкретизация и компиляция шаблона будет выполняться один раз.

Компиляция с разделением

При использовании модели компиляции с разделением объявления шаблонов включаются в заголовочные файлы в виде:

```
template<class T> min(T a, T b);
```

а описания – в исполняемые файлы в виде:

```
exporttemplate<class T> min(T a, T b) { . . . }
```

Компиляция с разделением является более привычной, но реализована не во всех компиляторах (в MS Visual Studio не работает).

8.1.5. Примеры шаблонов функций

Шаблон функции сортировки

Операция сортировки является одной из наиболее часто применяемых. Сортировать приходится массивы самых разных типов. Поэтому целесообразно иметь шаблон функции сортировки:

```
template<class T>
void SortAny(T* a, int size)
{
   T tmp;
for (int i = 0; i < size - 1; i++ )
for (int j = i + 1; j < size; j++ )
if ( a[i] >a[j] )
         {
tmp  = a[i];
a[i] = a[j];
            a[j] = tmp;
         }
}
```

Единственное ограничение такого шаблона состоит в том, что для сортировки массива объектов некоторого класса необходимо, чтобы в этом классе были перегружены операции присваивания и сравнения на больше.

Шаблон функций создания и освобождения двумерных массивов

Еще одной часто выполняемой операцией является операция по созданию в куче двумерного массива указанной размерности и типа и освобождение памяти, занимаемой ранее созданным массивом.

Шаблоны функций создания и освобождения двумерных массивов выглядят так:

```
// Создание двумерного динамического массива -----------
template<class T> T** newA2T(int n, int m)
{
   T **ms = new T*[n];
for ( int i = 0; i < n; i++ )
      ms[i] = new T[m];
return ms;
}
// Освобождениедвумерногодинамическогомассива -------
template<class T>void delA2T(T **ms, int n)
{
```

```
for ( int i = 0; i < n; i++ )
delete[] ms[i];
delete[] ms;
}
```

Пример их использования так:

```
int main()
{
int n1, n2;
   cout <<"n1 = "; cin >> n1;
cout <<"n2 = "; cin >> n2;
double **md = newA2T<double>(n1, n2);  // Созданиемассива
int    **mi = newA2T<int>   (n1, n2);  // Создание массива
//. . . . . . . . . . . . . . . . . . . . . . .
   delA2T(md, n1); // Освобождение массива
   delA2T(mi, n1); // Освобождение массива
return 0;
}
```

8.2. Шаблоны классов

8.2.1. Объявление и описание шаблонов классов

Аналогично шаблонам функций в языке C++ могут использоваться шаблоны классов (классов с шаблонами). В отношении шаблонов классов действуют те же общие правила, что и для шаблонов функций:

1. Шаблон класса имеет один или несколько параметров (формальных типов), используемых при объявлении класса и описании его методов.
2. Параметром шаблона класса может быть и переменная конкретного типа.
3. Компиляция шаблонов классов выполняется после их конкретизации (инстанцирования) при объявлении объектов класса шаблона.
4. Объявление объектов класса шаблона должно обязательно включать объявление конкретных типов параметров шаблона.
5. Компиляция шаблонов классов выполняется с использованием тех же моделей компиляции, что и у шаблонов функций.
6. Шаблоны классов могут быть специализированы.
7. Классы с шаблонами могут наследоваться.

При этом, объявление и описание методов шаблонов классов имеют следующие особенности.

Объявление шаблона класса

Объявление шаблона класса имеет вид:

```
template<class T, typename T1>
class A
{
// . . . . .
};
```

Параметры шаблона T и T1 используются при объявлении полей и методов класса как типы полей и типы параметров методов шаблона класса.

Пример:

```
template<class T>
class A
{
protected:
     T data;  // данныетипа T
public:
A();
     A(T _data);
     A(const A& _a);
     ~A();
     A f(A& _a);
A&operator=(constA& _a);
//. . . . . . . . . . . . . . . . .
};
```

Описание методов шаблона класса

Описание методов шаблона класса включается в заголовочный файл (см. Модели компиляции шаблонов).

Описание каждого метода шаблона класса имеет вид:

```
template<class T, typename T1>
<возвращаемый_тип> A<T, T1>:: <имя_метода>(<параметры_метода>)
{
//. . . .
}
```

В описании методов шаблона класса все упоминания имени класса (кроме имен конструкторов и деструктора) должны включать параметры шаблона класса:

A<T, T1>   A<T, T1>*   A<T, T1>&

Пример:

```
//---------------------------------------
```

```
template<class T>
A<T>::A(T _data): data(_data) { }
//-------------------------------------
template<class T>
A<T>::A(const A<T>& _a) { data = _a.data; }
//-------------------------------------
template<class T>
A<T> A<T>::operator+(const A<T>& _a)
{
   A<T> tmp;
   tmp.data = data + _a.data;
return tmp;
}
//-------------------------------------
template<class T> A<T> A<T>::f(A<T>* _a)
{
//...................
}
```

Объявление объектов шаблона класса

Объявление объектов шаблона класса должно обязательно включать конкретные типы параметров шаблона:

A<type1, type2> a;

Пример:

```
int main()
{
   A<int> ai1, ai2(5);
   A<double> ad1, ad2(7.);
   A<Comlpex> ac1, ac2(3., -1.7);
}
```

8.2.2. Пример. Класс контейнер с шаблоном

Понятие контейнера

Под контейнером обычно понимается набор, как правило, однотипных элементов. Отличие контейнера от обычного массива состоит в том, что:

1. Контейнеры могут быть фиксированными (размер устанавливается при создании и изменяться быть не может) или динамическими (размер может меняться в процессе работы с контейнером).
2. Контейнеры могут быть сортированными, несортированными.
3. В контейнере могут быть реализованы следующие операции:

- добавления (в конец), удаления, вставки элементов в указанное место;

- доступ к элементам контейнера может быть прямой (по индексу элемента), последовательный (перейти к первому, последнему, следующему, предыдущему элементу), ассоциативный (по указанию одного из признаков элемента);
- присваивание, объединение, пересечение, разность контейнеров (в теоретико-множественном смысле).

Объявление шаблона класса динамического контейнера

Объявление шаблона класса динамического контейнера выглядит так:

```
enumTContainerExeption { cntINDOUTOFRANGE, cntVALOUTOFRANGE };
template<classTELEM>
classTContainer
{
protected:
TELEM* elem; // массивхранимыхэлементов
int size;    // размер массива
int count;   // к-во реально хранимых элементов
conststaticint sizestep = 10; // шагнаращиванияразмерамассива

public:
   TContainer(int _size = 50);
   TContainer(const TContainer& _cnt);
   ~TContainer();

constint&Count() const { return count; }
int&Count()        { return count; }
constint& Sizestep() const { return sizestep; }

void Add(TELEM _elm);    // добавить элемент
void Del(TELEM _elm);    // удалить элемент по его значению
void Del(int index);     // удалить элемент по его номеру
TELEM&operator[](intindex        ); // индексацияпономеру
TELEM&operator[](constTELEM& _elm); // индексацияпосодержанию
const TELEM&operator[](int index        ) const; //
индексацияпономеру
const TELEM&operator[](const TELEM& _elm) const; //
индексацияпосодержанию

   TContainer&operator=(const TContainer& _cnt); //
присваиваниеконтейнеров
   TContainer  operator+(const TContainer& _cnt); //
объединениеконтейнеров
   TContainer  operator*(const TContainer& _cnt); //
пересечениеконтейнеров
```

```
   TContainer  operator-(const TContainer& _cnt); //
разностьконтейнеров

protected:
void resize(int dsize = 0);       // увеличитьдлинуконтейнера
int _find(const TELEM& _elm) const;    // найтиэлемент
};
```

Здесь объявлены:

1. Указатель на массив типа параметр шаблона elem; размер (размер буфера) этого массива size; количество реально хранимых элементов count и шаг наращивания размера этого массива sizestep при необходимости увеличения его размера.
2. Конструктор инициализатор и конструктор копирования.
3. Методы доступа к количеству элементов контейнера и шагу наращивания его длины.
4. Метод Add добавления нового элемента в контейнер; два метода Del удаления элемента из контейнера.
5. Перегружены операции: индексации, присваивания, объединения, пересечения и разности контейнеров.
6. В защищенном режиме объявлены методы увеличения длины контейнера и поиска номера элемента контейнера по его значению.

Описание методов шаблона класса динамического контейнера

Надеемся, что описания конструкторов и деструктора в комментариях не нуждаются:

```
template<class TELEM>
TContainer<TELEM>::TContainer(int _size) : size(_size), count(0)
{
   elem = new TELEM[size];
}
//------------------------------------------------------------------
--------------------
template<class TELEM>
TContainer<TELEM>::TContainer(const TContainer<TELEM>& _cnt)
{
   size  = _cnt.size;
   count = _cnt.count;
   elem = new TELEM[size];
for ( int i = 0; i < count; i++ )
      elem[i] = _cnt.elem[i];
}
```

```
//-----------------------------------------------------------------
--------------------
template<class TELEM>
TContainer<TELEM>::~TContainer()
{
if ( size > 0 ) {
delete[] elem;
      elem  = 0;
count = 0;
size  = 0;
}
}
```

Далее рассмотрим описание метода увеличения длины контейнера:

```
template<class TELEM>
void TContainer<TELEM>::resize(int dsize)  //
увеличитьдлинуконтейнера
{
if ( dsize == 0) dsize = sizestep;
   size  += dsize;
   TELEM *tmp = new TELEM[size];
for ( int i = 0; i < count; i++ )
tmp[i] = elem[i];
delete [] elem;
elem = tmp;
}
```

В этом методе:

1. Получает dsize – размер, на который надо увеличить длину контейнера.
2. Если этот размер равен 0, то будет использоваться значение sizestep.
3. Размер контейнера увеличивается на dsize и в куче под tmp заказывается память под увеличенную длину контейнера.
4. Старое содержание контейнера переписывается в tmp.
5. Память, занятая старым содержанием контейнера освобождается и elem присваивается значение tmp.

Метод поиска элемента контейнера возвращает номер этого элемента или -1, если искомого элемента в контейнере нет:

```
template<class TELEM>
int  TContainer<TELEM>::_find(const TELEM& _elm) const//
найтиэлемент
{
int nom = -1;
int i = 0;
while ( i < count && nom == -1 )
```

```
if ( elem[i] == _elm ) nom = i; else i++;
return nom;
}
```

Метод добавления элемента в контейнер:

```
template<class TELEM>
void TContainer<TELEM>::Add(TELEM _elm)      // добавитьэлемент
{
if ( _find(_elm) == -1)
   {
if ( size == count ) resize();
elem[count++] = _elm;
   }
}
```

Элементы контейнера не должны дублироваться. Поэтому, в этом методе:

1. Выполняется поиск добавляемого элемента в контейнере и, если его там нет:
2. Выполняется проверка: не заполнен ли контейнер и, если да, то размер контейнера увеличивается на стандартную длину.
3. Элемент добавляется в конец контейнера.

Методы удаления элемента контейнера по номеру и по элементу:

```
template<class TELEM>
void TContainer<TELEM>::Del(int index)    // удалитьэлемент
{
if ( index > -1 && index < count )
for ( int i = index; i < count - 1; i++ )
         elem[i] = elem[i + 1];
   count--;
}
//------------------------------------------------------------------
--------------------
template<class TELEM>
void TContainer<TELEM>::Del(TELEM _elm)    // удалитьэлемент
{
   Del(_find(_elm)); // удалить удаляемый элемент
}
```

При удалении элемента по номеру:

1. Выполняется проверка: правильный ли указан номер и, если да, то:
2. Выполняется сдвиг «хвоста» контейнера и количество его элементов уменьшается на единицу.

Удаление элемента по его значению выполняется с помощью поиска этого элемента в контейнере и его удалению по найденному номеру.

Перегрузка операции индексации по индексу:

```
template<class TELEM>
TELEM& TContainer<TELEM>::operator[](int index)
{
if ( index < 0 || index >= count )
   {
      TContainerExeption expt = cntINDOUTOFRANGE;
throw expt;
   }
return elem[index]; // типвозвращаемогозначения
}
```

Перегрузка операции индексации по индексу начинается с проверки принадлежности индекса диапазону. Если значение индекса выходит за пределы диапазона, то возбуждается исключение `cntINDOUTOFRANGE`. Если нет, то возвращается запрашиваемый элемент контейнера.

Перегрузка операции индексации по элементу контейнера:

```
template<class TELEM>
TELEM& TContainer<TELEM>::operator[](const TELEM& _elm)
{
int index;
if ( (index = _find(_elm)) == -1 )
   {
      TContainerExeption expt = cntVALOUTOFRANGE;
throw expt;
}
return elem[index];    // тип возвращаемого значения
}
```

Перегрузка операции индексации по элементу контейнера начинается с поиска этого элемента в контейнере. Если этого элемента в контейнере нет (переменная index получает значение -1), то возбуждается исключение `cntVALOUTOFRANGE`. Если есть, то возвращается запрашиваемый элемент контейнера.

На первый взгляд перегрузка операции индексации по элементу контейнера смысла не имеет, т.к. выполняет запрос элемента контейнера по самому этому элементу. Это так для контейнеров, элементы которых имеют простой тип: int, double, … Для контейнеров, элементы которых являются объектами сложных классов картина меняется.

Описание перегрузки операции присваивания имеет вид:

```
template<class TELEM>
```

```
TContainer<TELEM>& TContainer<TELEM>::operator=(const
TContainer<TELEM>& _cnt) // присваиваниеконтейнеров
{
if ( this != &_cnt ) {
delete [] elem;;
      size  = _cnt.size;
      count = _cnt.count;
      elem = new TELEM[size];
for ( int i = 0; i < count; i++ )
         elem[i] = _cnt.elem[i];
}
return *this;
}
```

Надеемся, что оно достаточно понятно.

Описание перегрузки операции объединения контейнеров имеет вид:

```
template<class TELEM>
TContainer<TELEM>  TContainer<TELEM>::operator+(const
TContainer<TELEM>& _cnt) // объединениеконтейнеров
{
int i;
   TContainer<TELEM> tmp(count + _cnt.count + sizestep);
for ( i = 0; i < count; i++ )
      tmp.elem[i] = elem[i];
   tmp.count = count;
for ( i = 0; i < _cnt.count; i++ )
      tmp.Add(_cnt.elem[i]);
return tmp;
}
```

Здесь:

1. Объявляется объединенный контейнер tmp, размер которого равен сумме длин объединяемых контейнеров с запасом sizestep.
2. Элементы первого из объединяемых контейнеров присваиваются контейнеру tmp.
3. Элементы второго из объединяемых контейнеров добавляются с помощью операции Add, чтобы исключить дублирование элементов в объединенном контейнере.

Описание перегрузки операции пересечения контейнеров имеет вид:

```
template<class TELEM>
TContainer<TELEM>  TContainer<TELEM>::operator*(const
TContainer<TELEM>& _cnt) // пересечениеконтейнеров
{
int i;
int ss = count;
```

```
if ( ss < _cnt.count ) ss = _cnt.count;
   TContainer<TELEM> tmp(ss + sizestep);
   tmp.count = 0;
for ( i = 0; i < count; i++ )
if ( _cnt._find(elem[i]) != -1 ) tmp.elem[tmp.count++] = elem[i];
return tmp;
}
```

Здесь:

1. Объявляется «пересеченный» контейнер tmp, размер которого равен минимальной длине пересекаемых контейнеров с запасом sizestep.
2. В контейнер tmp добавляются те элементы первого контейнера, которые есть во втором.

Описание перегрузки операции разности контейнеров имеет вид:

```
template<class TELEM>
TContainer<TELEM>  TContainer<TELEM>::operator-(const
TContainer<TELEM>& _cnt) // разностьконтейнеров
{
int i;
   TContainer<TELEM> tmp(count + sizestep);
   tmp.count = 0;
for ( i = 0; i < count; i++ )
if ( _cnt._find(elem[i]) == -1 ) tmp.elem[tmp.count++] = elem[i];
return tmp;
}
```

Здесь:

1. Объявляется «разностный» контейнер tmp, размер которого равен длине первого контейнера с запасом sizestep.
2. В контейнер tmp добавляются те элементы первого контейнера, которых нет во втором.

8.2.3. Специализация шаблона класса

Специализацию шаблона класса будем рассматривать на следующем примере. Пусть в заголовочном файле ATemp.h содержится следующий код:

```
#ifndef HATEMP
#define HATEMP
template<class T>class A
{
   T a;
public:
   A(T _a): a(_a) { }
   T amin(const A<T>& _a);
```

```
   T amax(const A<T>& _a);
};
template<class T> T A<T>::amin(const A<T>& _a)
{
if ( a < _a.a ) return a; elsereturn _a.a;
}
template<class T> T A<T>::amax(const A<T>& _a)
{
if ( a > _a.a ) return a; elsereturn _a.a;
}
#endif
```

В этом файле объявлен и описан класс A с шаблоном T. Класс имеет одно поле a типа T, конструктор инициализатор и два метода определения минимального и максимального значений.

Клиент этого шаблона класса (файл TempTest.cpp) может быть таким:

```
#include"ATemp.h"
int main()
{
   A<double> d1(1.2), d2(3.7);
double dd1 = d1.amax(d2);
double dd2 = d1.amin(d2);

   A<int> i1(1), i2(3);
int ii1 = i1.amax(i2);
int ii2 = i1.amin(i2);

   A<char> c1(1), c2(3);
char cc1 = c1.amax(c2);
char cc2 = c1.amin(c2);

return 0;
}
```

При компиляции функции main компилятором будет выполнено три конкретизации шаблона класса A: для double, int и char. Будет создано три варианта класса A, которые будут использоваться при вызове методов amax и amin соответствующих объектов.

Специализация шаблона класса может выполняться как для отдельных методов класса, так и для всего класса.

Специализация методов класса

Для выполнения специализации метода amin шаблона класса A для типа данных int надо в проект добавить два файла:

1. Заголовочный файл AIntSp.h:

```
#ifndef HAINTSP
#define HAINTSP
#include"ATemp.h"
// явная специализация метода amin класса A<T> ---------------
template<>int A<int>::amin(const A<int>& _a);
#endif
```

2. Исполняемый файл AIntSp.cpp:

```
#include"AIntSp.h"
// явная специализация метода amin класса A<T> ---------------
int A<int>::amin(const A<int>& _a)
{
if ( a < _a.a ) return a; elsereturn _a.a;
}
```

Если теперь в файл TempTest.cpp добавить директиву #include "AIntSp.h", то при компиляции функции main для случаев double и char будет по-прежнему выполнены две конкретизации класса A. Для случая int:

- так же будет выполнена конкретизация класса A, через которую будет работать метод i1.amax(i2);
- для метода i1.amin(i2) будет выполнена компиляция специализации этого метода и будет вызван специализированный метод.

Специализация класса

Для выполнения специализации шаблона класса А для типа данных char надо в проект добавить два файла:

1. Заголовочный файл ACharSp.h:

```
#ifndef HACHAR
#define HACHAR
#include"ATemp.h"
template<>class A<char>
{
char a;
public:
   A(char _a): a(_a) {}
char amin(const A<char>& _a);
char amax(const A<char>& _a);
};
#endif
```

2. Исполняемый файл ACharSp.cpp:

```
#include"ACharSp.h"
char A<char>::amin(const A<char>& _a)
{
if ( a < _a.a ) return a; elsereturn _a.a;
}
char A<char>::amax(const A<char>& _a)
{
if ( a > _a.a ) return a; elsereturn _a.a;
}
```

Если теперь в файл TempTest.cpp добавить директиву #include "ACharSp.h", то при компиляции функции main для случаев double и int будет по-прежнему выполнены две конкретизации класса А и одна специализация (как в разделе «Специализация методов класса»). Для случая char будет выполнена компиляция специализации класса А и вызваны методы этой специализации.

СПИСОК ЛИТЕРАТУРЫ

1. Подбельский В. В. Язык Си++: Учебное пособие. — 5-е изд., дораб. — М.: Финансы и статистика, 2006. — 560 с.
2. Липпман С. Основы программирования на С++. Вводный курс. Пер. с англ. — М.: Вильямс, 2002.
3. Страуструп Б. Язык программирования Си++. Специальное издание. Пер. с англ. — М.: ООО «Бином-Пресс», 2004. — 1104 с.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3

1. Введение в ООП 4

1.1. Парадигмы ООП 4
1.1.1. Парадигмы программирования 4
1.1.2. Парадигмы ООП 5
1.2.Как работают парадигмы ООП 9
1.2.1. Постановка задачи 9
1.2.2. Процедурное решение задачи 9
1.2.3. Решение в рамках ООП 12

2. Классы и объекты 15

2.1. Класс: объявление и описание 15
2.1.1. Объявление класса 15
2.1.2. Описание класса 16
2.2. Объект: объявление и использование 17
2.2.1. Объявления объектов 17
2.2.2. Взаимодействие объектов и методов 18
2.3. Сокрытие членов класса 19
2.3.1. Секции private и public 19
2.3.2. Что и как скрывать в объявлении класса 21
2.3.3. Описание методов в объявлении класса 22
2.3.4. Классы и структуры 23
2.4. Примеры 23
2.4.1. Класс Ccomplex 23
2.4.2. Класс Cmystring 25

3. Конструкторы и деструктор класса 29

3.1. Введение 29
3.2. Что такое конструктор? 29
3.3. Типы конструкторов 30
3.4. Деструктор класса 32
3.5. Когда нужны конструкторы и деструктор 34
3.6. Когда и как вызываются конструкторы и деструктор 34

4. Обработка исключений 36

4.1. Ошибки и их обработка 36
4.1.1. Код возврата 36
4.1.2. Код завершения 37
4.1.3. Обработка исключений 38
4.2. Возбуждение и обработка исключений 39
4.2.1. Возбуждение исключения 39
4.2.2. Обработка исключений 40
4.2.3. Выход из обработчика catch 41
4.3. Пример 41

5. Перегрузка операций 43

5.1. Общие правила перегрузки операций 43
5.1.1. Перегрузка операции - функция 43
5.1.2. Два способа перегрузки операций 43
5.1.3. Общие правила и ограничения перегрузки операций 44
5.2. Правила перегрузки отдельных типов операций 46
5.2.1. Перегрузка операции «=» (присваивания) 46
5.2.2. Перегрузка операций типа «+» 46
5.2.3. Перегрузка операций типа «+=» 47
5.2.4. Перегрузка операций сравнения (отношения) 47
5.2.5. Перегрузка операции «[]» (индексации) 48
5.2.6. Перегрузка унарных операций 49
5.2.7. Перегрузка операций обмена с потоком 50

6. Наследование 52

6.1. Наследование классов 52
6.1.1. Что такое наследование классов 52
6.1.2. Перегрузка и перекрытие методов 53
6.1.3. Спецификатор доступа protected 55
6.2. Спецификаторы доступа при наследовании 55
6.2.1. Доступ при наследовании 55
6.2.2. Взаимодействие спецификаторов доступа при наследовании 57
6.3. Конструирование и деструктурирование потомков 58
6.3.1. Конструирование потомков 58
6.3.2. Деструктурирование потомков 60
6.4. Пример 62
6.4.1. Класс Triad 62
6.4.2. Класс Time 63
7. Виртуальные методы и абстрактные классы 66
7.1. Понятие виртуального метода 66
7.1.1. Что такое виртуальный метод? 66
7.1.2. Особенности виртуального перекрытия методов 67
7.1.3. Механизм позднего связывания 68
7.1.4. Что дает использование виртуальных методов? 70
7.2. Абстрактные методы и классы 73
7.2.1. Понятие абстрактного метода и класса 73
7.2.2. Свойства абстрактных методов и классов 75
7.2.3. Пример классификации 75
8. Введение в шаблоны 78
8.1. Шаблоны функций 78
8.1.1. Перегрузка функций: параметрический полиморфизм 78
8.1.2. Шаблоны функций: полиметрический полиморфизм 79
8.1.3. Шаблоны функций. Подробнее 80
8.1.4. Модели компиляции шаблонов 85
8.1.5. Примеры шаблонов функций 87
8.2. Шаблоны классов 88
8.2.1. Объявление и описание шаблонов классов 88
8.2.2. Пример. Класс контейнер с шаблоном 90
8.2.3. Специализация шаблона класса 97
Список литературы 101

Сергей Николаевич Карпенко

Основы объектно-ориентированного программирования на языке C++

Учебно-методическое пособие

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать ____________. Формат ____________.

Электронная версия