Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»
(ТУСУР)

Кафедра моделирования и системного анализа

Ганджа Т.В., Панов С.А.

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Курс лекций

Томск
2015

Ганджа Т.В., Панов С.А. Объектно-ориентированное программирование / Курс лекций – Томск: Томский государственный университет систем управления и радиоэлектроники, Кафедра моделирования и системного анализа, 2015. – 101 с.

**Содержание**

# 1. Структурное программирование

## 1.1 Структура программы

Программа на языке C++ состоит из функций, описаний и директив препроцессора. Одна из функций должна иметь имя *main*. Выполнение программы начинается с первого оператора этой функции. Простейшее определение функции имеет следующий формат:

```
тип_возвращаемого_значения имя([параметры])
{
    операторы, составляющие тело функции
}
```

Как правило, функция используется для вычисления какого-либо значения, поэтому перед именем функции указывается его тип:

- если функция не должна возвращать значение, указывается тип void (подробнее типы данных рассмотрим на следующей лекции);
- тело функции является блоком и, следовательно, заключается в фигурные скобки;
- функции не могут быть вложенными;
- каждый оператор заканчивается точкой с запятой (кроме составного оператора).

Пример структуры программы, содержащей функции *main, fl* и *f2*:

Листинг 1.1. Структуры программы

```
директивы препроцессора
описания
int main()
{
    операторы главной функции
}
int fl()
{
    операторы функции fl
}
int f2()
{
    операторы функции f2
}
```

Программа может состоять из нескольких модулей (исходных файлов).

В языке C++ нет встроенных *средств ввода/вывода* - он осуществляется с помощью функций, типов и объектов, содержащихся в стандартных библиотеках. Используется два способа: функции, унаследованные из языка C, и объекты C++. Основные функции ввода/вывода в стиле C:

```
int scanf (const char* format. ... )                    // ввод
int printf(const char* format, ... )                    // вывод
```

Они выполняют форматированный ввод и вывод произвольного количества величин в соответствии со строкой формата *format*. Строка формата содержит символы, которые при выводе копируются в поток (на экран) или запрашиваются из потока (с клавиатуры) при вводе, и спецификации преобразования, начинающиеся со знака %, которые при вводе и выводе заменяются конкретными величинами.

Листинг 1.2. Использование стандартных операторов ввода/вывода языка C

```
#include <stdio.h>
int main()
{
    int i:
    print("Введите целое число\n");
    scanf("Id", &i);
    printf("Вы ввели число %i, спасибо!", i);
    return 0;
}
```

Первая строка этой программы - директива препроцессора, по которой в текст программы вставляется заголовочный файл *<stdio.h>*, содержащий описание использованных в программе функций ввода/вывода (в данном случае угловые скобки являются элементом языка). Все директивы препроцессора начинаются со знака #. Директивы препроцессора будут рассмотрены в пункте 2.2.

Третья строка - описание переменной целого типа с именем *i*.

Функция *printf* в четвертой строке выводит приглашение «Введите целое число» и переходит на новую строку в соответствии с управляющей последовательностью \n. Функция *scanf* заносит введенное с клавиатуры целое число в переменную *i* (знак & означает операцию получения адреса), а следующий оператор выводит на экран указанную в нем строку, заменив спецификацию преобразования значением этого числа.

Листинг 1.3. Операции ввода/вывода языка C++

```
#include <iostream.h>
int main()
{
      int i;
      cout « "Введите целое число\n";
      cin » i;
      cout « "Вы ввели число " « i « ". спасибо!";
      return 0;
}
```

Заголовочный файл *<iostream.h>* содержит описание набора классов для управления вводом/выводом. В нем определены стандартные объекты-потоки *cin* для ввода с клавиатуры и *cout* для вывода на экран, а также операции помещения в поток *<< чтения из потока >>*.

## 1.2 Операции

### 1.2.1 Инкремент и декремент

Операции увеличения и уменьшения значения переменной на единицу, называемые также инкрементом и декрементом, имеют две формы записи — *префиксную*, когда операция записывается перед операндом, и ***постфиксную***, когда операция записывается после операнда. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Листинг 1.4. Использование операций инкремента и декремента

```
#include <std1o.h>
int main()
{
      int x = 3, y = 3:
      printf("Значение префиксного выражения: %d\n", ++x);
      printf ("Значение постфиксного выражения: %d\n", y++):
      printf("Значение x после приращения: %d\n". x);
      rint("Значение y после приращения: %d\n", y):
      return 0:
}
```

Результаты программы (листинг 1.4) :

```
Значение префиксного выражения: 4
Значение постфиксного выражения: 3
Значение x после приращения: 4
Значение y после приращения: 4
```

Операндом операции инкремента в общем случае является так называемое *L-значение* (L-value). Так обозначается любое выражение, адресующее некоторый участок памяти, в который можно занести значение. Название произошло от операции присваивания, поскольку именно ее левая (Left) часть определяет, в какую область памяти будет занесен результат операции. Переменная является частным случаем L-значения.

### 1.2.2 Операция определения размера sizeof

Операция определения размера sizeof предназначена для вычисления размера объекта или типа в байтах, и имеет две формы:

```
sizeof   выражение
sizeof ( тип )
```

Листинг 1.5. Пример использования операции sizeof

```
#indude <iostream.h>
int main()
{
      float x = 1;
      cout « "sizeof (float)  :" « sizeof (float);
      cout « "\nsizeof x :" « sizeof x;
      cout « "\nsizeof (x + 1.0)  :" « sizeof (x + 1.0):
      return 0;
}
```

Результаты работы программы, представленной на листинге 1.5.

```
sizeof (float)  : 4
sizeof x : 4
sizeof (x + 1.0)  : 8
```

Последний результат связан с тем, что вещественные константы по умолчанию имеют тип *double*, к которому, как к более длинному, приводится тип переменной *x* и всего выражения. Скобки необходимы для того, чтобы выражение, стоящее в них, вычислялось раньше операции приведения типа, имеющей больший приоритет, чем сложение.

### 1.2.3 Операции отрицания

Арифметическое отрицание (унарный минус -) изменяет знак операнда целого или вещественного типа на противоположный. Логическое отрицание (!) дает в результате значение 0, если операнд есть истина (не пуль), и значение 1, если операнд равен нулю. Операнд должен быть целого или вещественного типа, а может иметь также тип указатель. Поразрядное отрицание (-), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении целочисленного операнда.

### 1.2.4 Деление и остаток от деления

Операция деления применима к операндам арифметического типа. Если оба операнда целочисленные, результат операции округляется до целого числа, в противном случае тип результата определяется правилами преобразования. Операция остатка от деления применяется только к целочисленным операндам. Знак результата зависит от реализации.

Листинг 1.6. Использование операций деления и остаток от деления

```
#include <stdio.h>
int main
{
      int x = 11, y = 4:
      float z = 4;
      printf(“Результаты деления: %d   %f\n”, x/y. x/z);
      printf("Остаток: %d\n",x%y);
      return 0:
}
```

Результаты работы программы, представленной на листинге 1.6.

```
Результаты деления:  2   2.750000
Остаток:  3
```

### 1.2.5 Операции сдвига

Операции сдвига (<< и >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом. При *сдвиге влево* (<<) освободившиеся разряды обнуляются. При *сдвиге вправо* (>>) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае. Операции сдвига не учитывают переполнение и потерю значимости.

### 1.2.6 Операции отношения

Операции отношения (<. <=, >, >=, =, !=) сравнивают первый операнд со вторым. Операнды могут быть арифметического типа или указателями. Результатом операции является значение *true* или *false* (любое значение, не равное нулю, интерпретируется как true). Операции сравнения на равенство и неравенство имеют меньший приоритет, чем остальные операции сравнения.

### 1.2.7 Поразрядные операции

*Поразрядные операции (&, |, ^)* применяются только к целочисленным операндам и работают с их двоичными представлениями. При выполнении операций операнды сопоставляются побитово (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго, и т д.).

При *поразрядной конъюнкции*, или *поразрядном И* (операция обозначается &) бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При *поразрядной дизъюнкции*, или *поразрядном ИЛИ* (операция обозначается |) бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

При *поразрядном исключающем ИЛИ* (операция обозначается ^) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

Листинг 1.7. Пример использования поразрядных операций

```
#include <iostream.h>
int main
{
      cout « "\n 6 & 5 = " « (6 & 5);
      cout « "\n 6  |  5 = " « (6  | 5);
      cout « "\n б ^ 5 = " « (6 ^ 5);
      return 0;
}
```

Результат работы программы:

```
б & 5 = 4
6 | 5 = 7
б ^ 5 = 3
```

### 1.2.8 Логические операции

*Операнды логических операций И* (&&) и *ИЛИ* (||) могут иметь арифметический тип или быть указателями, при этом операнды в каждой операции могут быть различных типов. Преобразования типов не производятся, каждый операнд оценивается с точки зрения его эквивалентности нулю (операнд, равный нулю, рассматривается как *false*, не равный пулю — как *true*).

Результатом логической операции является *true* или *false*. Результат операции логическое *И* имеет значение *true* только если оба операнда имеют значение *true*. Результат операции логическое *ИЛИ* имеет значение *true*, если хотя бы один из операндов имеет значение *true*. Логические операции выполняются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

### 1.2.9 Операции присваивания

*Операции присваивания* могут использоваться в программе как закопченные операторы. Формат операции простого присваивания (=):

```
операнд_1 = операнд_2
```

Первый операнд должен быть *L-значением*, второй — выражением. Сначала вычисляется выражение, стоящее в правой части операции, а потом его результат записывается в область памяти, указанную в левой части (мнемоническое правило: «присваивание — это передача данных "налево"»). То, что ранее хранилось в этой области памяти, естественно, теряется.

Листинг 1.8. Использование операций присваивания

```
#include <iostream.h>
int main()
{
      int a = 3, b = 5, c = 7;
      a = b; b = a;4 c = c + 1:
      cout « "a = " « a;
      cout « "\t b = " « b:
      cout « "\t c = " « c;
      return 0;
}
```

Результат работы программы (листинг 1.8. )

```
a=5   b=5   c=8
```

*В сложных операциях присваивания ( +=, *=, /= и т п.)* при вычислении выражения, стоящего в правой части, используется и L-значение из левой части. Например, при сложении с присваиванием ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение a += b является более компактной записью выражения $a = a + b$.

### 1.2.10 Условная операция (? 7:)

Эта операция тернарная, то есть имеет три операнда. Ее формат:

```
операнд_1 ? операнд_2 : операнд_3
```

Первый операнд может иметь арифметический тип или быть указателем. Он оценивается с точки зрения его эквивалентности пулю (операнд, равный нулю, рассматривается как *false*, не равный нулю — как *true*). Если результат вычисления операнда 1 равен *true*, то результатом условной операции будет значение второго операнда, иначе — третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться. Условная операция является сокращенной формой условного оператора if.

Листинг 1.9. Использование условной операции

```
#include <stdio.h>
int main()
{
      int a = 11, b = 4, max;
      max = (b > a)? 6 : a;
      printf("Наибольшее число: %d", max);
      return 0;
}
```

Результат работы программы листинг 1.9. :

```
Наибольшее число: 11
```

Другой пример применения условной операции. Требуется, чтобы некоторая целая величина увеличивалась на 1, если ее значение не превышает *n*, а иначе принимала значение 1:

```
i = (i < n) ? i + 1: 1;
```

## 1.3 Выражения

Выражения состоят из операндов, знаков операций и скобок и используются для вычисления некоторого значения определенного типа. Каждый операнд является, в свою очередь, выражением или одним из его частных случаев — константой или переменной.

Примеры выражений:

```
(a + 0.12)/6
x && y || !z
(t*sin(x)-1.05e4)/((2*k+2)*(2*k+3))
```

Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, унарные операции, условная операция и операции присваивания выполняются справа налево, остальные — слева направо. Например, *a=b=c* означает *a=(b=c),* а *a+b+c* означает *(a+b)+c*. Порядок вычисления подвыражений внутри выражений не определен; например, нельзя считать, что в выражении *(sin(x+2)+cos(y)+1))* обращение к синусу будет выполнено раньше, чем к косинусу, и что *x + 2* будет вычислено раньше, чем *y+ 1*

Результат вычисления выражения характеризуется значением и типом. Например, если a и b - переменные целого тина и описаны так:

```
int a = 2, b = 5;
```

то выражение *a + b* имеет значение 7 и тип *int*, а выражение *a = b* имеет значение, равное помещенному в переменную *a* (в данному случае 5) и тип, совпадающий с типом этой переменной. Таким образом, в C++ допустимы выражения вида *a=b=c*: сначала вычисляется выражение *b=c*, а затем его результат становится правым операндом для операции присваивания переменной a.

В выражение могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат операции будет иметь тот же тип. Если операнды разного типа, перед вычислениями выполняются преобразования типов по определенным правилам, обеспечивающим преобразование более коротких типов в более длинные д3ля сохранения значимости и точности.

Преобразования бывают двух типов:

- изменяющие внутреннее представление величин (с потерей точности или без потери точности);
- изменяющие только интерпретацию внутреннего представления.

К первому типу относится, например, преобразование целого числа в вещественное (без потери точности) и наоборот (возможно, с потерей точности), ко второму — преобразование знакового целого в беззнаковое.

В любом случае величины типов *char, signed char, unsigned char, short int u unsigned short int* преобразуются в тип *int*, если он может представить все значения, или в *unsigned int* в противном случае.

После этого операнды преобразуются к типу наиболее длинного из них, и он используется как тип результата. Правила преобразований приведены в приложении 3.

Дополнительно о структуре программы и операциях языка C++ рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 59 – 60.

2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 17 - 47.

## 1.4 Базовые конструкции структурного программирования

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур, называемых следованием, ветвлением и циклом. Этот результат установлен Боймом и Якопини еще в 1966 году путем доказательства того, что любую программу можно преобразовать в эквивалентную, состоящую только из этих структур и их комбинаций.

Следование, ветвление и цикл называют базовыми конструкциями структурного программирования. Следованием называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных). Ветвление задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. Цикл задает многократное выполнение оператора (рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.1**). Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следова-

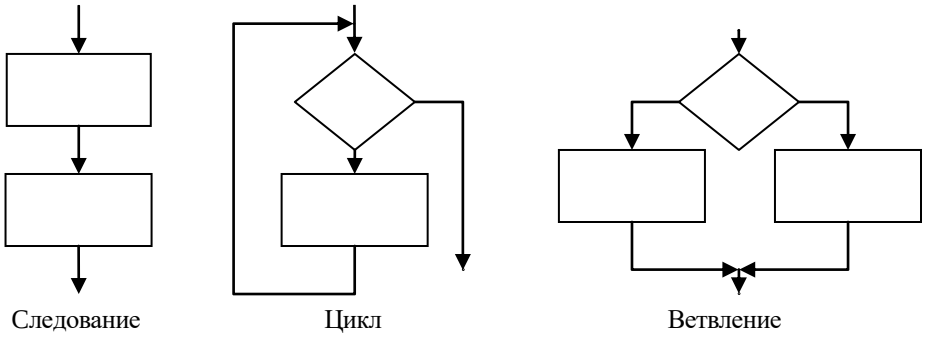


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.1. Базовые конструкции структурного программирования

ние из двух ветвлений, каждое из которых включает вложенные циклы.

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать (а программы чаще приходится читать, чем писать), отлаживать и при необходимости вносить в нее изменения. Структурное программирование часто называли «программированием без *goto*», и в этом есть большая доля правды: частое использование операторов передачи управления в произвольные точки программы затрудняет прослеживание логики ее работы. С другой стороны, никакие принципы нельзя возводить в абсолют, и есть ситуации, в которых использование *goto* оправдано и приводит, напротив, к упрощению структуры программы. О них говорится в разделе «Оператор *goto*»

В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в *C++* есть три вида циклов и два вида ветвлений (на два и на произвольное количество направлений). Они введены для удобства программирования, и в каждом случае надо выбирать наиболее подходящие средства. Главное, о чем нужно помнить даже при написании самых простых программ, — что они должны состоять из четкой последовательности блоков строго определенной конфигурации. «Кто ясно мыслит, тот ясно излагает» — практика давно показала, что программы в стиле «поток сознания» нежизнеспособны, не говоря о том, что они просто некрасивы.

Рассмотрим операторы языка, реализующие базовые конструкции структурного программирования.

### 1.4.1 Оператор «выражение»

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения. Частным случаем выражения явля-

ется пустой оператор ; (он используется, когда по синтаксису оператор требуется, а по смыслу — нет).

Листинг 1.10. Пример операторов «выражение»

```
i++;                    // выполняется операция инкремента
a* = b + c:             // выполняется умножение с присваиванием
fun(i,  k);             //выполняется вызов функции
```

### 1.4.2 Операторы ветвления

#### 1.4.2.1 Условный оператор if

Условный оператор *if* используется для разветвления процесса вычислений па два направления. Структурная схема оператора приведена на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2**. Формат оператора:

```
if (выражение) оператор_1;  [else оператор_2;]
```

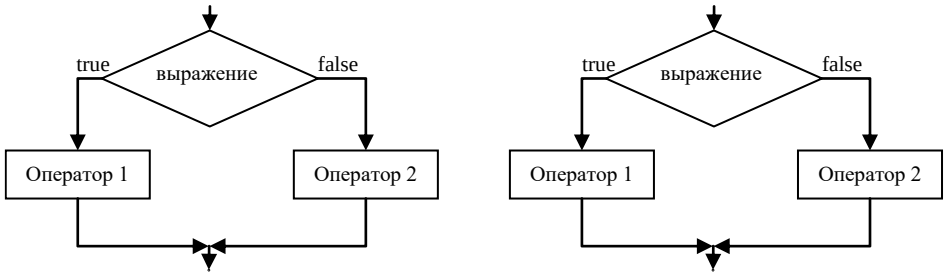


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2. Структурная схема условного оператора

Сначала вычисляется выражение, которое может иметь арифметический тип или тип указателя. Если оно не равно нулю (имеет значение *true*), выполняется первый оператор, иначе — второй. После этого управление передается на оператор, следующий за условным.

Одна из ветвей может отсутствовать, логичнее опускать вторую ветвь вместе с ключевым словом *else*. Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в блок, иначе компилятор не сможет попять, где заканчивается ветвление. Блок может содержать любые операторы, в том числе описания и другие условные операторы (но не может состоять из одних описаний). Необходимо учитывать, что переменная, описанная в блоке, вне блока не существует.

Листинг 1.11. Примеры использования условного оператора

```
if(a<0) b = 1;                                                                  //1
If(a<b && (a>d || a==0)) b++; else {b*=a; a=0;}                                 //2
If(a<b) {if (a<c) m = a; else m = c;} else {if(b<c) m = b; else m = c;}         //3
If(a++) b++                                                                     //4
If(b>a) max = b; else max = a;                                                  //5
```

В *примере 1* отсутствует ветвь *else*. Подобная конструкция называется «пропуск оператора», поскольку присваивание либо выполняется, либо пропускается в зависимости от выполнения условия.

Если требуется проверить несколько условий, их объединяют знаками логических операций. Например, выражение в *примере 2* будет истинно в том случае, если выполнится одновременно условие *a<b* и одно из условий в скобках. Если опустить внутренние скобки, будет выполнено сначала логическое *И*, а потом — *ИЛИ*.

Оператор *в примере 3* вычисляет наименьшее значение из трех переменных. Фигурные скобки в данном случае не обязательны, так как компилятор относит часть *else* к ближайшему *if*.

*Пример 4* напоминает о том, что хотя в качестве выражений в операторе *if* чаще всего используются операции отношения, это не обязательно.

Конструкции, подобные оператору *в примере 5*, проще и нагляднее записывать в виде условной операции (в данном случае: *max = (b > a) ? b : a;*).

Листинг 1.12. Выстрел по мишени

Производится выстрел по мишени, изображенной на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.3**. Определить количество очков.

```
#include <iostream.h>
int main()
{
      float x, y;
      int kol:
      cout « "Введите координаты выстрела \n";
      cin » x » y;
      if   ( x*x + y*y < 1 )
            kol = 2;
      else if(x*x + y*y < 4 )
                  kol = 1;
            else
                  kol =0;
      cout « "\n Очков" << kol:
      return 0:
}
```

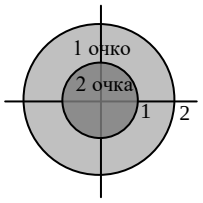


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.3. Мишень

Если какая-либо переменная используется только внутри условного оператора, рекомендуется объявить ее внутри скобок, например:

Листинг 1.13. Объявление переменной внутри блока

```
if (int i = fun(t))
      a -= i;
else
      a += i;
```

Объявление переменной в тот момент, когда она требуется, то есть когда ей необходимо присвоить значение, является признаком хорошего стиля и позволяет избежать случайного использования переменной до ее инициализации. Объявлять внутри оператора *if* можно только одну переменную. Область ее видимости начинается в точке объявления и включает обе ветви оператора.

### 1.4.2.2 Оператор switch

Оператор *switch* (переключатель) предназначен для разветвления процесса вычислений па несколько направлений. Структурная схема оператора приведена на рис. Error! Use the Home

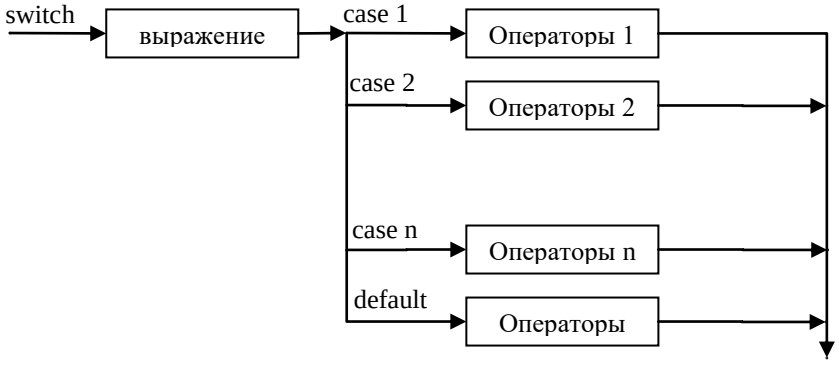


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.4. Структурная схема оператора switch

tab to apply Заголовок 1 to the text that you want to appear here..**4**.

Листинг 1.14. Формат оператора *switch*

```
switch ( выражение )
{
      case константное_вырмение_1: [список_операторов_1]
      case константное_выражение__2: [список_операторов_2]

      case константное_выражение_n:  [список_операторов_n]
      [default: операторы ]
}
```

Выполнение оператора начинается с вычисления выражения (оно должно быть целочисленным), а затем управление передается первому оператору из списка, помеченного константным выражением, значение которого совпало с вычисленным. После этого, если выход из переключателя явно не указан, последовательно выполняются все остальные ветви.

Выход из переключателя обычно выполняется с помощью операторов *break* или *return*. Оператор *break* выполняет выход из самого внутреннего из объемлющих его операторов *switch, for, while* и *do*. Оператор *return* выполняет выход из функции, в теле которой он записан.

Все константные выражения должны иметь разные значения, но быть одного и того же целочисленного типа. Несколько меток могут следовать подряд. Если совпадения не произошло, выполняются операторы, расположенные после слова *default* (а при его отсутствии управление передается следующему за *switch* оператору).

Листинг 1.15. Простейший калькулятор

Программа реализует простейший калькулятор на 4 действия.

```
#include <iostream.h>
int main()
{
      int a,  b,  res;
      char op;
      cout « “Введите 1й операнд ; ";  cin » a:
      cout « “Введите знак операции : "; cin » op;
      cout « “Введите 2й операнд : ";  cin » b;
      bool f = true;
      switch (op)
      {
            case '+': res = a + b; break;
            case '-': res = a - b: break;
            case '*': res = a * b; break;
            case '/' : res = a / b; break;
            default : cout «"\nНеизвестная операция"; f = false;
      }
      if (f)
            cout « “Результат” <<res;
      return 0;
}
```

Дополнительно об операторах ветвления рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 60 – 62.
2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 90 - 96.

### 1.4.3 Операторы цикла

Операторы цикла используются для организации многократно повторяющихся вычислений. Любой цикл состоит из тела цикла, то есть тех операторов, которые выполняются несколько раз, начальных установок, модификации *параметра цикла* и проверки *условия продолжения выполнения цикла* (рис. 1.5). Один проход цикла называется *итерацией*. Проверка условия выполняется на каждой итерации либо до тела цикла (тогда говорят о *цикле с предусло-*

*вием*), либо после тела цикла (*цикл с постусловием*). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

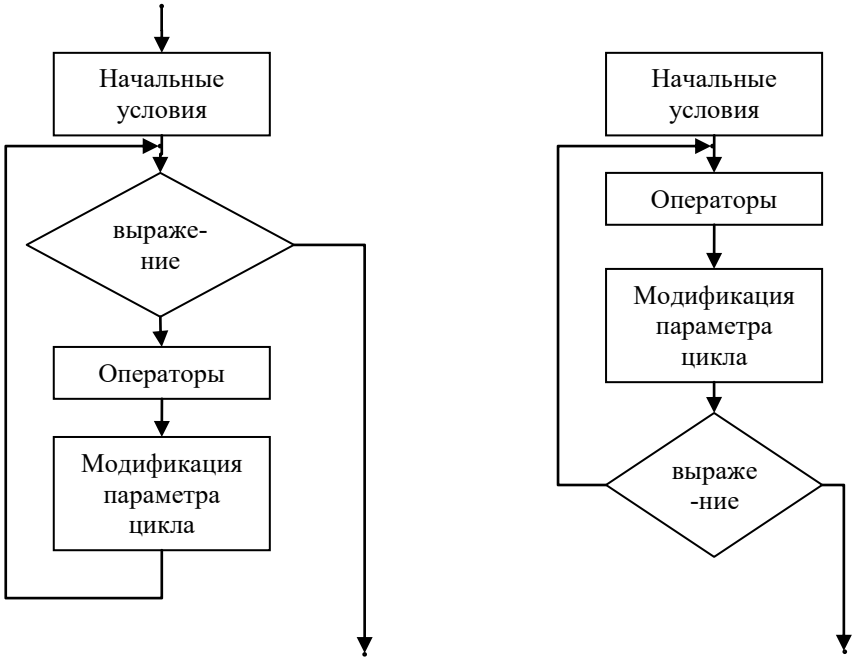


Рисунок 1.5. Структурные схемы операторов цикла:
a. цикл с предусловием
b. цикл с постусловием

Переменные, изменяющиеся в теле цикла и используемые при проверке условия продолжения, называются параметрами цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются *счетчиками цикла*.

Начальные установки могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в нем используются.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы *break, continue, return* и *goto* (данные операторы будут рассмотрены в разделе «Операторы передачи управления»). Передавать управление извне внутрь цикла не рекомендуется.

Для удобства, а не по необходимости, в C++ есть три разных оператора цикла — *while, do while* и *for*.

#### 1.4.3.1 Цикл с предусловием (while)

Цикл с предусловием реализует структурную схему, приведенную на рис. 1.5a, и имеет вид:

```
while(выражение) оператор
```

Выражение определяет условие повторения тела цикла, представленного простым или составным оператором. Выполнение оператора начинается с вычисления выражения. Если оно истинно (не равно *false*), выполняется оператор цикла. Если при первой проверке выражение равно *false*, цикл не выполнится ни разу. Тип выражения должен быть арифметическим или приводимым к нему. Выражение вычисляется перед каждой итерацией цикла.

Листинг 1.16. Табличные значения функции

Программа печатает таблицу значений функции $y = x^2 + 1$ во введенном диапазоне.

```
#include <stdio.h>
int main()
```

```
{
    float Xn, Xk, Dx;
    printf("Введите диапазон и шаг изменения аргумента: ");
    scanf("%f%f%f", &Xn, &Xk, &Dx);
    printf("|   X   |    y    \n");                          //Шапка цикла
    float X = Xn;                                           //установка параметра цикла
    while(X <= Xk)                                          //проверка условия продолжения
    {
        printf(" |   %5.2f   |   %5.2f  |\n", X, X*X+1);    //тело цикла
        X += Dx;
    }
    return 0;
}
```

Листинг 1.17. Делители положительного числа

Программа находит все делители целого положительного числа

```
#include <iostream.h>
int main()
{
    int num;
    cout « "\n Введите число :  ";
    cin » num;
    int half = num / 2;  // половина числа
    int div = 2;           // кандидат на делитель
    while (div <= half)
    {
        if (!(num % div))
            cout « div <<"\n";
        div++;
    }
    return 0;
}
```

Распространенный прием программирования — организация бесконечного цикла с заголовком *while (true)* либо *while (1)* и принудительным выходом из тела цикла по выполнению какого-либо условия.

В круглых скобках после ключевого слова *while* можно вводить описание переменной. Областью ее действия является цикл:

```
while (int x = 0){ ... /* область действия x */ }
```

#### 1.4.3.2 Цикл с постусловием (do while)

Цикл с постусловием реализует структурную схему, приведенную на рис. 1.5б, и имеет вид:

```
do оператор while выражение;
```

Сначала выполняется простой или составной оператор, составляющий тело цикла, а затем вычисляется выражение. Если оно истинно (не равно *false*), тело цикла выполняется еще раз. Цикл завершается, когда выражение станет равным *false* или в теле цикла будет выполнен какой-либо оператор передачи управления. Тип выражения должен быть арифметическим или приводимым к нему.

Листинг 1.18. Проверка ввода пароля

```
#include <iostream.h>
int main()
{
    char answer;
    do
    {
        cout « "\n Купи слоника! ";
        cin"» answer;
    }while (answer != 'y');
    return 0;
}
```

Листинг 1.19. Поиск квадратного корня

Программа вычисляет квадратный корень вещественного аргумента *X* с заданной точностью *Eps* по итерационной формуле:

$$y_n = ½*(y_{n-1}+x/y_{n-1})$$

где $y_{n-1}$ – предыдущее приближение к корню (в начале вычислений выбирается произвольно), $y_n$ — последующее приближение. Процесс вычислений прекращается, когда приближения станут отличаться друг от друга по абсолютной величине менее, чем на величину заданной точности. Для вычисления абсолютной величины используется стандартная функция *fabs()*, объявление которой находится в заголовочном файле <*math.h*>.

```
#include <stdio.h>
#include <math.h>
int main()
{
      double X, Eps;
      double Yp, Y = 1;
      printf("Введите аргумент и точность ");
      scanf("%lf%lf", &X, &Eps);
      do
      {
            Yp=Y;
            Y = (Yp+X/Yp)/2;
      }while(fabs(Y-Yp)>=Eps);
      printf("\nКорень из %lf равен %lf", X, Y);
      return 0;
}
```

### 1.4.3.3 Цикл с параметром

*Цикл с параметром* имеет следующий формат:

```
for (инициализация:  выражение;  модификации) оператор;
```

*Инициализация* используется для объявления и присвоения начальных значений величинам, используемым в цикле. В этой части можно записать несколько операторов, разделенных запятой (операцией «последовательное выполнение»), например, так:

```
for(int i=0; j = 2; …
int k, m;
for(k =1; m = 0; …
```

Областью действия переменных, объявленных в части инициализации цикла, является цикл. Инициализация выполняется один раз в начале исполнения цикла. *Выражение* определяет условие выполнения цикла: если его результат, приведенный к типу *bool*, равен *true*, цикл выполняется. Цикл с параметром реализован как цикл с предусловием.

*Модификации* выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую. Простой или составной *оператор* представляет собой тело цикла. Любая из частей оператора for может быть опущена (но точки с запятой надо оставить на своих местах!).

Листинг 1.20. Пример цикла с параметром

Оператор, вычисляющий сумму чисел от 1 до 100:

```
for (int i = l,  s = 0; i<=100; i++) s += 1;
```

Листинг 1.21. Табличные значения функции

Программа печатает таблицу значений функции $y=x^2+1$ во введенном диапазоне

```
#include <stdio.h>
int main()
{
      float Xn, Xk, Dx, X;
      printf("Введите диапазон и шаг изменения аргумента ");
      scanf("%f%f%f", &Xn, &Xk, &Dx);
      printf("|  X   |   Y   |\n");
      for(X=Xn; X<=Xk; X += Dx)
```

```
            printf("|   %5.2f     |  %5.2f  |\n", X, X*X+1);
      retunt 0;
}
```

Листинг 1.22. Делители целого положительного числа

Программа находит все делители целого положительного числа:

```
#include <iostream.h>
int main()
{
      int num, half, div;
      cout << "\nВведите число : ";
      cin >> num;
      for(half = num/2, div=2; div<=half; div++)
            if(!(num% div))
                  cout<<div <<"\n";
      return 0;
}
```

Два последних примера выполняют те же действия, что и примеры для цикла с предусловием, но записаны более компактно и наглядно: все действия, связанные с управлением циклом, локализованы в его заголовке.

Любой цикл *while* может быть приведен к эквивалентному ему циклу *for* и наоборот по следующей схеме:

```
for (b1; b2; b2) оператор
```

```
b1;
while (b2){ оператор;  b3;}
```

Часто встречающиеся *ошибки при программировании циклов* — использование в теле цикла неинициализированных переменных и неверная запись условия выхода из цикла.

Чтобы избежать ошибок, рекомендуется:

- проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения (а также возможно ли выполнение других операторов);
- проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
- предусмотреть аварийный выход из цикла по достижению некоторого количества итераций (см. пример в следующем разделе);
- и, конечно, не забывать о том, что если в теле цикла требуется выполнить более одного оператора, нужно заключать их в фигурные скобки.

Операторы цикла взаимозаменяемы, но можно привести некоторые *рекомендации* по выбору наилучшего в каждом конкретном случае.

Оператор *do while* обычно используют, когда цикл требуется обязательно выполнить хотя бы раз (например, если в цикле производится ввод данных).

Оператор *for* предпочтительнее в большинстве остальных случаев (однозначно — для организации циклов со счетчиками).

Оператором *while* удобнее пользоваться в случаях, когда число итераций заранее не известно, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла.

Об операторах цикла рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 60 – 62.
2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 96 – 101.

### 1.4.4 Операторы передачи управления

В *C++* есть четыре оператора, изменяющих естественный порядок выполнения вычислений:

- оператор безусловного перехода *goto*;
- оператор выхода из цикла *break*;

- оператор перехода к следующей итерации цикла *continue*;
- оператор возврата из функции *return*.

#### 1.4.4.1 Оператор *goto*

Оператор безусловного перехода *goto* имеет формат:

```
goto метка;
```

В теле той же функции должна присутствовать ровно одна конструкция вида:

```
метка: оператор;
```

Оператор *goto* передает управление на помеченный оператор. *Метка* — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан.

Использование оператора безусловного перехода оправдано в двух случаях:

- принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;
- переход из нескольких мест функции в одно (например, если перед выходом из функции всегда необходимо выполнять какие-либо действия).

В остальных случаях для записи любого алгоритма существуют более подходящие средства, а использование *goto* приводит только к усложнению структуры программы и затруднению отладки. Применение *goto* нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и один выход.

В любом случае не следует передавать управление внутрь операторов *if, switch* и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию переменных, на операторы, расположенные после нее, поскольку в этом случае инициализация не будет выполнена.

Листинг 1.23. Пример применения оператора *goto*

```
int k; ...
goto metka; ...
{
      int a = 3, b = 4;
      к = a + b;
metka:
      int m = к + 1; ...
}
```

После выполнения этого фрагмента программы значение переменной *m* не определено.

#### 1.4.4.2 Оператор *break*

Оператор *break* используется внутри операторов цикла или *switch* для обеспечения перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится *break*.

Листинг 1.24. Вычисление гиперболического синуса

Программа вычисляет значение гиперболического синуса вещественного аргумента x с заданной точностью eps с помощью разложения в бесконечный ряд.

$sh\,x = 1 + x^3/3! + x^5/5! + x77! + \ldots$

Вычисление закапчивается, когда абсолютная величина очередного члена ряда, прибавляемого к сумме, станет меньше заданной точности.

```
#include <iostream.h>
#include <math.h>
int main()
{
      const int MaxIter = 500:      // ограничитель количества итераций
      double x, eps;
      cout « "\nВведите аргумент и точность: ";
      cin » x » eps;
      bool flag = true;    // признак успешного вычисления
      double y = x, ch = x;          // сумма и первый член ряда
      for (int n = 0: fabs(ch) > eps: n++)
      {
```

```
            ch *= x * x /(2 * n + 2)/(2 * n + 3); // очередной член ряда
            y += ch;
            if (n > MaxIter)
            {
                cout « "\nРяд расходится!";
                flag = false;
                break;
            }
        }
        if (flag) cout « "\nЗначение функции: " « y;
        return 0;
}
```

#### 1.4.4.3 Оператор *continue*

Оператор перехода к следующей итерации цикла *continue* пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации.

#### 1.4.4.4 Оператор *return*

Оператор возврата из функции *return* завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

```
return [ выражение ];
```

Выражение должно иметь скалярный тип. Если тип возвращаемого функцией значения описан как *void*, выражение должно отсутствовать.

## 1.5 Указатели и массивы

### 1.5.1 Указатели

Когда компилятор обрабатывает оператор определения переменной, например,

```
int i=10;
```

он выделяет память в соответствии с типом (*int*) и инициализирует ее указанным значением (*10*). Все обращения в программе к переменной по ее имени (*1*) заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Итак, *указатели предназначены для хранения адресов областей памяти.* В C++ различают три вида указателей — *указатели на объект, на функцию и на void*, отличающиеся свойствами и набором допустимых операций. Указатель не является самостоятельным типом, он всегда связан с каким-либо другим конкретным типом.

*Указатель на функцию* содержит адрес в сегменте кода, по которому располагается исполняемый код функции, то есть адрес, по которому передается управление при вызове функции. Указатели на функции используются для косвенного вызова функции (не через ее имя, а через обращение к переменной, хранящей ее адрес), а также для передачи имени функции в другую функцию в качестве параметра. Указатель функции имеет тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа»:

```
тип (*имя) ( список_типов_аргументов );
```

Например, объявление:

```
int (*fun) (double, double);
```

задает указатель с именем fun на функцию, возвращающую значение типа *int* и имеющую два аргумента типа *double*.

*Указатель на объект* содержит адрес области памяти, в которой хранятся данные определенного типа (основного или составного). Простейшее объявление указателя на объект (в дальнейшем называемого просто указателем) имеет вид:

```
тип *имя;
```

где тип может быть любым, кроме ссылки и битового поля, причем тип может быть к этому моменту только объявлен, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

*Звездочка относится непосредственно к имени,* поэтому для того, чтобы объявить несколько указателей, требуется ставить ее перед именем каждого из них. Например, в операторе

```
int *a, b, *c;
```

описываются два указателя на целое с именами *a* и *c*, а также целая переменная *b*.

Размер указателя зависит от модели памяти. Можно определить указатель на указатель и т. д.

*Указатель на void* применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю на *void* можно присвоить значение указателя любого типа, а также сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом.

Указатель может быть константой или переменной, а также указывать на константу или переменную.

Листинг 1.25. Примеры инициализации переменных и указателей

```
int i;                          // целая переменная
const int ci = 1;               // целая константа
int* pi;                        // указатель на целую переменную
const int* pci;                 // указатель на целую константу
int * const cp = &i;            // указатель-константа на целую переменную
const int* const cpc =&ci:      // указатель-константа на целую константу
```

Как видно из примеров, модификатор *const*, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а *const* слева от звездочки задает постоянство значения, на которое он указывает. Для инициализации указателей использована операция получения адреса &.

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни.

#### 1.5.1.1 Инициализация указателей

Указатели чаще всего используют при работе с динамической памятью, называемой некоторыми эстетами кучей (перевод с английского языка слова *heap*). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти, называемым динамическими переменными, производится только через указатели. Время жизни динамических переменных — от точки создания до конца программы или до явного освобождения памяти. В C++ используется два способа работы с динамической памятью. Первый использует семейство функций *malloc* и достался в наследство от *C*, второй использует операции *new* и *delete*.

При определении указателя надо стремиться выполнить его инициализацию, то есть присвоение начального значения. Непреднамеренное использование неинициализированных указателей — распространенный источник ошибок в программах. Инициализатор записывается после имени указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. *Присваивание указателю адреса существующего объекта*

   – с помощью операции получения адреса:

```
int a = 5:                      // целая переменная
int* p = &a:                    // в указатель записывается адрес a
int* p(&a);                     // то же самое другим способом
```

   – с помощью значения другого инициализированного указателя:

```
int*  r =  p;
```

– с помощью имени массива или функции, которые трактуются как адрес

```
int b[10];                          // массив
int* t = b:                         // присваивание адреса начала массива
void f(int a ){ /* ... */ }         // определение функции
void (*pf)(int);                    // указатель на функцию
pf = f;                             // присваивание адреса функции
```

*2. Присваивание указателю адреса области памяти в явном виде:*

```
char* vp = (char *)OxB8000000;
```

Здесь *OxB8000QOO* — шестнадцатеричная константа, *(char *)* — операция приведения типа: константа преобразуется к типу «*указатель на char*».

*3. Присваивание пустого значения:*

```
int* suxx = NULL;
int* rulez = 0:
```

В первой строке используется константа *NULL*, определенная в некоторых заголовочных файлах *C* как указатель, равный нулю. Рекомендуется использовать просто 0, так как это значение типа *int* будет правильно преобразовано стандартными способами в соответствии с контекстом. Поскольку гарантируется, что объектов с нулевым адресом нет, пустой указатель можно использовать для проверки, ссылается указатель на конкретный объект или нет.

*4.* Выделение участка динамической памяти и присваивание ее адреса указателю*;*

– с помощью операции new:

```
int* n = new int;                   //      1
int* m = new int (10);              //      2
int* q = new int [10];              //      3
```

– с помощью функции ma11oc:

```
int* u = (int *)malloc(sizeof(int));        // 4
```

В операторе 1 операция *new* выполняет выделение достаточного для размещения величины типа *int* участка динамической памяти и записывает адрес начала этого участка в переменную *n*. Память под саму переменную *n* (размера, достаточного для размещения указателя) выделяется на этапе компиляции.

В операторе 2, кроме описанных выше действий, производится инициализация выделенной динамической памяти значением 10.

В операторе 3 операция *new* выполняет выделение памяти под 10 величин типа *int* (массива из 10 элементов) и записывает адрес начала этого участка в переменную *q*, которая может трактоваться как имя массива. Через имя можно обращаться к любому элементу массива.

Если память выделить не удалось, по стандарту должно порождаться исключение *bad_alloc*. Старые версии компиляторов могут возвращать 0.

В операторе 4 делается то же самое, что и в операторе 1, но с помощью функции выделения памяти *malloc*, унаследованной из библиотеки C. В функцию передается один параметр — количество выделяемой памяти в байтах. Конструкция *(int*)* используется для приведения типа указателя, возвращаемого функцией, к требуемому типу. Если память выделить не удалось, функция возвращает 0.

Операцию *new* использовать предпочтительнее, чем функцию *malloc*, особенно при работе с объектами.

Освобождение памяти, выделенной с помощью операции *new*, должно выполняться с помощью *delete*, а памяти, выделенной функцией *malloc* — посредством функции *free*. При этом переменная-указатель сохраняется и может инициализироваться повторно. Приведенные выше динамические переменные уничтожаются следующим образом:

```
delete n;
delete m;
delete [] q;
free (u);
```

Если память выделялась с помощью *new[ ]*, для освобождения памяти необходимо применять *delete[ ]*. Размерность массива при этом не указывается. Если квадратных скобок нет, то никакого сообщения об ошибке не выдается, но помечен как свободный будет только первый

элемент массива, а остальные окажутся недоступны для дальнейших операций. Такие ячейки памяти называются мусором.

Если переменная-указатель выходит из области своего действия, отведенная под нее память освобождается. Следовательно, динамическая переменная, на которую ссылался указатель, становится недоступной. При этом память из-под самой динамической переменной не освобождается. Другой случай появления «мусора» — когда инициализированному указателю присваивается значение другого указателя. При этом старое значение бесследно теряется.

С помощью комбинаций звездочек, круглых и квадратных скобок можно описывать составные типы и указатели на составные типы, например, в операторе

```
int *(*p[10])();
```

объявляется массив из 10 указателей на функции без параметров, возвращающих указатели на *int*.

По умолчанию квадратные и круглые скобки имеют одинаковый приоритет, больший, чем звездочка, и рассматриваются слева направо. Для изменения порядка рассмотрения используются круглые скобки.

При интерпретации сложных описаний необходимо придерживаться правила «изнутри наружу»:

- если справа от имени имеются квадратные скобки, это массив, если скобки круглые — это функция;
- если слева есть звездочка, это указатель на проинтерпретированную ранее конструкцию;
- если справа встречается закрывающая круглая скобка, необходимо применить приведенные выше правила внутри скобок, а затем переходить наружу;
- в последнюю очередь интерпретируется спецификатор типа.

Для приведенного выше описания порядок интерпретации указан цифрами:

```
int *(*p[10])();
5  4 2 1  3                                   // порядок интерпретации описания
```

### 1.5.1.2 Операции с указателями

С указателями можно выполнять следующие операции: *разадресация*, или *косвенное обращение к объекту* (*), *присваивание, сложение с константой, вычитание, инкремент (++), декремент (—), сравнение, приведение типов*. При работе с указателями часто используется операция получения адреса (&).

**Операция разадресации**, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа):

```
char a;                                    // переменная типа char
char * p = new char;                       /* выделение памяти под указатель и под
                                             динамическую переменную типа char */
*p = 'Ю'; a = *p:                          // присваивание значения обеим переменные
```

Как видно из примера, конструкцию **имя_указателя* можно использовать в левой части оператора присваивания, так как она является *L-значением*, то есть определяет адрес области памяти. Для простоты эту конструкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа (если указатель инициализирован). На одну и ту же область памяти может ссылаться несколько указателей различного типа. Примененная к ним операция разадресации даст разные результаты.

Листинг 1.26. Пример операций разадресации

```
#include <stdio.h>
int main()
{
 unsigned long int A = 0Xcc77ffaa;
```

```
unsigned short int* pint = (unsigned short int*) &A;          //типы данных необходимо изучить самостоятельно!!!
unsigned char* pchar = (unsigned char *) &A;
printf("  | %x | %x | %x |", A, *pint, *pchar);
return 0;
}
```

на IBM PC-совместимом компьютере выведет на экран строку:

Листинг 1.27. Результаты программы листинга 1.26.

```
  | cc77ffaa  |  ffaa | aa |
```

Значения указателей pint и pchar одинаковы, но разадресация pchar дает в результате один младший байт по этому адресу, а pint — два младших байта.

В приведенном выше примере при инициализации указателей были использованы операции приведения типов. Синтаксис *операции явного приведения типа* прост: перед именем переменной в скобках указывается тип, к которому ее требуется преобразовать. При этом не гарантируется сохранение информации, поэтому в общем случае явных преобразований типа следует избегать.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме *void**. Указатель может неявно преобразовываться в значение типа *bool* (например, в выражении условного оператора), при этом ненулевой указатель преобразуется в *true*, а нулевой в *false*.

Присваивание без явного приведения типов допускается в двух случаях:

- указателям типа void*;
- если тип указателей справа и слева от операции присваивания один и тот же.

Таким образом, неявное преобразование выполняется только к типу *void**. Значение 0 неявно преобразуется к указателю на любой тип. Присваивание указателей на объекты указателям на функции (и наоборот) недопустимо. Запрещено и присваивать значения указателям-константам, впрочем, как и константам любого типа (присваивать значения указателям на константу и переменным, на которые ссылается указатель-константа, допускается).

**Арифметические операции** с указателями (*сложение с константой, вычитание, инкремент* и *декремент*) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например, с массивами.

*Инкремент* перемещает указатель к следующему элементу массива, декремент — к предыдущему. Фактически значение указателя изменяется на величину *sizeof* (тип). Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

Листинг 1.28. Применение операции инкремента к указателям

```
short * p = new short [5]:
p++;                                          // значение p увеличивается на 2
long * q = new long [5];
q++;                                          // значение q увеличивается на 4
```

*Разность двух указателей* — это разность их значений, деленная на размер типа в байтах (в применении к массивам разность указателей, например, на третий и шестой элементы равна 3). Суммирование двух указателей не допускается.

При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

```
*p++ = 10;
```

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево, но, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10;
p++;
```

Выражение *(*p)++,* напротив, инкрементирует значение, на которое ссылается указатель.

*Унарная операция получения адреса &* применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной. Примеры операции приводились выше.

### 1.5.2 Ссылки

Ссылка представляет собой синоним имени, указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

```
тип &имя;
```

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, значающий, что следующее за ним имя является именем переменной ссылочного типа.

Листинг 1.29. Применение ссылок

```
int kol:
int& pal = kol:                          // ссылка pal - альтернативное имя для kol
const char& CR = '\n';                   // ссылка на константу
```

Запомните следующие правила.

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции, описана как *extern* или ссылается на поле данных класса.
- После инициализации ссылке не может быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- Не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

### 1.5.3 Массивы

#### 1.5.3.1 Одномерные массивы

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнять однообразные действия, им дают одно имя, а различают по порядковому номеру. Это позволяет компактно записывать множество операций с помощью циклов. Конечная именованная последовательность однотипных величин называется *массивом***.** Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива (размерность):

```
float a[10];     // описание массива из 10 вещественных чисел
```

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, *const* и инициализатор), что и для простых переменных. Инициализирующие значения для массивов записываются в фигурных скобках. Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

```
int b[5] = {3, 2, 1}: // b[0]=3, b[l]=2. b[2]=l, b[3]=0, b[4]=0
```

Размерность массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, поэтому размерность может быть задана только целой положительной *константой* или *константным выражением*. Если при описании массива не указана размерность, должен присутствовать инициализатор, в этом случае компилятор выделит память по количеству инициализирующих значений. В дальнейшем мы увидим, что размерность может быть опущена также в списке формальных параметров.

Для доступа к элементу массива после его имени указывается номер элемента (индекс) в квадратных скобках.

Листинг 1.30. Расчет суммы элементов массива

```
#include <iostream.h>
int main()
{
        const int n = 10;
        int I, sum;
        int marks[n] = {3, 4, 5, 4, 4};
        for (i = 0, sum = 0; i<n; i++)
            sum += marks[i];
        cout « "Сумма элементов: " « sum;
        return 0;
}
```

Размерность массивов предпочтительнее задавать с помощью *именованных констант,* как это сделано в примере, поскольку при таком подходе для ее изменения достаточно скорректировать значение константы всего лишь в одном месте программы. Обратите внимание, что последний элемент массива имеет номер, на единицу меньший заданной при его описании размерности.

Листинг 1.31. Сортировка целочисленного массива методом выбора.

Алгоритм состоит в том, что выбирается наименьший элемент массива и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом, и так далее *n-1* раз (при последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива).

```
#include <iostream.h>
int main()
{
    const int n = 20;                       //количество элементов массива
    int b[n];                               //описание массива
    int i;
    for (i = 0;  i<n;  i++)
        cin » b[i]                          //ввод массива
    for (i = 0:  i<n-l;  i++)`              //n-1 раз ищем наименьший элемент
    {
                                            // принимаем за наименьший первый из рассматриваемых
                                                    элементов:
        int imin = 1;
                                            // поиск номера минимального элемента из неупорядоченных:
        for (int j = i + 1; j<n; j++)       // если нашли меньший элемент, запоминаем его номер:
            if (b[j] < b[imin])
                imin = j;
        int a = b[i];                       // обмен элементов
        b[i] = b[imin];                     // с номерами
        b[imin] = a;                        //i в imin
    }
                                            // вывод упорядоченного массива:
    for (i = 0; i<n: i++)
        cout « b[i] « '';
    return 0;
}
```

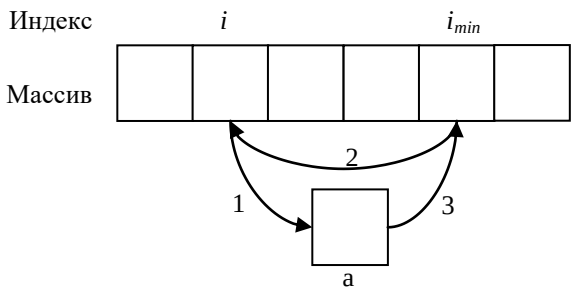


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.6. Обмен значений двух переменных

Процесс обмена элементов массива с номерами i и imin через буферную переменную a на 1-м проходе цикла проиллюстрирован на рис.Рисунок Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.6**. Цифры около стрелок обозначают порядок действий.

Идентификатор массива является константным указателем на его нулевой элемент. Например, для массива из предыдущего листинга имя *b* — это то же самое, что *&b[0],* а к *i*-му элементу массива можно обратиться, используя выражение **(b+i).* Можно описать указатель, присвоить ему адрес начала массива и работать с массивом через указатель.

Листинг 1.32. Копирование массива

```
int a[100], b[100];
int *pa = a;                                    //или int *p = &a[0]
int *pb = b;
for(int i = 0;  i<100;  i++)
      *pb++ = *pa++;                            // или pb[i] = pa[1];
```

### 1.5.3.2 Динамические массивы

*Динамические массивы* создают с помощью операции *new,* при этом необходимо указать тип и размерность.

Листинг 1.33. Пример инициализации динамического массива

```
int n = 100;
float *p = new float [n];
```

В этой строке создается переменная-указатель на *float*, в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель *p*. Динамические массивы нельзя при создании инициализировать, и они не обнуляются.

Преимущество динамических массивов состоит в том, что размерность может быть переменной, то есть объем памяти, выделяемой под массив, определяется па этане выполнения программы. Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведенного выше массива можно обратиться как *p[5]* или **(p+5).*

Альтернативный способ создания динамического массива — использование функции *malloc* библиотеки С:

Листинг 1.34. Пример инициализации динамического массива

```
int n = 100;
float *q = (float *) malloc(n * sizeof(float));
```

Операция преобразования типа, записанная перед обращением к функции *malloc*, требуется потому, что функция возвращает значение указателя типа *void*,* а инициализируется указатель на *float*.

Память, зарезервированная под динамический массив с помощью *new[]*, должна освобождаться оператором *delete []*, а память, выделенная функцией *malloc* — посредством функции *free*.

Листинг 1.35. Освобождение памяти, занятой под динамический массив

```
delete [] p;
free (q);
```

При несоответствии способов выделения и освобождения памяти результат не определен. Размерность массива в операции *delete* не указывается, но квадратные скобки обязательны.

#### 1.5.3.3 Многомерные массивы

*Многомерные массивы* задаются указанием каждого измерения в квадратных скобках, например, оператор

```
int matr [6][8];
```

задает описание двумерного массива из 6 строк и 8 столбцов. В памяти такой массив располагается в последовательных ячейках построчно. Многомерные массивы размещаются так, что при переходе к следующему элементу быстрее всего изменяется последний индекс. Для доступа к элементу многомерного массива указываются все его индексы, например, *matr[i][j]*, или более экзотическим способом: **(matr[i]+j)* или **(*(matr+i)+j)*. Это возможно, поскольку *matr[i]* является адресом начала *i*-й строки массива.

При инициализации многомерного массива он представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задается общий список элементов в том порядке, в котором элементы располагаются в памяти:

```
int mass2 [][2] = {  {1,  1},  {0, 2}.  {1, 0}  };
int mass2 [3][2] - {1,  1, 0, 2, 1, 0};
```

Листинг 1.36. Поиск строки с максимальным количеством элементов равных нулю

```
#include <stdio.h>
int main()
{
      const int nstr = 4, nstb = 5;
      int b[nstr][nstb];
      for(int i=0;i<nstr;i++)
            for(int j=0;j<nstb;j++)
                  scanf("%d",&b[i][j]);
      int istr=-1, MaxKol = 0;
      for(int i=0;i<nstr;i++)
      {
            int Kol=0;
            for(int j=0;j<nstb;j++)
                  if(b[i][j]==0)
                  Kol++;
            If(Kol>MaxKol)
            {
                  istr=i;
                  MaxKol=Kol;
            }
      }
      printf(" Исходный массив: \n");
      for(int i=0;i<nstr;i++)
      {
            for(int j=0;j<nstb;j++)
                  printf(" %d  ",b[i][j]);
            printf("\n');
      }
      if(istr== -1)
            printf("Нулевых элементов нет");
      else
            printf(" Номер строки:  %d", istr);
      return 0;
}
```

Номер искомой строки хранится в переменной *istr*, количество нулевых элементов в текущей *(i-й)* строке — в переменной *Kol*, максимальное количество нулевых элементов — в переменной *MaxKol*. Массив просматривается по строкам, в каждой из них подсчитывается количество нулевых элементов (обратите внимание, что переменная *Kol* обнуляется перед просмотром каждой строки). Наибольшее количество и номер соответствующей строки запоминаются.

Для создания динамического многомерного массива необходимо указать в операции new все его размерности (самая левая размерность может быть переменной).

Листинг 1.37. Создание многомерного динамического массива

```
int nstr = 5;
int ** m = (int **) new int [nstr][10];
```

Более универсальный и безопасный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы.

Листинг 1.38. Создание многомерного динамического массива

```
int nstr, nstb;
cout « " Введите количество строк и столбцов :";
cin » nstr » nstb;
int **a = new int *[nstr].;                                    // 1
for (int i = 0;  i<nstr;  i++)                                 // 2
 a[i] = new int [nstb];                                        // 3
```

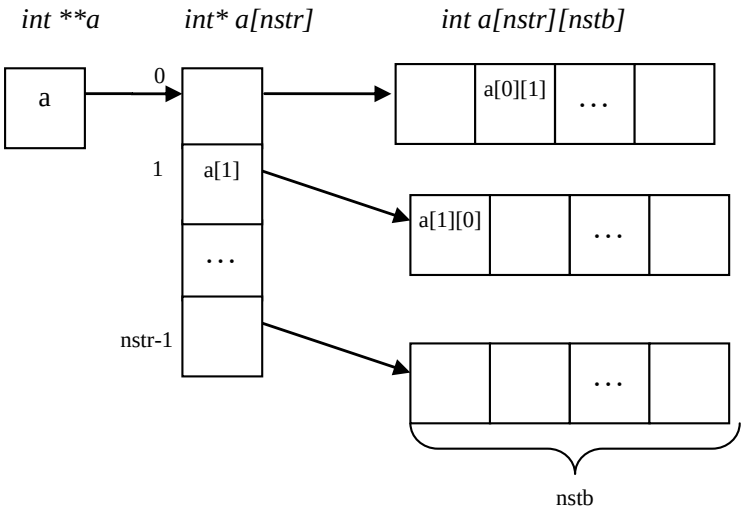


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.7. Выделение памяти под двумерный массив

В операторе 1 объявляется переменная типа *«указатель на указатель на int»* и выделяется память под массив указателей на строки массива (количество строк — *nstr*). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из *nstb* элементов типа *int* (рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.7**).

Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции *delete []*. Указатель на константу удалить нельзя.

### 1.5.4 Строки

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным 0, что записывается в виде управляющей последовательности '\0'. По положению нуль-символа определяется фактическая длина строки. Строку можно инициализировать строковым литералом.

```
char str[10] = "Vasia";                    // выделено 10 элементов с номерами от 0 до 9
                                           // первые элементы - 'V. 'a'. V, 'i', 'a'. '\0'
```

В этом примере под строку выделяется 10 байт, 5 из которых занято под символы строки, а шестой — под нуль-символ. Если строка при определении инициализируется, ее размерность можно опускать (компилятор сам выделит соответствующее количество байт):

```
char str[] = "Vasia";                                  // выделено и заполнено 6 байт Оператор
```

Оператор

```
char *str = "Vasia"
```

создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно (к примеру, оператор *str[l]='o'* не допускается). Знак равенства перед строковым литералом означает инициализацию, а не присваивание. Операция присваивания одной строки другой не определена (поскольку строка является массивом) и может выполняться с помощью цикла или функций стандартной библиотеки. Библиотека предоставляет возможности копирования, сравнения, объединения строк, поиска подстроки, определения длины строки и т. д. (возможности библиотеки описаны в разделе «Функции работы со строками и символами», с. 91, и в приложении 6), а также содержит специальные функции ввода строк и отдельных символов с клавиатуры и из файла.

Листинг 1.39. Программа запроса пароля

Программа запрашивает пароль не более трех раз.

```
#include <stdio.h>
#include <string.h>
int main()
{
      char s[80], passw[ ] = "kuku";                          // passw - эталонный пароль.
                                                              // Можно описать как *passw =
      "kuku":
      int i, k = 0;
      for (i = 0; !k && i<3; i++)
      {
            printf("\nВведите пароль:\n"):

            gets(s);                                          // функция ввода строки
            if (strstr(s, passw))
                  k = 1;                                      // функция сравнения строк
      }
      if (k)
            printf("Пароль принят");
      else
            printf("\nПароль не принят");
      return 0;
}
```

При работе со строками часто используются указатели.

Распространенные ошибки при работе со строками — отсутствие нуль-символа и выход указателя при просмотре строки за се пределы.

Рассмотрим процесс копирования строки *src* в строку *dest*.

Листинг 1.40. Алгоритм копирования строк как массивов

```
char src[10], dest[10];
for (int i = 0; i<=strlen(src); i++)
      dest[i] = src[i];
```

Длина строки определяется с помощью функции *strlen,* которая вычисляет длину, выполняя поиск нуль-символа. Таким образом, строка фактически просматривается дважды. Более эффективным будет использовать проверку на нуль-символ непосредственно в программе. Увеличение индекса можно заменить инкрементом указателей (для этого память под строку *src* должна выделяться динамически, а также требуется определить дополнительный указатель и инициализировать его адресом начала строки *dest*).

Листинг 1.41. Алгоритм копирования строк при помощи указателей

```
#include <iostream.h>
int main()
{
      char *src = new char [10];
      char *dest = new char [1], *d = dest;
      cin » src;
      while ( *src != 0)
            *d++ = *src++;
```

```
    *d = 0;                                          // завершающий нуль
    cout « dest:
    return 0;
}
```

В цикле производится посимвольное присваивание элементов строк с одновременной инкрементацией указателей. Результат операции присваивания — передаваемое значение, которое, собственно, и проверяется в условии цикла, поэтому можно поставить присваивание на место условия, а проверку на неравенство нулю опустить (при этом завершающий нуль копируется в цикле, и отдельного оператора для его присваивания не требуется). В результате цикл копирования строки принимает вид:

```
while ( *d++ = *src++);
```

Оба способа работы со строками (через массивы или указатели) приемлемы и имеют свои плюсы и минусы, но в общем случае лучше не изобретать велосипед, а пользоваться функциями библиотеки или определенным в стандартной библиотеке *C++* классом *string*, который обеспечивает индексацию, присваивание, сравнение, добавление, объединение строк и поиск подстрок, а также преобразование из *C*-строк, то есть массивов типа *char*, в *string*, и наоборот.

Дополнительно об указателях, массивах и строках рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 127 – 141.
2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 108-157.

## 1.6 Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей. Язык *C++* позволяет программисту определять свои типы данных и правила работы с ними. Исторически для таких типов сложилось наименование, вынесенное в название главы, хотя правильнее было бы назвать их типами, определяемыми программистом.

### 1.6.1 Переименование типов (typedef)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова typedef:

```
typedef тип новое_имя [ размерность ];
```

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать.

Листинг 1.42. Примеры переименования типов

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct
{
    char fio[30];
    int date, code;
    double salary;
} Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов.

Листинг 1.43. Примеры использования введенных типов данных

```
UINT i, j;                                   //две переменных типа unsigned int
```

```
Msg str[10];                                    // массив из 10 строк по 100 символов
Worker staff[100];                              // массив из 100 структур
```

Кроме задания типам с длинными описаниями более коротких псевдонимов, *typedef* используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов *typedef,* при переносе программы потребуется внести изменения только в эти операторы.

### 1.6.2 Перечисления(enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

```
enum [ имя_типа ] { список_констант };
```

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей

Листинг 1.44. Пример использования перечислений

```
enum Err {ERR_READ,  ERR_WRITE,  ERR_CONVERT};
Err error;
switch (error)
{
      case ERR_READ:          /* операторы */ break;
      case ERR_WRITE:         /* операторы */ break;
      case ERR_CONVERT:       /* операторы */ break;
}
```

Константам *ERR_READ, ERR_WRITE, ERR__CONVERT* присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

Листинг 1.45. Пример изменения значений констант

```
enum {two = 2, three,  four, ten - 10. eleven, fifty = ten + 40};
```

Константам *three* и *four* присваиваются значения 3 и 4, константе *eleven* - 11.

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой *#define* (см. раздел «Директива *#define*») состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов.

При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции (см. «Перегрузка операций»).

### 1.6.3 Структуры (struct)

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов. В языке C++ структура является видом класса и обладает всеми его свойствами, но во многих случаях достаточно использовать структуры так, как они определены в языке C.

Листинг 1.46. Описание структуры как типа данных

```
struct [ имя_типа ]
{
      тип_1 элемент_1;
      тип 2 элемент 2;
```

```
        ……….
        тип_n эленент_n;
} [ список_описателей ];
```

Элементы структуры называются полями структуры и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов этого списка

Листинг 1.47. Примеры определения массива структур и указателя на структуру

```
struct
{
        char fio[30];
        int date, code;
        double salary;
}staff[100],  *ps;
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами.

Листинг 1.48. Пример описания типа данных «Структура Worker» и введение массива и указателя на структуру

```
struct Worker                                   // описание нового типа Worker
{
        char fio[30];
        int date, code;
        double salary;
};                                              // описание заканчивается точкой с запятой /
Worker staff[100], *ps;
                                                // определение массива типа Worker и
                                                        указателя на тип Worker
```

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее) в тех случаях, когда компилятору не требуется знать размер структуры.

Листинг 1.49. Пример использования имени структуры при описании другой структуры

```
struct List;                                    // объявление структуры List
struct Link
{
        List *p;                                // указатель на структуру List
        Link *prev. *succ;                       // указатели на структуру Link
};
struct List { /* определение структуры List */};
```

Это позволяет создавать связные списки структур.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания.

Листинг 1.50. Пример инициализации структуры при ее введении

```
struct
{
        char fio[30];
        int date, code;
        double salary;
}worker = {"Страусенке/.  31.  215.  3400.55};
```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов).

Листинг 1.51. Введение многомерного массива структур

```
struct complex
{
        float real, im;
} compl  [2][3] = {
                        {{1, 1}.  {1,  1},  {1. 1}},      // строка 1. то есть массив compl[0]
                        {{2, 2},  {2. 2},  (2, 2}}        // строка 2, то есть массив compl[1]
```

```
};
```

Для переменных одного и того же структурного типа определена операция присваивания, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем (см. «Перегрузка операций»). Размер структуры не обязательно равен сумме размеров ее элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора . (точка) при обращении к полю через имя структуры и -> при обращении через указатель.

Листинг 1.52. Пример доступа к полям структуры

```
Worker worker, staff[100], *ps;
worker.fio = "Страусенко";
staff[8].code = 215;
ps->salary = 0.12;
```

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

Листинг 1.53. Доступ к полям структуры

```
struct A {int a; double x;};
struct B (A a;  double x:}          x[2];
x[0].a.a = 1;
x[1].x = 0.1;
```

Как видно из примера, поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово *struct*, но не советую это делать — запутать компилятор труднее, чем себя.

### 1.6.4 Битовые поля

Битовые поля — это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти — 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа).

Листинг 1.54. Пример описания битового поля

```
struct Options
{
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4;
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом — по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды, и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

### 1.6.5 Объединения (union)

***Объединение (union)*** представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только вместо ключевого слова *struct* используется слово *union*. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на

программисте. Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется.

Листинг 1.55. Пример применения объединений

```
#include <iostream.h>
int main()
{
	enum paytype {CARD, CHECK};
	paytype ptype;
	union payment
	{
		char card[25];
		long check;
	} info;
	//Присваивание значений info и ptype;
	switch(ptype)
	{
			case CARD: cout<< “Оплата по карте: ” << info.card; break;
			case CHECK: cout<<”Оплата чеком: : << info.check;break;
	}
	return 0;
}
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно.

Листинг 1.56. Пример обращения к полям объединения

```
#include <iostream.h>
int main()
{
enum paytype{CARD, CHECK};
	struct
	{
		paytype ptype;
		union
		{
			char card[25];
			long check;
		};
	}info;
	//Присваивание значения info
	switch(info.ptype)
	{
			case CARD: cout<< “Оплата по карте: ” << info.card; break;
			case CHECK: cout<<”Оплата чеком: : << info.check;break;
	}
	return  0;
}
```

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов). В качестве примера рассмотрим работу со структурой, содержащей битовые поля

Листинг 1.57. Структура, содержащая битовые поля

```
struct Options
{
	bool centerX:1;
	bool centerY: 1;
	unsigned int chadow:2;
	unsigned int palette:2;
}
union
{
	unsigned char ch;
	Options bit;
}option = {0xC4};
cout « option.bit.palette;
option.ch & = OxFO; // наложение маски
```

По сравнению со структурами на объединения налагаются некоторые ограничения. Смысл некоторых из них станет понятен позже:

- объединение может инициализироваться только значением его первого элемента;
- объединение не может содержать битовые поля;
- объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
- объединение не может входить в иерархию классов.

Дополнительно о типах данных, определяемых пользователем, рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 115 - 120.

2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 237 - 262.

# 2. Модульное программирование

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части. В *C++* задача может быть разделена на более простые и обозримые с помощью *функций,* после чего программу можно рассматривать в более укрупненном виде — на уровне взаимодействия функций. Это важно, поскольку человек способен помнить ограниченное количество фактов. Использование функций является первым шагом к повышению степени абстракции программы и ведет к упрощению ее структуры.

Разделение программы на функции позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Процесс отладки программы, содержащей функции, можно лучше структурировать. Часто используемые функции можно помещать в библиотеки. Таким образом создаются более простые в отладке и сопровождении программы.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (*модули*), компилируемые раздельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (или разными программистами).

Модуль содержит данные и функции их обработки. Другим модулям нежелательно иметь собственные средства обработки этих данных, они должны пользоваться для этого функциями первого модуля. Для того чтобы использовать модуль, нужно знать только его интерфейс, а не все детали его реализации. Чем более независимы модули, тем легче отлаживать программу. Это уменьшает общий объем информации, которую необходимо одновременно помнить при отладке. Разделение программы на максимально обособленные части является сложной задачей, которая должна решаться на этапе проектирования программы.

Скрытие деталей реализации называется *инкапсуляцией*. Инкапсуляция является ключевой идеей как структурного, так и объектно-ориентированного программирования. Пример инкапсуляции — помещение фрагмента кода в функцию и передача всех необходимых ей данных в качестве параметров. Чтобы использовать такую функцию, требуется знать только ее интерфейс, определяемый заголовком (имя, тип возвращаемого значения и типы параметров). Интерфейсом модуля являются заголовки всех функций и описания доступных извне "типов, переменных и констант. Описания глобальных программных объектов во всех модулях программы должны быть согласованы.

*Модульность* в языке C++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строго говоря, раздельная компиляция не является элементом языка, а относится к его реализации).

## 2.1 Функции

### 2.1.1 Объявление и определение функций

***Функция*** — это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Любая программа на C++ состоит из функций, одна из которых должна иметь имя *main* (с нее начинается выполнение программы). Функция начинает выполняться в момент вызова. Любая функция должна быть объявлена и определена. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

***Объявление функции (прототип, заголовок, сигнатура)*** задает ее имя, тип возвращаемого значения и список передаваемых параметров. ***Определение функции*** содержит, кроме объявления, тело функции, представляющее собой последовательность операторов и описаний в фигурных скобках:

Листинг 2.1 Определение и объявление функции

```
[ класс ] тип имя ([ список_параметров ])[throw ( исключения )] { тело функции }
```

- С помощью необязательного модификатора класс можно явно задать область видимости функции, используя ключевые слова *extern* и *static*:
- *extern* — глобальная видимость во всех модулях программы (по умолчанию);
- *static* — видимость только в пределах модуля, в котором определена функция.
- Тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, указывается тип *void*.
- Список параметров определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя (в объявлении имена можно опускать).

В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать. На имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Функцию можно определить как встроенную с помощью модификатора *inline*, который рекомендует компилятору вместо обращения к функции помещать ее код непосредственно в каждую точку вызова. Модификатор *inline* ставится перед типом функции. Он применяется для коротких функций, чтобы снизить накладные расходы на вызов (сохранение и восстановление регистров, передача управления). Директива *inline* носит рекомендательный характер и выполняется компилятором по мере возможности. Использование inline-функций может увеличить объем исполняемой программы. Определение функции должно предшествовать ее вызовам, иначе вместо *inline*-расширения компилятор сгенерирует обычный вызов.

Тип возвращаемого значения и типы параметров совместно определяют тип функции.

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция. Если тип возвращаемого функцией значения не *void*, она может входить в состав выражений или, в частном случае, располагаться в правой части оператора присваивания.

Листинг 2.2 Функция, возвращающая сумму двух целых величин

```
#include <iostream.h>
int sum(int a, int b);      //Объявление функции
int main()
{
      int a=2, b=3, c, d;
      c = sum(a,b);                  //Вызов функции
      cin>>a;
      cout<<sum(c,d);   //Вызов функции
      return 0;
}

int sum(int a, int b)
{
      return (a+b);
}
```

Листинг 2.3 Функция, выводящая на экран поля переданной ей структуры

```
#include <iostream.h>
struct Worker
{
      char fio[30];
      int date, code;
      double salary;
}:
void print_worker(Worker);                                          //объявление функции
int main()
{
      Worker staff[100];
      /* формирование массива staff */,
      for (int i = 0: i<100; i++)
      print_worker(staff[i]);                                       // вызов функции
      return 0;
}
void print_worker(Worker w)         //определение функции
{
      cout « w.fio «'   '« w.date « '  ' « w.code « '   ' « w.salary « endl;
}
```

Все величины, описанные внутри функции, а также ее параметры, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции для того, чтобы при выходе из нее можно было продолжить выполнение вызывающей функции.

При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются. Если этого требуется избежать, при объявлении локальных переменных используется модификатор *static*.

Листинг 2.4 Пример использования статической переменной

```
#include <iostream.h>
void f(int a)
{
int m = 0;
      cout « "n m p\n";
      while (a--)
      {
            static int n = 0;
            int p = 0;
            cout « n++ « '  ' « m++ « '   ' « p++ << '\n';
      }
}
int main()
{
      f(3);
      f(2);
      return 0;
}
```

Статическая переменная n размещается в сегменте данных и инициализируется один раз при первом выполнении оператора, содержащего ее определение. Автоматическая переменная m инициализируется при каждом входе в функцию. Автоматическая переменная p инициализируется при каждом входе в блок цикла. Программа выведет на экран:

| | | |
|---|---|---|
| n | m | p |
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 2 | 2 | 0 |
| n | m | p |
| 3 | 0 | 0 |
| 4 | 1 | 0 |

При совместной работе функции должны обмениваться информацией. Это можно осуществить с помощью глобальных переменных, через параметры и через возвращаемое функцией значение.

### 2.1.2 Глобальные переменные

Глобальные переменные видны во всех функциях, где не описаны локальные переменные с теми же именами, поэтому использовать их для передачи данных между функциями очень легко. Тем не менее это не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению функций в библиотеки общего пользования. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

### 2.1.3 Возвращаемое значение

Механизм возврата из функции в вызвавшую ее функцию реализуется оператором

```
return [ выражение ];
```

Функция может содержать несколько операторов *return* (это определяется потребностями алгоритма). Если функция описана как *void*, выражение не указывается. Оператор *return* можно опускать для функции типа *void*, если возврат из нее происходит перед закрывающей фигурной скобкой, и для функции main. В этой книге для экономии места оператор *return* в функции *main* не указан, поэтому при компиляции примеров выдается предупреждение. Выражение, указанное после *return*, неявно преобразуется к типу возвращаемого функцией значения и передается в точку вызова функции.

Листинг 2.5 Примеры правильного и неправильного возвращения значений

```
int f1(){return 1;}                    // правильно
void f2(){return 1:}                   // неправильно, f2 не должна возвращать значение
double f3(){return 1;}                 // правильно, 1 преобразуется к типу double
```

Нельзя возвращать из функции указатель на локальную переменную, поскольку память, выделенная локальным переменным при входе в функцию, освобождается после возврата из нее

Листинг 2.6 Пример неправильного возвращения ссылки на локальную переменную

```
int* f()
{
      int a = 5:
     return &a: // нельзя!
}
```

### 2.1.4 Параметры функции

Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Параметры, перечисленные в заголовке описания функции, называются *формальными параметрами*, или просто параметрами, а записанные в операторе вызова функции — *фактическими параметрами, или аргументами.*

При вызове функции в первую очередь вычисляются выражения, стоящие на месте аргументов; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего аргумента. При этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение. Существует два способа передачи параметров в функцию: по значению и по адресу.

*При передаче по значению* в стек заносятся копии значений аргументов, и операторы функции работают с этими копиями. Доступа к исходным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

*При передаче по адресу в стек* заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов.

Листинг 2.7 Пример функции с различной передачей параметров

```
#include <iostream.h>
void f(int i, int* j, int& k):
int main()
{
      int i = 1, j = 2, k = 3;
      cout <<"i  j  k \n";
      cout « i «'   '« j «'   '« k «'\n';
      f(i. &j. k):
      cout « i «'  '« j «'  '« k;
      return 0:
}
void f(int i, int* j,  int& k)
{
      i++; (*j)++; k++;
}
```

Результат работы программы:

```
i     j      k
1     2     3
1     3     4
```

Первый параметр (1) передается по значению. Его изменение в функции не влияет на исходное значение. Второй параметр (j) передается по адресу с помощью указателя, при этом для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения в функции требуется операция разыменования. Третий параметр (к) передается по адресу с помощью ссылки.

*При передаче по ссылке* в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются. Поэтому использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования. Использование ссылок вместо передачи по значению более эффективно, поскольку не требует копирования параметров, что имеет значение при передаче структур данных большого объема.

Если требуется запретить изменение параметра внутри функции, используется модификатор const.

```
int f(const char*);
char* t(char* a. const int* b);
```

Таким образом, исходные данные, которые не должны изменяться в функции, предпочтительнее передавать ей с помощью константных ссылок.

По умолчанию параметры любого типа, кроме массива и функции (например, вещественного, структурного, перечисление, объединение, указатель), передаются в функцию по значению.

#### 2.1.4.1 Передача массивов в качестве параметров

При использовании в качестве параметра массива в функцию передается указатель на его первый элемент, иными словами, массив всегда передается по адресу. При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр (в случае массива символов, то есть строки, ее фактическую длину можно определить по положению нуль-символа).

Листинг 2.8 Функция, возвращающая сумму элементов массива

```
#include <iostream.h>
int sum(const Int* mas, const int n):
int const n = 10:
int main()
{
```

```
    int marks[n] = {3, 4. 5, 4, 4};
    cout « "Сумма элементов массива: " « sum(marks, n);
    return 0;
}

int sum(const int* mas, const int n)
{
    // варианты: int sum(int mas[], int n)
    // или     int sumCint mas[n],  int n)
    // (величина п должна быть константой)
    int s = 0:
    for (int i = 0; i<n: i++)
        s += mas[i];
    return s;
}
```

При передаче многомерных массивов все размерности, если они не известны на этапе компиляции, должны передаваться в качестве параметров. Внутри функции массив интерпретируется как одномерный, а его индекс пересчитывается в программе. В приведенном ниже примере с помощью функции подсчитывается сумма элементов двух двумерных массивов. Размерность массива *b* известна на этапе компиляции, под массив а память выделяется динамически.

Листинг 2.9 Передача динамического массива в функцию

```
#indude <stdio.h>
#include <stdlib.h>
int sum(const int *a, const int nstr, const int nstb);
int main()
{
    int b[2][2] = {{2, 2). {4, 3}};
    printf("Сумма элементов b: %d\n", sum(&b[0][0]. 2, 2));
    // имя массива передавать в sum нельзя из-за несоответствия типов
    int i, j. nstr, nstb, *a;
    printf("Введите количество строк и столбцов: \n");
    scanf("%d%d", &nstr, &nstb);
    a = (int *)malloc(nstr * nstb * sizeof(int));
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)
            scanf("%d" , &a[i*nstb + j]);
    printf("Сумма элементов а: %d\n", sum(a, nstr. nstb));
    return 0;
}
int sum(const int *a, const int nstr, const int nstb)
{
    int i, j, s = 0;
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)
            s += a[i * nstb + j];
    return s;
}
```

Для того, чтобы работать с двумерным массивом естественным образом, применить альтернативный способ выделения памяти.

Листинг 2.10 Альтернативный способ выделения памяти под динамический массив

```
#include <iostream.h>
int sum(int **a, const int nstr, const int nstb);
int main()
{
    int nstr, nstb;
    cin » nstr » nstb;
    int **a. i, j;
    // Формирование матрицы а:
    a = new int* [nstr]:
    for (i = 0; i<nstr: i++)
        a[i] = new int [nstb];
    for (i = 0; i<nstr; i++)
        for (j = 0; j<nstb; j++)
            cin » a[i][j];
    cout « sum(a, nstr, nstb);
    return 0;
```

```
}
int sum(int **a, const int nstr, const int nstb)
{
      int.i, j, s = 0:
      for (i = 0; i<nstr; i++)
            for (j = 0; j<nstb; j++)
                  s += a[i][j];
      return s;
}
```

В этом случае память выделяется в два этапа: сначала под столбец указателей на строки матрицы, а затем в цикле под каждую строку, как показано на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.7**. Освобождение памяти должно выполняться в обратном порядке.

### 2.1.4.2 Передача имен функций в качестве параметров

Функцию можно вызвать через указатель на нее. Для этого объявляется указатель соответствующего типа и ему с помощью операции взятия адреса присваивается адрес функции.

Листинг 2.11 Присвоение переменным адреса функции

```
void f(int a ){/*.,.*/.}                    // определение функции
void (*pf)(int);                            // указатель на функцию
pf = &f;                                    // указателю присваивается адрес функции
                                            // (можно написать pf = f;)
pf(10);                                     // функция f вызывается через указатель pf
                                            // (можно написать (*pf)(10) )
```

Для того чтобы сделать программу легко читаемой, при описании указателей на функции используют переименование типов (*typedef*). Можно объявлять массивы указателей на функции (это может быть полезно, например, при реализации меню).

Листинг 2.12 Массив указателей на функцию

```
// Описание типа PF как указателя
// на функцию с одним параметром типа int:
typedef void (*PF)(int);
// Описание и инициализация массива указателей:
PF rnenu[ ] = {&new, &open, &save};
menu[1](10);                                // Вызов функции open
```

Здесь *new, open* и *save* — имена функций, которые должны быть объявлены ранее.

Указатели на функции передаются в подпрограмму таким же образом, как и параметры других типов.

Листинг 2.13 Передача указателей на функцию в качестве параметров

```
#include <iostream.h>
typedef void (*PF)(int);
void fl(PF pf)
{                                           // функция fl получает в качестве параметра указатель типа PF
      pf(5);                                // вызов функции, переданной через указатель
}
void f(int i)
{
      cout « i;
}
int main()
{
      f1(f);
      return 0;
}
```

Тип указателя и тип функции, которая вызывается посредством этого указателя, должны совпадать в точности.

### 2.1.4.3 Параметры со значениями по умолчанию

Чтобы упростить вызов функции, в ее заголовке можно указать значения параметров по умолчанию. Эти параметры должны быть последними в списке и могут опускаться при вызове

функции. Если при вызове параметр опущен, должны быть опущены и все параметры, стоящие за ним. В качестве значений параметров по умолчанию могут использоваться константы, глобальные переменные и выражения.

Листинг 2.14 Описание функций с параметрами по умолчанию

```
int f(int a, int b = 0);
void f(int,  int = 100, char*  = 0);          /* обратите внимание на пробел между * и
                                              - (без него получилась бы операция сложного присваивания *=) */
void err(int errValue = errno);               // errno - глобальная переменная
……….
f(100); f(a, 1);
fl(a); fl(a, 10);  fl(a, 10,  "Vasia");
f1(a, , "Vasia");
```

#### 2.1.4.4 Функции с переменным числом параметров

Если список формальных параметров функции заканчивается многоточием, это означает, что при ее вызове на этом месте можно указать еще несколько параметров. Проверка соответствия типов для этих параметров не выполняется, *char* и *short* передаются как *int,* а *float* — как *double*. В качестве примера можно привести функцию printf, прототип которой имеет вид:

```
int printf(const char*. ...);
```

Это означает, что вызов функции должен содержать по крайней мере один параметр типа *char** и может либо содержать, либо не содержать другие параметры.

Листинг 2.15 Примеры вызова функций с переменным числом параметров

```
printf("Введите исходные данные");                  // один параметр
printf("Сумма: X5.2f рублей", sum);                 // два параметра
printf("%d %d %d %d", a, b, c, d);                  // пять параметров
```

Для доступа к необязательным параметрам внутри функции используются макросы библиотеки *va_start, va_arg* и *va_end*, находящиеся в заголовочном файле *<:stdarg.h>*.

Поскольку компилятор не имеет информации для контроля типов, вместо функций с переменным числом параметров предпочтительнее пользоваться параметрами по умолчанию или перегруженными функциями, хотя можно представить случаи, когда переменное число параметров является лучшим решением.

### 2.1.5 Рекурсивные функции

*Рекурсивной* называется функция, которая вызывает саму себя. Такая рекурсия называется *прямой*. Существует еще *косвенная рекурсия*, когда две или более функций вызывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется. Ясно, что для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

Классическим примером рекурсивной функции является вычисление факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа n, требуется умножить на n факториал числа(n-1). Известно также, что 0! = 1 и 1!=1.

Листинг 2.16 Вычисление факториала с помощью рекурсивной функции

```
long fact(long n)
{
    if (n==0 | | n=l)
```

```
        return 1;
    return (n * fact(n - D);
}

//То же самое можно записать короче:
long fac(long n)
{
    return (n>1) ? n * fact(n - 1) : 1;
}
```

Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно, например, с двоичными деревьями. Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно. Достоинством рекурсии является компактная запись, а недостатками — расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, главное, опасность переполнения стека.

### 2.1.6 Перегрузка функций

Часто бывает удобно, чтобы функции, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если это имя мнемонично, то есть несет нужную информацию, это делает программу более понятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких функций с одним и тем же именем, но с различными типами параметров, называется *перегрузкой функций*.

Компилятор определяет, какую именно функцию требуется вызвать, по типу фактических параметров. Этот процесс называется разрешением перегрузки (перевод английского слова *resolution* в смысле «уточнение»). Тип возвращаемого функцией значения в разрешении не участвует. Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сводится к тому, чтобы использовать функцию с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта функции, определяющей наибольшее значение.

Листинг 2.17 Перегрузка функций поиска максимального из двух параметров

```
// Возвращает наибольшее из двух целых:
int max(int,  int);
// Возвращает подстроку наибольшей длины:
char* max(char*. char*);
// Возвращает наибольшее из первого параметра и длины второго:
int max (int. char*);
// Возвращает наибольшее из второго параметра и длины первого:
int max (char*, int);
void f(int a, int b. char* c, char* d)
{
    cout « max (a, b) « max(c, d) « max(a, c) « max(c, b):
}
```

При вызове функции *max* компилятор выбирает соответствующий типу фактических параметров вариант функции (в приведенном примере будут последовательно вызваны все четыре варианта функции).

Если точного соответствия не найдено, выполняются продвижения порядковых типов в соответствии с общими правилами, например, *bool* и *char* в *int, float* в *double* и т. п. Далее выполняются стандартные преобразования типов, например, *int* в *double* или указателей в *void**. Следующим шагом является выполнение преобразований типа, заданных пользователем, а также поиск соответствий за счет переменного числа аргументов функций. Если соответствие на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке.

Неоднозначность может появиться при:

- преобразовании типа;
- использовании параметров-ссылок;

– использовании аргументов по умолчанию.

Листинг 2.18 Пример неоднозначности при преобразовании типа

```
#include <iostream.h>
float f(float i)
{
      cout « "function float f(float 1)" « endl;
      return i;
}
double f(double i)
{
      cout « "function double f(double i)" « endl;
      return i*2;
}
int main()
{
      float x = 10.09;
      double y = 10.09;
      cout « f(x) « endl;                                                    // Вызывается f(float)
      cout « f(y) « endl;                                                    // Вызывается f(double)
      /* cout « f(10) « endl; Неоднозначность - как преобразовать 10: во float или double? */
      return 0;

}
```

Для устранения этой неоднозначности требуется явное приведение типа для константы 10.

Пример неоднозначности при использовании параметров-ссылок: если одна из перегружаемых функций объявлена как *int f(int a, int b),* а другая — как *int f (int a. int &b),* то компилятор не сможет узнать, какая из этих функций вызывается, так как нет синтаксических различий между вызовом функции, которая получает параметр по значению, и вызовом функции, которая получает параметр по ссылке.

Листинг 2.19 Неоднозначности при использовании аргументов по умолчанию

```
#include <iostream.h>
int f(int a){return a;}
int f(int a, int b =1){return a * b;}
int main()
{
      cout « f(10. 2);      // Вызывается   f(int. int)
      /* cout « f(10):     Неоднозначность - что вызывается:    f(int. int) или
      f(int) ? */
      return 0;
}
```

Ниже приведены правила описания перегруженных функций:

- Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.
- Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.
- Функции не могут быть перегружены, если описание их параметров отличается только модификатором *const* или использованием ссылки (например, *int* и *const int* или *int* и *int&*).

### 2.1.7 Шаблоны функций

Многие алгоритмы не зависят от типов данных, с которыми они работают (классический пример — сортировка). Естественно желание параметризовать алгоритм таким образом, чтобы его можно было использовать для различных типов данных. Первое, что может прийти в голову — передать информацию о типе в качестве параметра (например, одним параметром в функ-

цию передается указатель на данные, а другим — длина элемента данных в байтах). Использование дополнительного параметра означает генерацию дополнительного кода, что снижает эффективность программы, особенно при рекурсивных вызовах и вызовах во внутренних циклах; кроме того, отсутствует возможность контроля типов. Другим решением будет написание для работы с различными типами данных нескольких перегруженных функций, но в таком случае в программе будет несколько одинаковых по логике функций, и для каждого нового типа придется вводить новую.

В C++ есть мощное средство параметризации — *шаблоны*. Существуют шаблоны функций и шаблоны классов (о шаблонах классов см. с. 87). С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тин данных передается функции в виде параметра на этапе компиляции. Компилятор автоматически генерирует правильный код, соответствующий переданному типу. Таким образом, создается функция, которая автоматически перегружает сама себя и при этом не содержит накладных расходов, связанных с параметризацией.

Листинг 2.20 Формат простейшей функции-шаблона

```
template <class Type> заголовок
{
/* тело функции */
}
```

Вместо слова Type может использоваться произвольное имя.

В общем случае шаблон функции может содержать несколько параметров, каждый из которых может быть не только типом, но и просто переменной.

```
template <class A, class B,  int i> void f(){  ...  }
```

Например, функция, сортирующая методом выбора массив из *n* элементов любого типа, в виде шаблона может выглядеть так.

Листинг 2.21 Шаблон функции сортировки массива

```
template <class Type>
void sort_vybor(Type *b. int n)
{
      Type a;  //буферная переменная для обмена элементов
      for (int i = 0:  i<n-1;  i++)
      {
            int imin = i:
            for (int j = i + 1;  j<n; j++)
                  if (b[j] < b[imin])
                        imin = j;
            a = b[i];  b[i] = b[imin]; b[imin] = a;
      }
}
```

Главная функция программы, вызывающей эту функцию-шаблон, может иметь вид.

Листинг 2.22 Примеры вызова шаблона функции сортировки массива

```
#include <iostream.h>
template <class Type> void sort_vybor(Type *b. int n);
 int main()
{
      const int n = 20;
      int i, b[n];
      for (i = 0;  i<n;  i++)
            cin » b[i];
      sort_vybor(b, n);   // Сортировка целочисленного массива
      for (i = 0;  i<n;  i++)
            cout « b[i] «  '   ';
      cout « endl;
      double a[]  = {0.22,  117.  -0.08,  0.21. 42.5};
      sort_vybor(a. 5):   // Сортировка массива вещественных чисел
      for (i = 0; i<5;  i++)
            cout « a[i] « '   ';
      return 0;
}
```

Первый же вызов функции, который использует конкретный тип данных, приводит к созданию компилятором кода для соответствующей версии функции. Этот процесс называется *инстанцированием* шаблона (*instantiation*). Конкретный тип для инстанцирования либо определяется компилятором автоматически, исходя из типов параметров при вызове функции, либо задается явным образом. При повторном вызове с тем же типом данных код заново не генерируется. На месте параметра шаблона, являющегося не типом, а переменной, должно указываться константное выражение.

Листинг 2.23 Пример явного задания аргументов шаблона при вызове

```
template<class X, class Y, class Z> void f(Y, Z):
void g()
{
      f<int, char*, double>("Vasia", 3.0);
      f<int. char*>("Vasia". 3.0);                    // Z определяется как double
      f<int>("Vasia", 3.0);                           // Y определяется как char*, а Z - как double
      // f('Vasia". 3.0);                             ошибка: X определить невозможно
}
```

Чтобы применить функцию-шаблон к типу данных, определенному пользователем (структуре или классу), требуется перегрузить операции для этого типа данных, используемые в функции .

Как и обычные функции, шаблоны функций могут быть перегружены как с помощью *шаблонов*, так *и обычными функциями*.

Можно предусмотреть специальную обработку отдельных параметров и типов с помощью специализации шаблона функции. Допустим, мы хотим более эффективно реализовать общий алгоритм сортировки для целых чисел. В этом случае можно «вручную» задать вариант шаблона функции для работы с целыми числами.

Листинг 2.24 Ручной способ задания шаблонов функции

```
void sort_vibor<int>(int *b.  int n)
{
      ...   // Тело специализированного варианта функции
}
```

Сигнатура шаблона функции включает не только ее тип и типы параметров, но и фактический аргумент шаблона. Обычная функция никогда не считается специализацией шаблона, несмотря на то, что может иметь то же имя и тип возвращаемого значения.

### 2.1.8 Функция main()

Функция, которой передается управление после запуска программы, должна иметь имя main. Она может возвращать значение в вызвавшую систему и принимать параметры из внешнего окружения. Возвращаемое значение должно быть целого типа. Стандарт предусматривает два формата функции:

Листинг 2.25 Формат функции *main*

```
// без параметров:
тип main()
{
      /* ... */
}

// с двумя параметрами:
тип main(int argc,  char* argv[])
{
      /* ... */
}
```

При запуске программы параметры разделяются пробелами. Имена параметров в программе могут быть любыми, но принято использовать *argc* и *argv*. Первый параметр (*argc*) определяет количество параметров, передаваемых функции, включая имя самой программы, второй параметр (*argv*) является указателем на массив указателей типа *char**. Каждый элемент массива содержит указатель на отдельный параметр командной строки, хранящийся в виде *C*-

строки, оканчивающейся *нуль-символом.* Первый элемент массива (*argv[0]*) ссылается на полное имя запускаемого на выполнение файла, следующий (*argv[l]*) указывает на первый параметр, *argv[2]* — на второй параметр, и так далее. Параметр *argv[argc]* должен быть равен 0.

Если функция *main()* ничего не возвращает, вызвавшая система получит значение, означающее успешное завершение. Ненулевое значение означает аварийное завершение. Оператор возврата из *main()* можно опускать.

Листинг 2.26 Функция *main*, осуществляющая вывод на экран массив переданных ей параметров

```
#include <iostream.h>
void main(int argc, char* argv[])
{
      for (int i = 0;  i<argc; i++)
      cout « argv[i] « '\n;
}
```

Пусть исполняемый файл программы имеет имя main.exe и вызывается из командной строки:

```
d:\cpp\main.exe one two three
```

На экран будет выведено:

```
D:\CPP\MAIN.EXE
one
two
three
```

Дополнительно о функциях, их параметрах рекомендуется прочитать в литературе:

3. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 185 - 209.

4. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 169 - 236.

## 2.2 Директивы препроцессора

Препроцессором называется первая фаза компилятора. Инструкции препроцессора называются директивами. Они должны начинаться с символа #, перед которым в строке могут находиться только пробельные символы.

### 2.2.1 Директива #include

Директива *#include <имя_файла>* вставляет содержимое указанного файла в ту точку исходного файла, где она записана. Включаемый файл также может содержать директивы *#include*. Поиск файла, если не указан полный путь, ведется в стандартных каталогах включаемых файлов. Вместо угловых скобок могут использоваться кавычки (" ") — в этом случае поиск файла ведется в каталоге, содержащем исходный файл, а затем уже в стандартных каталогах.

Директива *#include* является простейшим средством обеспечения согласованности объявлений в различных файлах, она включает в них информацию об интерфейсе из заголовочных файлов.

Заголовочные файлы обычно имеют расширение .h и могут содержать:

- определения типов, констант, встроенных функций, шаблонов, перечислений;
- объявления функций, данных, имен, шаблонов;
- пространства имен;
- директивы препроцессора;
- комментарии.

В заголовочном файле не должно быть определений функций и данных. Эти правила не являются требованием языка, а отражают разумный способ использования директивы.

При указании заголовочных файлов стандартной библиотеки расширение .h можно опускать. Это сделано для того, чтобы не ограничивать способы их хранения. Для каждого файла библиотеки *C* с именем *<name.h>* имеется соответствующий файл библиотеки C++ *<cname>*, в котором те же средства описываются в пространстве имен std. Например, директива *#include<cstdio>* обеспечивает те же возможности, что и *#include <stdio.h>*, но при обращении к стандартным функциям требуется указывать имя пространства имен std.

### 2.2.2 Директива #define

Директива #define определяет подстановку в тексте программы. Она используется для определения:

- символических констант:

```
#define имя текст_подстановки                    //все вхождения имени заменяются на текст подстановки;
```

- *макросов*, которые выглядят как функции, но реализуются подстановкой их текста в текст программы:

```
#define имя( параметры ) текст_подстановки
```

- символов, управляющих условной компиляцией. Они используются вместе с директивами *#ifdef* и *#ifndef*. Формат:

```
#define имя
```

Листинг 2.27 Примеры использования директивы *#define*

```
#define VERSION I
#define VASIA "Василий Иванович"
#define MAX(x.y)  ((x)>(y)?(x):(y))
#define MUX
```

Имена рекомендуется записывать прописными буквами, чтобы зрительно отличать их от имен переменных и функций. Параметры макроса используются при макроподстановке, например, если в тексте программы используется вызов макроса *y = MAX(sum1, sum2);* он будет заменен на

```
y = ((sum1)>(sum2)?(sum1):(sum2));
```

Отсутствие круглых скобок может привести к неправильному порядку вычисления, поскольку препроцессор не оценивает вставляемый текст с точки зрения синтаксиса. Например, если к макросу *#define sqr(x) (x*x)* обратиться как *sqr(y+1),* в результате подстановки получится выражение *(y+1*y+1).* Макросы и символические константы унаследованы из языка *C*, при написании программ на *C++* их следует избегать. Вместо символических констант предпочтительнее использовать *const* или *enum*, а вместо макросов – встроенные функции или шаблоны.

### 2.2.3 Директивы условной компиляции

Директивы условной компиляции *#if, #ifdef* и *#ifndef* применяются для того, чтобы исключить компиляцию отдельных частей программы. Это бывает полезно при отладке или, например, при поддержке нескольких версий программы для различных платформ.

Листинг 2.28 Формат директивы *#if*

```
#if константное_выражение
     [ #elif константное_выражение
          [ #elif константное_выражение
          ~]
     [ #else
     -]
#endif
```

Количество директив *#elif* — произвольное. Исключаемые блоки кода могут содержать как описания, так и исполняемые операторы.

Листинг 2.29 Пример условного включения различных версий заголовочного файла

```
#if VERSION — 1
      #define INCFILE "versl.h" #elif VERSION — 2
      #define INCFILE "vers2.h"  /* и так далее */ #else
      #define INCFILE "versN.h" #endif
#include INCFILE
```

В константных выражениях может использоваться проверка, определена ли константа, с помощью *defined(имя_константы)*.

Листинг 2.30 примеры использования константных выражений

```
#if defIned(__BORLANDC__) && __BORLANDC__ == 0x530                    // BC5.3:
typedef istream_iterator<int, char, char_traits<char>, ptrdiff_t> istrean_iter;
#elif defined(__BORLAND__)                                           // BC5.2:
typedef istream_terator<int, ptrdiff_t> istream_iter;
#else                                                                // VC5.0:
typedef istream_iterator<int> istreamjter;
#endif
```

Другое назначение директивы — временно закомментировать фрагменты кода/

Листинг 2.31 Пример комментария фрагмента кода

```
#if 0
      int I,  j;
      double x, y;
#endif
```

Поскольку допускается вложенность директив, такой способ весьма удобен. Наиболее часто в программах используются директивы *#ifdef* и *#ifndef*, позволяющие управлять компиляцией в зависимости от того, определен ли с помощью директивы *#define* указанный в них символ (хотя бы как пустая строка например, *#define 32_BIT_SUPPORT*).

Листинг 2.32 Пример условной компиляции

```
#ifdef символ
// Расположенный ниже код компилируется, если символ определен
 #ifndef символ
// Расположенный ниже код компилируется, если символ не определен
```

Действие этих директив распространяется до первого *#elif, #else* или *#endif*.

Директива *#ifndef* часто применяется для того, чтобы обеспечить включение заголовочного файла только один раз/

Листинг 2.33 Пример однократного включения заголовочного файла

```
#ifndef HEADERJNCLUDED
#include "myheader.h"
#define HEADERJNCLUDED
#endif
```

### 2.2.4 Директива #undef

Директива *#undef* имя удаляет определение символа. Используется редко, например, для отключения какой-либо опции компилятора.

### 2.2.5 Предопределенные макросы

В C++ определено несколько макросов, предназначенных в основном для того, чтобы выдавать информацию о версии программы или месте возникновения ошибки.

*__cplusplus* - определен, если программа компилируется как файл *C++*. Многие компиляторы при обработке файла с расширением *.c* считают, что программа написана на языке *C*. Использование этого макроса позволяет указать, что можно использовать возможности C++.

Листинг 2.34 Разделения языков *C* и *C++*

```
#ifdef __cplusplus
// Действия, специфические для C++
#endif
```

Применяется, если требуется переносить код из C в C++ и обратно.

*__DATE__* - содержит строку с текущей датой в формате месяц день год.

Листинг 2.35 Вывод на экран даты компиляции программы

```
printf("Дата компиляции - #s \n",__DATE__);
```

*__FILE__* - содержит строку с полным именем текущего файла.

*__LINE__* — текущая строка исходного текста.

*__TIME__* — текущее время, например:

```
printf("Ошибка в файле %s \n Время компиляции: %s\n ",__FILE__,__TIME__);
```

Дополнительно о директивах препроцессора рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 203 - 205.

2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 263 - 280.

## 2.3 Области действия идентификаторов

Каждый программный объект имеет область действия, которая определяется видом и местом его объявления. Существуют следующие области действия: блок, файл, функция, прототип функции, класс и поименованная область.

**Блок.** Идентификаторы, описанные внутри блока, являются локальными. Область действия идентификатора начинается в точке определения и заканчивается в конце блока, видимость — в пределах блока и внутренних блоков, время жизни — до выхода из блока. После выхода из блока память освобождается.

**Файл.** Идентификаторы, описанные вне любого блока, функции, класса или пространства имен, имеют глобальную видимость и постоянное время жизни и могут использоваться с момента их определения.

**Функция.** Единственными идентификаторами, имеющими такую область действия, являются метки операторов. В одной функции все метки должны различаться, но могут совпадать с метками других функций.

**Прототип функции.** Идентификаторы, указанные в списке параметров прототипа (объявления) функции, имеют областью действия только прототип функции.

**Класс**. Элементы структур, объединений и классов (за исключением статических элементов) являются видимыми лишь в пределах класса. Они образуются при создании переменной указанного типа и разрушаются при ее уничтожении.

**Поименованная область.** C++ позволяет явным образом задать область определения имен как часть глобальной области с помощью оператора *namespace*.

Напоминаем, что область видимости совпадает с областью действия за исключением ситуации, когда во вложенном блоке описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя он и входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя операцию доступа к области видимости ::. Способ обратиться к скрытой локальной переменной отсутствует.

В каждой области действия различают так называемые пространства имен. Пространство имен — область, в пределах которой идентификатор должен быть уникальным. В разных пространствах имена могут совпадать, поскольку разрешение ссылок осуществляется по контексту идентификатора в программе.

Листинг 2.36 Пример совпадения имен

```
struct Node{
    int Node;
    int i; }Node;
```

В данном случае противоречия нет, поскольку имена типа, переменной и элемента структуры относятся к разным пространствам. В C++ определено четыре раздельных класса идентификаторов, в пределах каждого из которых имена должны быть уникальными.

- К одному пространству имен относятся имена переменных, функций, типов, определенных пользователем (*typedef*) и констант перечислений в пределах одной области видимости. Все они, кроме имен функций, могут быть переопределены во вложенных блоках.
- Другой класс имен образуют имена типов перечислений, структур, классов и объединений. Каждое имя должно отличаться от имен других типов в той же области видимости.
- Отдельный класс составляют элементы каждой структуры, класса и объединения. Имя элемента должно быть уникально внутри структуры, но может совпадать с именами элементов других структур. Метки образуют отдельное пространство имен.

### 2.3.1 Внешние объявления

Любая функция автоматически видна во всех модулях программы. Если требуется ограничить область действия функции файлом, в котором она описана, используется модификатор static.

Для того чтобы сделать доступной в нескольких модулях переменную или константу, необходимо:

- определить ее ровно в одном модуле как глобальную;
- в других модулях объявить ее как внешнюю с помощью модификатора *extern.*

Другой способ — поместить это объявление в заголовочный файл и включить его в нужные модули.

*Все описания одной и той же переменной должны быть согласованы.*

Листинг 2.37 Пример описания двух глобальных переменных в файлах one.cpp и two.cpp с помощью заголовочного файла my_header.h

```
// my_header.h - внешние объявления
extern int a;
extern double b:

// one.cpp
#include "my_header.h"
int a;

// two.cpp
#include "my_header.h"
double b;
```

Обе переменные доступны в файлах one.cpp и two.cpp.

Если переменная описана как static, область ее действия ограничивается файлом, в котором она описана.

При *описании типа* следует придерживаться *правила одного определения,* то есть тип, используемый в программе, должен быть определен ровно один раз. Как правило, это делается в заголовочном файле, который затем подключается к модулям, использующим этот тип. Нарушение этого правила приводит к ошибкам, которые трудно обнаружить, поскольку компиляторы, как правило, не обладают возможностью сличать определения одного и того же типа в различных файлах.

### 2.3.2 Поименованные области

Поименованные области служат для логического группирования объявлений и ограничения доступа к ним. Чем больше программа, тем более актуально использование поименованных областей. Простейшим примером применения является отделение кода, написанного одним человеком, от кода, написанного другим. При использовании единственной глобальной области видимости формировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имен. Использование поименованных областей препятствует доступу к ненужным средствам.

Объявление поименованной области (ее также называют пространством имен) имеет формат.

Листинг 2.38 Формат поименованной области

```
namespace [ имя_области ]
{
    /* Объявления */
}
```

Поименованная область может объявляться неоднократно, причем последующие объявления рассматриваются как расширения предыдущих. Таким образом, поименованная область может объявляться и изменяться за рамками одного файла.

Если имя области не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора, различного для каждого модуля. Объявление объекта в неименованной области равнозначно его описанию как глобального с модификатором static. Помещать объявления в такую область полезно для того, чтобы сохранить локальность кода, Нельзя получить доступ из одного файла к элементу неименованной области другого файла.

Листинг 2.39 Пример поименованной области

```
namespace demo
{
    int i = 1;
    int k = 0;
    void func1(int);
    void func2(int) { /* ., */ }
}
namespace demo                          //расширение
{
    // int i = 2;                       //Неверно  - двойное объявление переменной
    void func1(double);                 //Перегрузка
    void func2(int);                    //Верно (повторное объявление)
}
```

В объявлении поименованной области могут присутствовать как объявления, так и определения. Логично помещать в нее только объявления, а определять их позднее с помощью имени области и оператора доступа к области видимости ::

```
void demo:: fund (int) { /* ... */ }
```

Это применяется для разделения интерфейса и реализации. Таким способом нельзя объявить новый элемент пространства имен.

Объекты, объявленные внутри области, являются видимыми с момента объявления. К ним можно явно обращаться с помощью имени области и оператора доступа к области видимости ::.

```
demo::i = 100;
demo::func2(10);
```

Если имя часто используется вне своего пространства, можно объявить его доступным с помощью *оператора using*:

```
using demo::i;
```

После этого можно использовать имя без явного указания области. Если требуется сделать доступными все имена из какой-либо области, используется оператор *using namespace*:

```
using namespace demo;
```

Операторы using и using namespace можно использовать и внутри объявления поименованной области, чтобы сделать в ней доступными объявления из другой области:

```
namespace Department_pf_App1ied_Mathematics{ using demo::i;
```

Имена, объявленные в поименованной области явно или с помощью оператора *using*, имеют приоритет, но отношению к именам, объявленным с помощью оператора *using namespace* (это имеет значение при включении нескольких поименованных областей, содержащих совпадающие имена).

Короткие имена пространств имен могут войти в конфликт друг с другом, а длинные непрактичны при написании реального кода, поэтому допускается вводить синонимы имен:

```
namespace DAM = Department_of_Applied_Mathematics;
```

Пространства имен стандартной библиотеки. Объекты стандартной библиотеки определены в пространстве имен *std*. Например, объявления стандартных средств ввода/вывода С в заголовочном файле <*stdio.h*> помещены в пространство имен следующим образом:

Листинг 2.40 Пример пространства имен

```
// stdio.h
namespace std{
int feof(FILE *f);
using namespace std;
```

Это обеспечивает совместимость сверху вниз. Для тех, кто не желает присутствия неявно доступных имей, определен новый заголовочный файл <*cstdio*>:

```
// cstdio namespace std{ int feof(FILE *f);
```

Если в программу включен файл <cstdio>, нужно указывать имя пространства имен явным образом:

```
std::feof(f);
```

Механизм пространств имен вместе с директивой #include обеспечивают необходимую при написании больших программ гибкость путем сочетания логического группирования связанных величин и ограничения доступа.

Как правило, в любом функционально законченном фрагменте программы можно выделить интерфейсную часть (например, заголовки функций, описания типов), необходимую для использования этого фрагмента, и часть реализации, то есть вспомогательные переменные, функции и другие средства, доступ к которым извне не требуется. Пространства имен позволяют скрыть детали реализации и, следовательно, упростить структуру программы и уменьшить количество потенциальных ошибок. Продуманное разбиение программы на модули, четкая спецификация 'интерфейсов и ограничение доступа позволяют организовать эффективную работу над проектом группы программистов.

Дополнительно об областях действия идентификаторов рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования С++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 203 - 205.

2. Подбельский В.В. Язык С++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 263 - 280.

## 2.4 Динамические структуры данных

Любая программа предназначена для обработки данных, от способа организации которых зависят алгоритмы работы, поэтому выбор структур данных должен предшествовать созданию алгоритмов. Выше были рассмотрены стандартные способы организации данных, предоставляемые языком *C++,* — основные и составные типы. Наиболее часто в программах используются массивы, структуры и их сочетания, например, массивы структур, полями которых являются массивы и структуры.

Память под данные выделяется либо на этапе компиляции (в этом случае необходимый объем должен быть известен до начала выполнения программы, то есть задан в виде константы), либо во время выполнения программы с помощью операции *new* или функции *malloc* (необходимый объем должен быть известен до распределения памяти.). В обоих случаях выделяется непрерывный участок памяти.

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память выделяется по мере необходимости отдельными блоками, связанными друг с другом с помощью указателей. Такой способ организации данных называется *динамическими структурами данных,* поскольку их размер изменяется во время выполнения программы. Из динамических структур в программах чаще всего используются *линейные списки, стеки, очереди* и *бинарные деревья.* Они различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки, поскольку упорядочивание динамических структур не требует перестановки элементов, а сводится к изменению указателей на эти элементы. Например, если в процессе выполнения программы требуется многократно упорядочивать большой массив данных, имеет смысл организовать его в виде линейного списка. При решении задач поиска элемента в тех случаях, когда важна скорость, данные лучше всего представить в виде бинарного дерева.

*Элемент* любой динамической структуры данных представляет собой структуру (в смысле *struct*), содержащую по крайней мере два поля: для хранения данных и для указателя. Полей данных и указателей может быть несколько. Поля данных могут быть любого типа: основного, составного или типа указатель. Описание простейшего элемента (компоненты, узла) выглядит следующим образом:

Листинг 2.41 Описание простейшего элемента динамической структуры данных

```
struct Node
{
    Data d:    // тип данных Data должен быть определен ранее
    Node *p;
};
```

Рассмотрим реализацию основных операций с динамическими структурами данных (в дальнейшем будет приведен пример реализации списка в виде шаблона класса).

### 2.4.1 Линейные списки

Самый простой способ связать множество элементов — сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называется *однонаправленным (односвязным).* Если добавить в каждый элемент вторую ссылку — на предыдущий элемент, получится *двунаправленный список (двусвязиый),* если последний элемент связать указателем с первым, получится *кольцевой список.*

Каждый элемент списка содержит *ключ,* идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол,

любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, к примеру, ветеранов труда ключом будет стаж. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять следующие *операции:*

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

Рассмотрим двунаправленный линейный список. Для формирования списка и работы с ним требуется иметь по крайней мере один указатель — на начало списка. Удобно завести еще один указатель — на конец списка. Для простоты допустим, что список состоит из целых чисел, то есть описание элемента списка выглядит следующим образом.

Листинг 2.42 Элемент двухсвязного списка

```
struct Node
{
      int d;
      Node *next;
      Node *prev;
};
```

Ниже приведена программа, которая формирует список из 5 чисел, добавляет число в список, удаляет число из списка и выводит список на экран. Указатель па начало списка обозначен *pbeg*, на конец списка — *pend*, вспомогательные указатели – *pv* и *pkey*.

Листинг 2.43 Программа, формирующая список из 5 чисел

```
#include <iostream.h>
 struct Node
{
      int d;
      Node *next;
      Node *prev;
};
//-------------------------------------------------------------------------
Node *first(int d);
void add(Node **pend, int d);
Node *find(Node * const pbeg, int i);
bool remove(Node **pbeg, Node **pend, int key);
Node *insert(Node * const pbeg, Node **pend, int key. int d);
//-----------------------------------------------------
int main()
{
      Node *pbeg = first(1);                          // Формирование первого элемента списка
      Node *pend = pbeg;                              // Список заканчивается, едва начавшись
      // Добавление в конец списка четырех элементов 2, 3, 4, и 5:
      for (int i - 2; i<6; i++)
            add(&pend, i);
      // Вставка элемента 200 после элемента 2
      Insert(pbeg, &pend, 2, 200);
      // Удаление элемента 5:
      if(!remove (&pbeg, &pend. 5))
            cout « "не найден";
      Node *pv = pbeg;
      while (pv)
      {                                               // вывод списка на экран
            cout « pv->d« ' ';
            pv = pv->next;
      }
      return 0;
}
```

```
// Формирование первого элемента
Node * first(int d)
{
      Node *pv = new Node;
      pv->d = d;
      pv->next = 0;
      pv->prev = 0;
      return pv;
}

// Добавление в конец списка
void add(Node **pend, int d)
{
      Node *pv = new Node;
      pv->d = d;
      pv->next = 0;
      (*pend)->next = pv;
      *pend = pv;
}

// Поиск элемента по ключу
Node * find(Node * const pbeg, int d)
{
      Node *pv - pbeg;
      while (pv)
      {
            if(pv-xj == d)
                  break;
            pv = pv->next;
      }
      return pv;
}

// Удаление элемента
bool remove(Node **pbeg, Node **pend. int key)
{
      if(Node *pkey = find(*pbeg. key))
      {                                                                  // 1
      if (pkey == *pbeg)
      {                                                                  // 2
            *pbeg = (*pbeg)-^next;
            (*pbeg)->prev = 0;
      }
      else
      if (pkey == *pend)
      {                                                                  // 3
            *pend = (*pend)->prev;
            (*pend)->next =0;
      }
      else
      {                                                                  // 4
            (pkey->prev)->next = pkey->next:
            (pkey->next)->prev = pkey->prev;
      }
      delete pkey;
      return true;                                                       // 5
      }
      return false;                                                      // 6
}
// Вставка элемента
Node * inser(Node * const pbeg, Node **pend, int key, int d)
{
      if(Node *pkey = find(pbeg, key))
      {
            Node *pv f new Node;
            pv->d = d;                    // 1 - установление связи нового узла с последующим:
            pv->next = pkey->next;        // 2 - установление связи нового узла с предыдущим:
            pv->prev - pkey;              // 3 - установление связи предыдущего узла с новым:
            pkey->next = pv;              // 4 - установление связи последующего узла с новым:
            if( pkey ! = *pend)
                  (pv->next)->prev = pv;  // Обновление, указателя на конец списка,
            // если узел вставляется в конец:
            else
                  *pend = pv;
            return pv;
      }
      return 0;
}
```

Результат работы программы: 1 2 200 3 4

Все параметры, не изменяемые внутри функций, должны передаваться с модификатором const. Указатели, которые могут измениться (например, при удалении из списка последнего элемента указатель на конец списка требуется скорректировать), передаются по адресу.

Рассмотрим подробнее *функцию удаления элемента из списка remove*. Ее параметрами являются указатели на начало и конец списка и ключ элемента, подлежащего удалению. В строке 1 выделяется память под локальный указатель *pkey*, которому присваивается результат выполнения функции нахождения элемента по ключу *find*. Эта функция возвращает указатель на элемент в случае успешного поиска и 0, если элемента с таким ключом в списке нет. Если *pkey* получает ненулевое значение, условие в операторе *if* становится истинным (элемент существует), и управление передается оператору 2, если нет — выполняется возврат из функции со значением *false* (оператор 6).

Удаление из списка происходит по-разному в зависимости от того, находится элемент в начале списка, в середине или в конце. В операторе 2 проверяется, находится ли удаляемый элемент в начале списка — в этом случае следует скорректировать указатель *pbeg* на начало списка так, чтобы он указывал на следующий элемент в списке, адрес которого находится в поле *next* первого элемента. Новый начальный элемент списка должен иметь в своем поле указателя на предыдущий элемент значение 0.

Если удаляемый элемент находится в конце списка (оператор 3), требуется сместить указатель *pend* конца списка на предыдущий элемент, адрес которого можно получить из поля *prev* последнего элемента. Кроме того, нужно обнулить для нового последнего элемента указатель на следующий элемент. Если удаление происходит из середины списка, то единственное, что надо сделать, — обеспечить двустороннюю связь предыдущего и последующего элементов. После корректировки указателей память из-под элемента освобождается, и функция возвращает значение true.

Работа *функции вставки элемента* в список проиллюстрирована на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.1**. Номера около стрелок соответствуют номерам операторов в комментариях.

Сортировка связанного списка заключается в изменении связей между элементами. Алгоритм состоит в том, что исходный список просматривается, и каждый элемент вставляется в новый список на место, определяемое значением его ключа.

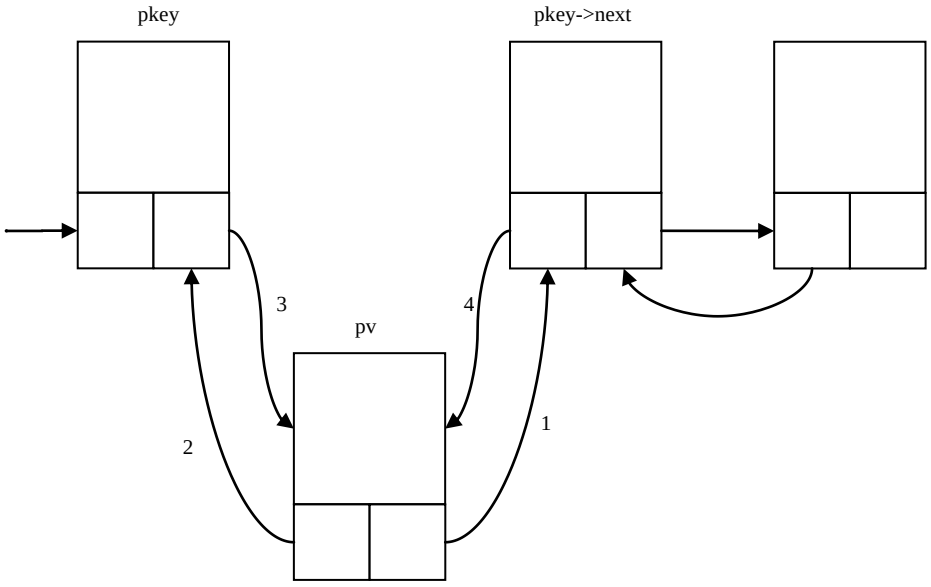


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.1. Вставка элемента в список

Ниже приведена функция формирования упорядоченного списка (предполагается, что первый элемент существует).

Листинг 2.44 Функция сортировки линейного списка

```
void add_sort(Node **pbeg, Node **pend, int d)
{
      Node *pv - new Node;                              // добавляемый элемент
      pv->d = d;
      Node * pt - *pbeg:
      while (pt)
      {     // просмотр списка
            if (d < pt->d)
            {                                           // занести перед текущим элементом (pt)
                  pv->next = pt;
                  if (pt == *pbeg)
                  {                                     // в начало списка
                        pv->prev = 0;
                        *pbeg = pv;
                  }
                  else
                  {                                     // в середину списка
                        (pt->prev)->next = pv;
                        pv->prev = pt->prev;
                  }
                  pt->prev - pv;
                  return;
            }
            pt == pt->next;
      }
      // в конец списка
      pv->next = 0;
      pv->prev = *pend;
      (*pend)->next = pv;
      *pend = pv;
}
```

### 2.4.2 Стеки

*Стек* — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания *LIFO* (*last in – first out*, последним пришел — первым ушел). Стек проще всего представить себе как закрытую с одного конца узкую трубу, в которую бросают мячи. Достать первый брошенный мяч можно только после того, как вынуты все остальные. Кстати, сегмент стека назван так именно потому, что память под локальные переменные выделяется по принципу *LIFO*. Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

Ниже приведена программа, которая формирует стек из пяти целых чисел (1, 2, 3, 4, 5) и выводит его на экран. Функция помещения в стек по традиции называется *push*, а выборки — *pop*. Указатель для работы со стеком (*top*) всегда ссылается на его вершину.

Листинг 2.45 Пример работы со стеком

```
#include <iostream.h>
struct Node
{
      int d;
      Node *p;
};
Node * first(int d);
void push(Node **top, int d);
int pop(Node **top);
//-----------------------------------------------------
int main()
{
      Node *top = first(l):
      for (int i = 2;  i<6;  i++)
            push (&top,  i);
      while (top)
                  cout « pop(Stop) « '   ';
```

```
    return 0;
}

// Начальное формирование стека
Node * first(int d)
{
    Node *pv = new Node;
    pv->d = d;
    pv->p = 0;
    return pv;
}
// Занесение в стек
void push(Node **top. int d)
{
    Node *pv = new Node;
    pv->d = d;
    pv->p = *top;
    *top = pv;
}
// Выборка из стека
int pop(Node **top)
{
    int temp = (*top)->d;
    Node *pv = *top;
    top = (*top)->p;
    delete pv;
    return temp;
}
```

Результат работы программы:

```
5 4 3 2 1
```

### 2.4.3 Очереди

*Очередь* — это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания *FIFO* (*first in — first out*, первым пришел — первым ушел). Очередь проще всего представить себе, постояв в ней час-другой. В программировании очереди применяются, например при моделировании, диспетчеризации задач операционной системой, буферизованном вводе/выводе.

Ниже приведена программа, которая формирует очередь из пяти целых чисел и выводит ее на экран. Функция помещения в конец очереди называется *add*, а выборки — *del*. Указатель на начало очереди называется *pbeg*, указатель на конец — *pend*.

Листинг 2.46 Пример работы с очередью

```
#include <iostream.h>
struct Node
{
    int d;
    Node *p;
}
Node * first(int d);
void add(Node **pend. int d);
int del(Node **pbeg);
int main()
{
    Node *pbeg = first(1);
    Node *pend = pbeg:
    for (int i = 2; i<6; i++)
        add(&pend, i)
    while (pbeg)
        cout « del(&pbeg) « '  ';
    return 0;
}
// Начальное формирование очереди
Node * flrst(int d)
{
    Node *pv = new Node;
    pv->d - d;
    pv->p =0;
```

```
    return pv;
}
// Добавление в конец
void add(Node **pend, int d)
{
    Node *pv = new Node;
    pv->d = d;
    pv->p = 0;
    (*pend)->p = pv;
    *pend = pv;
}

int del(Node **pbeg)
{
    int temp = (*pbeg)->d;
    Node *pv= *pbeg;
    *pbeg = (*pbeg)->p;
    delete pv;
    return temp;
}
```

Результат работы программы:

```
12 3 4 5
```

### 2.4.4 Бинарные деревья

*Бинарное дерево* — это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева.

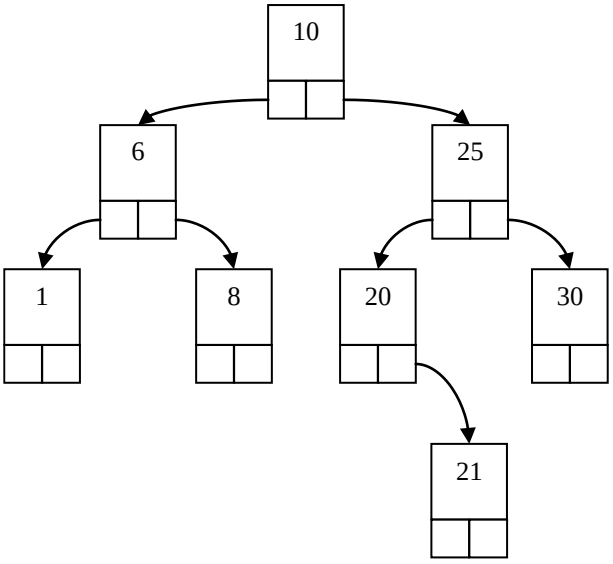


Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2. Бинарное дерево

На рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2** приведен пример бинарного дерева (корень обычно изображается сверху, впрочем, книгу можно и перевернуть). Узел, не имеющий поддеревьев, называется *листом.* Исходящие узлы называются предками, входящие — потомками. *Высота дерева* определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска.* Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму количества узлов.

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью *рекурсивных алгоритмов.*

Листинг 2.47 Рекурсивный обход всех узлов дерева

```
function way_around ( дерево )
{
    way__around ( левое поддерево )
    посещение корня
    way_around ( правое поддерево )
}
```

Можно обходить дерево и в другом порядке, например, сначала корень, потом поддеревья, но приведенная функция позволяет получить на выходе отсортированную последовательность ключей, поскольку сначала посещаются вершины с меньшими ключами, расположенные в левом поддереве. Результат обхода дерева, изображенного на рис. 3.3:

```
1, 6, 8,  10, 20, 21, 25, 30
```

Если в функции обхода первое обращение идет к правому поддереву, результат обхода будет другим:

```
30, 25, 21, 20, 10, 8, 6,  1
```

Таким образом, деревья поиска можно применять для сортировки значений. При обходе дерева узлы не удаляются.

Для бинарных деревьев определены операции:

- включения узла в дерево;
- поиска по дереву;
- обхода дерева;
- удаления узла.

Для каждого рекурсивного алгоритма можно создать его нерекурсивный эквивалент. В приведенной ниже программе реализована нерекурсивная *функция поиска по дереву с включением* и рекурсивная функция обхода дерева. Первая функция осуществляет поиск элемента с заданным ключом. Если элемент найден, она возвращает указатель на него, а если нет — включает элемент в соответствующее место дерева и возвращает указатель на него. Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка.

Программа формирует дерево из массива целых чисел и выводит его на экран.

Листинг 2.48 Программа формирования дерева из массива целых чисел

```
#include <iostream,h>
struct Node
{
      int d;
      Node *left;
      Node *right;
};
Node* first(int d);
Node* search_insert(Node *root, nt d);
void print_tree(Node *root. Int i);
int main()
{
      int b[] = {10,  25,  20, 6,  21, 8,  1, 30}:
      Node *root = first(b[0]);
      for (int i = 1; i<8; i++)
            search_insert(root, b[i])
      print_tree(root.  0);
      return 0:
}
//---------------------------------------
// Формирование первого элемента дерева
Node * first(int d)
{
      Node *pv = new Node;
      pv->d = d;
      pv->left = 0;
      pv->right = 0;
      return pv;
}
// Поиск с включением
Node * search_insert(Node *root, int d)
{
      Node *pv = root.*prev;
      bool found - false;
      while (pv && !found)
      {
            prev - pv;
```

```
            if    (d.= pv->d)
                  found = true;
            else
                  if (d < pv->d)
                  pv  = pv->left:
                  else
                        pv  = pv->right:
      }
      if (found)
            return pv;
      // Создание нового узла
      Node *pnew = new Node;
      pnew->d = d;
      pnew->left = 0;
      pnew->right = 0;
       if (d < prev->d)
            // Присоединение к левому поддереву предка
            prev->left = pnew;
      else
            // Присоединение к правому поддереву предка
            prev->right = pnew:
      return pnew;
}
// Обход дерева
void print_tree(Node *p, int level)
{
      if (p)
      {
            print_tree(p->left, level +1);  // вывод левого поддерева
            for (int i = 0: i<level; i++)
                  cout « "  ";
            cout « p->d « endl;        // вывод корня поддерева
            print_tree(p->right, level +1);  // вывод правого поддерева
      }
}
```

Текущий указатель для поиска по дереву обозначен *pv*, указатель на предка *pv* обозначен *prev*, переменная *pnew* используется для выделения памяти под включаемый в дерево узел. Рекурсии удалось избежать, сохранив всего одну переменную (*prev*) и повторив при включении операторы, определяющие, к какому поддереву присоединяется новый узел.

Результат работы программы для дерева, изображенного на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.3**:

```
            1
      6
            8
10
            20
                  21
      25
            30
```

Рисунок **Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.3. Результаты программы листинга 2.48

Рассмотрим подробнее *функцию обхода дерева.* Вторым параметром в нее передается целая переменная, определяющая, на каком уровне находится узел. Корень находится па уровне 0. Дерево печатается по горизонтали так, что корень находится слева (посмотрите па результат работы программы, наклонив голову влево, и сравните с рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.3**). Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, отсортированный по возрастанию массив будет выведен в столбик. Заметьте, что функция обхода дерева длиной всего в несколько строк может напечатать дерево любого размера — ограничением является только размер стека.

*Удаление узла из дерева* представляет собой не такую простую задачу, поскольку удаляемый узел может быть корневым, содержать две, одну или пи одной ссылки на поддеревья. Для узлов, содержащих меньше двух ссылок, удаление тривиально. Чтобы сохранить упорядоченность дерева при удалении узла с двумя ссылками, его заменяют па узел с самым близким к нему ключом. Это может быть самый левый узел его правого поддерева или самый правый узел левого поддерева (например, чтобы удалить из дерева на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2** узел с ключом 25, его нужно заменить на 21 или 30, узел 10 заменяется на 20 или 8, и т. д.). Реализация функции удаления из дерева оставлена читателю для самостоятельной работы.

### 2.4.5 Реализация динамических структур с помощью массивов

Операции выделения и освобождения памяти — дорогое удовольствие, поэтому если максимальный размер данных можно определить до начала использования и в процессе работы он не изменяется (например, при сортировке содержимого файла), более эффективным может оказаться однократное выделение непрерывной области памяти. Связи элементов при этом реализуются не через указатели, а через вспомогательные переменные или массивы, в которых хранятся номера элементов.

Проще всего реализовать таким образом стек. Кроме массива элементов, соответствующих типу данных стека, достаточно иметь одну переменную целого типа для храпения индекса элемента массива, являющегося вершиной стека. При помещении в стек индекс увеличивается на единицу, а при выборке — уменьшается.

Для реализации очереди требуются две переменных целого типа — для хранения индекса элементов массива, являющихся началом и концом очереди.

Для реализации линейного списка требуется вспомогательный массив целых чисел и еще одна переменная, например:

```
10 25 20  6  21  8    1    30 - массив данных
1   2   3   4  5    6    7    1   - вспомогательный массив
0     - индекс первого элемента в списке
```

*i*-й элемент вспомогательного массива содержит для каждого *i*-го элемента массива данных индекс следующего за ним элемента. Отрицательное число используется как признак конца списка. Тот же массив после сортировки:

```
10 25 20  6    21 8    1    30 - массив данных
2   7   4    5   1   0    3    1   -  вспомогательный массив
6   - индекс первого элемента в списке
```

Для создания бинарного дерева можно использовать два вспомогательных массива (индексы вершин его правого и левого поддерева). Отрицательное число используется как признак пустой ссылки. Например, дерево, приведенное на рис. Error! Use the Home tab to apply Заголовок 1 to the text that you want to appear here.**.2**, можно представить следующим образом:

```
10   25   20  6    21   8    1   30 - массив данных
3    2    1     6    1    1     1   1 - левая ссылка
1    7    4     5    1    1     1   1 - правая ссылка
```

Память под такие структуры можно выделить либо на этапе компиляции, если размер можно задать константой, либо во время выполнения программы.

Листинг 2.49 Статический и динамический способ выделения памяти

```
struct Node
{
      Data d;                                  // тип данных Data должен быть определен ранее
      int i;
}
Node spisok1[100];                             // на этапе компиляции
Node *pspisok2 = new Node[m];                  // на этапе выполнения
```

При работе с подобными структурами необходимо контролировать возможный выход индексов за границу массива.

Приведенный выше способ реализации позволяет использовать преимущества динамических структур (например, сортировать структуры из громоздких элементов данных без их физического перемещения в памяти), и при этом не расходовать время на выделение и освобождение памяти для каждого элемента данных.

# 3. Объектно-ориентированное программирование

В окончательном виде любая программа представляет собой набор инструкций процессора. Все, что написано на любом языке программирования, — более удобная, упрощенная запись этого набора инструкций, облегчающая написание, отладку и последующую модификацию программы. Чем выше уровень языка, тем в более простой форме записываются одни и те же действия. Например, для реализации цикла на языке ассемблера требуется записать последовательность инструкций, позаботившись о размещении переменных в регистрах, а в C или Паскале для этого достаточно одного оператора.

С ростом объема программы становится невозможным удерживать в памяти все детали, и становится необходимым структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением степени абстракции программы.

Первым шагом к повышению абстракции является использование функций, позволяющее после написания и отладки функции отвлечься от деталей ее реализации, поскольку для вызова функции требуется знать только ее интерфейс. Если глобальные переменные не используются, интерфейс полностью определяется заголовком функции.

Следующий шаг — описание собственных типов данных, позволяющих структурировать и группировать информацию, представляя ее в более естественном виде. Например, можно представить с помощью одной структуры все разнородные сведения, относящиеся к одному виду товара на складе. Для работы с собственными типами данных требуются специальные функции. Естественно сгруппировать их с описанием этих типов данных в одном месте программы, а также по возможности отделить от ее остальных частей. При этом для использования этих типов и функций не требуется полного знания того, как именно они написаны — необходимы только описания интерфейсов. Объединение в модули описаний типов данных и функций, предназначенных для работы с ними, со скрытием от пользователя модуля несущественных деталей, является дальнейшим развитием структуризации программы.

Все три описанных выше ***метода повышения абстракции*** преследуют цель упростить структуру программы, то есть представить ее в виде меньшего количества более крупных блоков и минимизировать связи между ними. Это позволяет управлять большим объемом информации и, следовательно, успешно отлаживать более сложные программы.

Введение понятия *класса* является естественным развитием идей *модульности*. В классе структуры данных и функции их обработки объединяются. Класс используется только через его *интерфейс* — детали реализации для пользователя класса несущественны. Идея классов отражает строение объектов реального мира — ведь каждый предмет или процесс обладает набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы часто предназначены для моделирования предметов, процессов и явлений реального мира, поэтому в языке программирования удобно иметь адекватный инструмент для представления моделей.

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы. Тип задает внутреннее представление данных в памяти компьютера, множество значений, которое могут принимать величины этого типа, а также операции и функции, применяемые к этим величинам. Все это можно задать и в классе.

Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за *интерфейсом* (ведь и в реальном мире можно, например, управлять автомобилем, не имея представления о принципе внутреннего сгорания и устройстве двигателя, а пользоваться телефоном — не зная, «как идет сигнал, принципов связи и кто клал кабель»). Интерфейсом класса являются заголовки его методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Идея классов является основой *объектно-ориентированного программирования (ООП).* Основные принципы ООП были разработаны еще в языках Simula-673 и Smalltalk, но в то вре-

мя не получили широкого применения из-за трудностей освоения и низкой эффективности реализации. В C++ эти концепции реализованы эффективно, красиво и непротиворечиво, что и явилось основой успешного распространения этого языка и внедрения подобных средств в другие языки программирования.

**ООП** — это не просто набор новых средств, добавленных в язык (на C++ можно успешно писать и без использования ООП, и наоборот, возможно написать объектную по сути программу на языке, не содержащим специальных средств поддержки объектов). ООП часто называют *новой парадигмой программирования*. Красивый термин «*парадигма*» означает набор теорий, стандартов и методов, которые совместно представляют собой способ организации знаний — иными словами, способ видения мира. В программировании этот термин используется для определения модели вычислений, то есть способа структурирования информации, организации вычислений и данных. *Объектно-ориентированная программа* строится в терминах объектов и их взаимосвязей.

Выбор степени абстракции определяется типом задачи, которую требуется решить. Не имеет смысла использовать сложные технологии для решения простых задач, а попытка «врукопашную» справиться со сложными проблемами обречена на провал. С другой стороны, сложные технологии требуют больших затрат времени на их освоение.

Например, если требуется напечатать письмо, для этой цели подойдет простейший текстовый редактор, имеющий минимум возможностей, которым можно за 10 минут обучить даже собственную бабушку; подготовка статьи с формулами потребует освоения более сложного текстового процессора типа *Microsoft Word*, а для создания рекламной брошюры с иллюстрациями лучше всего подойдет один из издательских пакетов, для овладения которым потребуется не одна неделя. Так и в программировании: идеи ООП не очень просты для понимания и, в особенности, для практического использования (их неграмотное применение приносит гораздо больше вреда, чем пользы), а освоение существующих стандартных библиотек требует времени и достаточно высокого уровня первоначальной подготовки.

Конкретные величины типа данных «класс» называются экземплярами класса, или *объектами*. Объекты взаимодействуют между собой, посылая и получая сообщения. *Сообщение* — это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая «*событийно-управляемая модель*», когда данные активны и управляют вызовом того или иного фрагмента программного кода.

ПРИМЕЧАНИЕ. Примером реализации событийно-управляемой модели может служить любая программа, управляемая с помощью меню, После запуска такая программа пассивно ожидает действий пользователя и должна уметь правильно отреагировать на любое из них. Событийная модель является противоположностью традиционной (директивной), когда код управляет данными: программа после старта предлагает пользователю выполнить некоторые действия (ввести данные, выбрать режим) в соответствии с жестко заданным алгоритмом.

Основными свойствами ООП являются *инкапсуляция*, *наследование* и *полиморфизм*. Ниже кратко поясняется их смысл, а полное представление о них можно получить после изучения этой части книги.

Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется ***инкапсуляцией*** (*encapsulation*). Эта идея не нова и применялась в структурном и модульном программировании, а в ООП получила свое логическое завершение. Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и для написания программы информация о них не требуется. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним (например, при необходимости сменить способ хранения данных с массива на стек). Простота модификации, как уже неоднократно отмечалось, является очень важным критерием качества программы.

Инкапсуляция позволяет использовать класс в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки классов для применения во многих программах.

***Наследование*** — это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы, Выделение общих черт различных классов в один класс-предок является мощным механизмом абстракции — ведь и любая наука начинается с абстрагирования и классификации, которые помогают справиться со сложностью рассматриваемой предметной области.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях. В C++ каждый класс может иметь сколько угодно потомков и предков. Иногда предки называются надклассами или суперклассами, а потомки — подклассами или субклассами.

Третьим китом, на котором стоит ООП, является *полиморфизм* - возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

Понятие полиморфизма используется в C++ весьма широко. Простым примером полиморфизма может служить рассмотренная в первой части книги перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам. Другой пример — использование шаблонов функций (в дальнейшем мы рассмотрим и *шаблоны классов*), когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Благодаря тому, что программа представляется в терминах поведения объектов, при программировании используются понятия, более близкие к предметной области, следовательно, программа легче читается и понимается. Это является большим преимуществом ООП. Однако проектирование объектно-ориентированной программы представляет собой весьма сложную задачу, поскольку в процесс добавляется еще один важный этап — разработка иерархии классов.

Плохо спроектированная иерархия приводит к созданию сложных и запутанных программ. Другим препятствием к применению ООП является большой объем информации, которую требуется освоить, и ее значительная сложность. Важно до начала проектирования правильно определить, требуется ли вообще применять объектно-ориентированный подход. Если в иерархии классов нет необходимости, то, как правило, достаточно ограничиться модульной технологией, при этом можно успешно использовать классы как стандартной библиотеки, так и собственной разработки. Естественно, для применения стандартных классов требуется сначала изучить необходимый синтаксис и механизмы, а затем — конкретные свойства этих классов.

## 3.1 Классы

### 3.1.1 Описание класса

*Класс* является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные класса называются *полями* (но аналогии с полями структуры), а функции класса — *методами.* Поля и методы называются *элементами класса.*

Листинг 3.1 Описание класса

```
class <имя>
{
[ private: ]
      <описание скрытых элементов>
public:
      <описание доступных элементов>
};  // Описание заканчивается точкой с запятой
```

Спецификаторы доступа *private* и *public* управляют видимостью элементов класса. Элементы, описанные после служебного слова *private*, видимы только внутри класса. Этот вид доступа принят в классе но умолчанию. Интерфейс класса описывается после спецификатора *public*. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций *private* и *public*, порядок их следования значения не имеет.

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором *const*, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны с модификатором *static*.

Инициализация полей при описании не допускается.

Классы могут быть глобальными (объявленными вне любого блока) и локальными (объявленными внутри блока, например, функции или другого класса).

Ниже перечислены некоторые особенности локального класса:

- внутри локального класса можно использовать типы, статические (*static*) и внешние (*extern*) переменные, внешние функции и элементы перечислений из области, в которой он описан; запрещается использовать автоматические переменные из этой области;
- локальный класс не может иметь статических элементов;
- методы этого класса могут быть описаны только внутри класса;
- если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

В качестве примера создадим класс, моделирующий геометрические фигуры редактора геометрических чертежей или для игры Тетрис. Для этого нам необходимо задать ее длину и ширину и поведение. Пример будем схематическим для демонстрации синтаксиса и возможностей ООП.

Листинг 3.2 Описание класса *CFigure*

```
class CFigure
{
    int Length, Width;                          //длина и ширина фигуры
    char* name;                                 //Имя фигуры
public:
    CFigure(int len = 100, int wid = 10){ Length = len, Width = wid;};
    void draw(int x, int y);
    int GetLength(){return Length;};
    int GetWidth(){return Width;};
}
```

В этом классе два скрытых поля — *Length* и *Width*, получить значения которых извне можно с помощью методов *GetLength()* и *GetWidth()*. Доступ к полям с помощью методов в данном случае кажется искусственным усложнением, но надо учитывать, что полями реальных классов могут быть сложные динамические структуры, и получение значений их элементов не так тривиально. Кроме того, очень важной является возможность вносить в эти структуры изменения, не затрагивая интерфейс класса.

Все методы класса имеют непосредственный доступ к его скрытым полям, иными словами, тела функций класса входят в область видимости *private* элементов класса.

В приведенном классе содержится три определения методов и одно объявление (метод *draw*). Если тело метода определено внутри класса, он является встроенным (*inline*). Как правило, встроенными делают короткие методы. Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (::).

Листинг 3.3 Реализация метода *draw*

```
void CFigure::Draw(int x, int y)
{
      /* тело метода */
}
```

Метод можно определить как встроенный и вне класса с помощью директивы *inline* (как и для обычных функций, она носит рекомендательный характер).

Листинг 3.4 Реализация встроенного метода

```
inline int CFigure::GetLength()
{
     return ammo;
}
```

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется *конструктором* и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

### 3.1.2 Описание объектов

Конкретные переменные типа «класс» называются экземплярами класса, или объектами. Время жизни и видимость объектов зависят от вида и места их описания и подчиняются общим правилам C++.

Листинг 3.5 Описание и инициализация переменных, массивов и указателей на класс *CFigure*

```
CFigure Figure;                          // Объект класса CFigure с параметрами по умолчанию
CFigure Quadrate(100, 100);              // Объект с явной инициализацией
CFigure picture[100];                    // Массив объектов с параметрами по умолчанию
CFigure *pRectangle = new CFigure(20);   // Динамический объект
                                         //(второй параметр задается по умолчанию)
CFigure &Circle = Figure;                // Ссылка на объект
```

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель.

Листинг 3.6 Доступ к элементам класса при обращении к объекту и через указатель

```
int n = Figure.GetLength();
picture[5].draw();
cout « pRectangle->GetWidth();
```

Обратиться таким образом можно только к элементам со спецификатором *public*. Получить или изменить значения элементов со спецификатором *private* можно только через обращение к соответствующим методам.

Можно создать константный объект, значения полей которого изменять запрещается. К нему должны применяться только константные методы.

Листинг 3.7 Создание константного объекта

```
class CFigure
{
     int GetLength() const {return Length;}
};
const CFigure Pixel(0,0);          // Константный объект
cout « Pixel.GetWidth();
```

Константный метод:

- объявляется с ключевым словом *const* после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;
- может вызываться для любых (не только константных) объектов.

Рекомендуется описывать как константные те методы, которые предназначены для получения значений полей.

### 3.1.3 Указатель this

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра *this*, в котором хранится константный указатель на вызвавший функцию объект. Указатель *this* неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (*return this;*) или ссылки (*return *this;*) на вызвавший объект.

Для иллюстрации использования указателя *this* добавим в приведенный выше класс *CFigure* новый метод, возвращающий ссылку на наиболее длинный (поле *Length*) из двух геометрических фигур, один из которых вызывает метод, а другой передается ему в качестве параметра (метод нужно поместить в секцию public описания класса).

Листинг 3.8 Использование указателя *this*

```
CFigure& CFigure::GetLongly(CFigure M)
{
    if( Length > M.Length)
        return *this:
    return M;
}
...
CFigure Rect1(100,20), Rect2(200,30);
CFigure Long = Rect1.GetLongly(Rect2);
```

Указатель *this* можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода. Другой способ идентификации поля использует операцию доступа к области видимости.

Листинг 3.9 Доступ к полям класса с помощью указателя *this*

```
void CFigure::SetNewSize(int NewLength, int NewWidth)
{
    this->Length = NewLength;
    this->Width = NewWidth;
}
```

### 3.1.4 Конструкторы

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов.

- Конструктор не возвращает значение, даже типа *void*. Нельзя получить указатель на конструктор.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
- Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
- Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

- Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.
- Конструкторы не наследуются.
- Конструкторы нельзя описывать с модификаторами *const, virtual* и *static*.
- Конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (на пример, при передаче объекта из функции).
- Конструктор вызывается, если в программе встретилась какая-либо из синтаксических конструкций.

Листинг 3.10 Синтаксические конструкции вызова конструктора

```
имя_класса имя_объекта [(список параметров)];          // Список параметров не должен быть пустым
имя_класса (список параметров);                         //Создается объект без имени (список может быть пустым)
имя_класса имя_объекта = выражение;                     // Создается объект без имени и копируется
```

Листинг 3.11 Примеры создание объектов при вызове различных конструкторов

```
CFigure Figure;                                         // Объект класса CFigure с параметрами по умолчанию
CFigure Quadrate(100, 100), Rectangle(30);              // Объекты с явной инициализацией
CFigure F1 = CFigure(50,40);
CFigure picture[100];                                   // Массив объектов с параметрами по умолчанию
CFigure *pRectangle = new CFigure(20);                  // Динамический объект
```

В первом и во втором операторах создаются три объекта. Значения не указанных параметров устанавливаются по умолчанию.

В третьем операторе создается безымянный объект со значением параметра *Lenght = 1000* (значение второго параметра устанавливается по умолчанию). Выделяется память под объект X, в которую копируется безымянный объект.

В последнем операторе создается безымянный объект со значением параметра *health* - 500 (значение второго параметра устанавливается по умолчанию). Выделяется память под объект Y, в которую копируется безымянный объект. Такая форма создания объекта возможна в том случае, если для инициализации объекта допускается задать один параметр.

В качестве примера класса с несколькими конструкторами усовершенствуем описанный ранее класс *CFigure*, добавив в него поля, задающие цвет (*skin*) и имя (*name*).

Листинг 3.12 Реализация класса *CFigure* с конструкторами

```
enum Color {red, green, blue};   // Возможные значения цвета
class CFigure
{
    int Length, Width;                              //длина и ширина фигуры
    Color skin;
    char* name;
    public:
    CFigure(int len = 100, int wid = 10){ Length = len, Width = wid;name = "Figure";};
    CFigure(char* sName);
    CFigure(Color clr);
    void draw(){printf(name);printf("\n");};
    int GetLength() const {return Length;};
    int GetWidth(){return Width;};
    void SetName(char* str){name = str;};
    CFigure& GetLongly(CFigure M);
    void SetNewSize(int NewLength, int NewWidth);
};

CFigure::CFigure(int len = 100, int wid = 10)
{
```

```
    Length = len;
    Width = wid;
    name = "Figure";
}

CFigure::CFigure(Color clr)
{
    switch(clr)
    {
        case red: Length=100;Width=10;skin=red; name=0;break;
        case green: Length=100; Width=20;skin=green; name=0;break;
        case blue: Length=100; Width=40; skin = blue; name=0;break;
    }
}

CFigure::CFigure(char* sName)
{
    name = new char[strlen(sName)+1];  //К длине строки прибавляется 1 для хранения нуль-символа
    strcpy(name, sName);
    Length =100;
    Width = 20;
    skin=red;
}

//---------------------------------------------------------------
CFigure Rect1(100,20), Rect2(200,30);
```

Первый из приведенных выше конструкторов является конструктором по умолчанию, поскольку его можно вызвать без параметров. Объекты класса *CFigure* теперь можно инициализировать различными способами, требуемый конструктор будет вызван в зависимости от списка значений в скобках. При задании нескольких конструкторов следует соблюдать те же правила, что и при написании перегруженных функций — у компилятора должна быть возможность распознать нужный вариант.

Перегружать можно не только конструкторы, но и другие методы класса.

Существует еще один способ инициализации полей в конструкторе (кроме использованного в приведенной выше программе присваивания полям значений формальных параметров) — с помощью списка инициализаторов, расположенных после двоеточия между заголовком и телом конструктора:

Листинг 3.13 Инициализация полей классов с помощью списка инициалиизаторов

```
CFigure::CFigure(int len, int wid )
:   Length(len),
    Width(wid),
    name ("Figure"),
    skin(red)
{
}
```

Поля перечисляются через запятую. Для каждого поля в скобках указывается инициализирующее значение, которое может быть выражением. Без этого способа не обойтись при инициализации полей-констант, полей-ссылок и полей-объектов. В последнем случае будет вызван конструктор, соответствующий указанным в скобках параметрам.

Конструктор не может возвратить значение, чтобы сообщить об ошибке во время инициализации. Для этого можно использовать механизм обработки исключительных ситуаций.

#### 3.1.4.1 Конструктор копирования

*Конструктор копирования* — это специальный вид конструктора, получающий в качестве единственного параметра указатель па объект этого же класса:

```
T::T(const T&) { ... /* Тело конструктора */ }
```

где *T* — имя класса.

Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего:

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции .

Если программист не указал ни одного конструктора копирования, компилятор создает его автоматически. Такой конструктор выполняет поэлементное копирование полей. Если класс содержит указатели или ссылки, это, скорее всего, будет неправильным, поскольку и копия, и оригинал будут указывать на одну и ту же область памяти.

Запишем конструктор копирования для класса *CFigure*. Поскольку в нем есть поле паше, содержащее указатель на строку символов, конструктор копирования должен выделять память под новую строку и копировать в нее исходную.

Листинг 3.14 Конструктор копирования для класса *CFigure*

```
CFigure::CFigure(CFigure &M)
{
    Length = M.Length;
    Width = M.Width;
    skin = M.skin;
    if(M.name)
    {
        name = new char[strlen(M.name)+1];
        strcpy(name, M.name);
    }
}
```

Любой конструктор класса, принимающий один параметр какого-либо другого типа, называется *конструктором преобразования*, поскольку он осуществляет преобразование из типа параметра в тип этого класса.

### 3.1.5 Статические элементы класса

С помощью модификатора *static* можно *описать статические поля и методы* класса. Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

#### 3.1.5.1 Статические поля

*Статические поля* применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс. Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются. Ниже перечислены особенности статических полей.

Память под статическое иоле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне функций).

Листинг 3.15 Использование статического метода класса для подсчета количества элементов

```
class CFigure
{
    int Length, Width; //длина и ширина фигуры
    Color skin;
    char* name;
    static int count;
public:
  ……….
};

int CFigure::count = 0;                    // Определение в глобальной области
                                           // По умолчанию инициализируется нулем
// int CFigure::count = 10;                // Пример инициализации произвольным значением
```

- Статические поля доступны как через имя класса, так и через имя объекта:

```
CFigure *a, b;

_____
cout « CFigure::count « a->count « b.count;        // Будет выведено одно и то же
```

- На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как *private*, нельзя изменить с помощью операции доступа к области действия, как описано выше. Это можно сделать только с помощью статических методов.
- Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции *sizeof*.

#### 3.1.5.2 Статические методы

*Статические методы* предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель *this*. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Листинг 3.16 Реализация статического метода в классе *CFigure*

```
class CFigure
{
      int Length, Width; //длина и ширина фигуры
      Color skin;
      char* name;
      static int count;
      public:
      ……….
      static int GetCount();
};

int CFigure::GetCount()
{
      return count;
}
```

Статические методы не могут быть константными (*const*) и виртуальными (*virtual*).

### 3.1.6 Дружественные функции и классы

Иногда желательно иметь непосредственный доступ извне к скрытым полям класса, то есть расширить интерфейс класса. Для этого служат дружественные функции и дружественные классы.

#### 3.1.6.1 Дружественная функция

*Дружественные функции* применяются для доступа к скрытым полям класса и представляют собой альтернативу методам. Метод, как правило, используется для реализации свойств объекта, а в виде дружественных функций оформляются действия, не представляющие свойства класса, по концептуально входящие в его интерфейс и нуждающиеся в доступе к его скрытым полям, например, переопределенные операции вывода объектов или прорисовки объектов на поле редактора.

Ниже перечислены правила описания и особенности дружественных функций.

- Дружественная функция объявляется внутри класса, к элементам которого ей нужен доступ, с ключевым словом *friend*. В качестве параметра ей должен передаваться объект или ссылка на объект класса, поскольку указатель *this* ей не передается.
- Дружественная функция может быть обычной функцией или методом другого ранее определенного класса. На нее не распространяется действие спецификаторов доступа, место размещения ее объявления в классе безразлично.
- Одна функция может быть дружественной сразу нескольким классами.

В качестве примера ниже приведено описание функции, дружественных классу *CFigure*. Функция *Draw* является методом класса некоторого графического окна, в котором будем рисовать данные геометрические фигуры.

Листинг 3.17 Описание дружественной функции класса *CFigure*

```
class CFigure
{
    friend void Draw(CDC* dc); //CDC - в MFC класс контекста графического устройства
}
```

Использования дружественных функций нужно по возможности избегать, поскольку они нарушают принцип инкапсуляции и, таким образом, затрудняют отладку и модификацию программы.

#### 3.1.6.2 Дружественный класс

Если все методы какого-либо класса должны иметь доступ к скрытым полям другого, весь класс объявляется дружественным с помощью ключевого слова *friend*. В приведенном ниже примере класс *mistress* объявляется дружественным классу *hero*.

Листинг 3.18 Пример реализации дружественного класса

```
class hero
{
    friend class mistress;
}
class mistress
{
    void f1();
    void f2();
}
```

Функции *f1* и *f2* являются дружественными по отношению к классу *hero* (хотя и описаны без ключевого слова *friend*) и имеют доступ ко всем его полям.

Объявление friend не является спецификатором доступа и не наследуется.

### 3.1.7 Деструкторы

*Деструктор* — это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

- для локальных объектов — при выходе из блока, в котором они объявлены;
- для глобальных — как часть процедуры выхода из main:
- для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции *delete*.

Автоматический вызов деструктора объекта при выходе из области действия указателя на него не производится.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса. Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как *const* или *static*;
- не наследуется;
- может быть виртуальным.

Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на

которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Деструктор для рассматриваемого примера должен выглядеть так.

Листинг 3.19 Реализация деструктора для класса *CFigure*

```
class CFigure
{
.............
      complex* Pixel;
      int CountPixel;
public:
...........
      ~CFigure();
}

CFigure::~CFigure()
{
      if(Pixel)
            delete [] Pixel;
}
```

Деструктор можно вызвать явным образом путем указания полностью уточненного имени,

Листинг 3.20 Явный вызов деструктора

```
CFigure* pFigure = new CFigure;
delete pFigure;
```

Это может понадобиться для объектов, которым с помощью перегруженной операции *new* выделялся конкретный адрес памяти. Без необходимости явно вызывать деструктор объекта не рекомендуется.

О классах, их полях и методам рекомендуется прочитать в литературе:

5. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 269 - 309.

6. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 281 - 322.

### 3.1.8 Перегрузка операций

*C++* позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. Это дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в *C++*, за исключением:

```
.   .*   ?:   ::   #   ##   sizeof
```

Перегрузка операций осуществляется с помощью методов специального вида (функций-операций) и подчиняется следующим правилам:

- при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
- для стандартных типов данных переопределять операции нельзя;
- функции-операции не могут иметь аргументов по умолчанию;
- функции-операции наследуются (за исключением =);
- функции-операции не могут определяться как *static*.

Функцию-операцию можно определить тремя способами: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних слу-

чаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.

Функция-операция содержит ключевое слово *operator*, за которым следует знак переопределяемой операции.

Листинг 3.21 Описание перегрузки операции

```
тип operator операция ( список параметров) { тело функции }
```

### 3.1.8.1 Перегрузка унарных операций

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например.

Листинг 3.22 Перегрузка операции инкремента для класса *CFigure*

```
class CFigure
{
      CFigure& operator++();
};
CFigure& CFigure::operator++()
{
      ++Length;
      ++Width
      return *this;
}
CFigure* M = new CFigure("Rect");
++(*M);
M->Draw();
```

Если функция определяется вне класса, она должна иметь один параметр типа класса.

Листинг 3.23 Перегрузка операции декремента как дружеской к классу *CFigure*

```
class CFigure
{
      friend CFigure& operator--(CFigure& Figure);
};
CFigure& operator--(CFigure& Figure)
{
      Figure.Length--;
      Figure.Width--;
      return Figure;
}
```

Если не описывать функцию внутри класса как дружественную, нужно учитывать доступность изменяемых полей. В данном случае поля *Length* и *Width* недоступны извне, так как описано со спецификатором private, поэтому для его изменения требуется использование соответствующего метода. Введем в описание класса *CFigure* метод *ChangeSize*, позволяющий изменить значение полей *Length* и *Width*.

Листинг 3.24 Реализация метода *ShangeSize* класса *CFigure*

```
void CFigure::ChangeSize(int Width, int Length)
{
      this->Length=Length;
      this->Width=Width;
}
```

Тогда можно перегрузить операцию инкремента с помощью обычной функции, описанной вне класса.

Листинг 3.25 Реализация функции инкремента вне класса *CFigure*

```
CFigure& operator++(CFigure &M)
{
```

```
        int L = M.GetLength();
        int W = M.GetWidth();
        L++;
        W++;
        M.ChangeSize(W,L);
        return M;
}
```

Операции постфиксного инкремента и декремента должны иметь первый параметр типа int. Он используется только для того, чтобы отличить их от префиксной формы.

Листинг 3.26 Реализация операции постфиксного инкремента класса *CFigure*

```
CFigure& CFigure::operator++(int)
{
        Length++;
        Width++;
        return *this;
}
Rect1.SetNewSize(400,30);
CFigure NewLong = Rect1.GetLongly(Rect2);
NewLong.draw();
NewLong++;
NewLong.draw();
```

### 3.1.8.2 Перегрузка бинарных операций

Бинарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

Листинг 3.27 Реализация операции сравнения для объектов класса *CFigure*

```
class CFigure
{
……...
public:
        bool operator>(const CFigure &M)
}

bool CFigure::operator>(const CFigure &M)
{
        if(Width>M.Width)
                return true;
        else
                return false;
}
```

### 3.1.8.3 Перегрузка операции присваивания

Операция присваивания определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля, память под которые выделяется динамически, необходимо определить собственную операцию присваивания. Чтобы сохранить семантику присваивания, операция-функция должна возвращать ссылку на объект, для которого она вызвана, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект.

Листинг 3.28 Перегрузка операции присваивания для класса *CFigure*

```
const CFigure& CFigure::operator=(CFigure &M)
{
        //Проверка на самоприсваивание
        if(&M==this)
                return *this;
        if(name)
                delete [] name;
        if(M.name)
        {
```

```
            name = new char[strlen(M.name)+1];
            strcpy(name, M.name);
        }
        Length=M.Length;
        Width=M.Width;
        skin=M.skin;
        return *this;
    }
```

Возврат из функции указателя на объект делает возможной цепочку операций присваивания.

Листинг 3.29 Применение операции присваивания для объектов класса *CFigure*

```
CFigure A;
A.SetName("Rect");
CFigure B = A;
B.draw();
```

Операцию присваивания можно определять только как метод класса. Она не наследуется.

О правилах перегрузки операций рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 309 - 349.
2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 322 - 335.

## 3.2 Наследование

*Механизм наследования классов* позволяет строить *иерархии*, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

### 3.2.1 Ключи доступа

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью ключей доступа *private, protected* и *public*:

```
class имя : [private | protected | public] базовый_класс { тело класса };
```

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом.

Листинг 3.30 Пример реализации иерархии классов

```
class A {...};
class B {...};
class C {...}:
class D: A,  protected B,  public C { ... };
```

По умолчанию для классов используется ключ доступа *private*, а для структур — *public*.

До сих пор мы рассматривали только применяемые к элементам класса спецификаторы доступа private и public. Для любого элемента класса может также использоваться специфика-

тор protected, который для одиночных классов, не входящих в иерархию, равносилен private. Разница между ними проявляется при наследовании, что можно видеть из приведенной таблицы.

Таблица 6.1. Ключи доступа к полям родительского класса

| Ключ доступа | Спецификатор в базовом классе | Доступ в производном классе |
|---|---|---|
| private | private<br>protected<br>public | нет<br>private<br>private |
| protected | private<br>protected<br>public | нет<br>protected<br>protected |
| public | private<br>protected<br>public | нет<br>protected<br>public |

Как видно из таблицы, *private* элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.

Элементы *protected* при наследовании с ключом *private* становятся в производном классе *private*, в остальных случаях права доступа к ним не изменяются.

Доступ к элементам *public* при наследовании становится соответствующим ключу доступа.

Если базовый класс наследуется с ключом private, можно выборочно сделать некоторые его элементы доступными в производном классе, объявив их в секции *public* производного класса с помощью операции доступа к области видимости.

Листинг 3.31 Пример явного изменения видимости для методов базового класса

```
class Base
{
      public: void f();
};
class Derived : private Base
{
      public: Base: :void f();
}
```

### 3.2.2 Простое наследование

Простым называется наследование, при котором производный класс имеет одного родителя. Для различных методов класса существуют разные правила наследования — например, конструкторы и операция присваивания в производном классе не наследуются, а деструкторы наследуются. Рассмотрим наследование классов и проблемы, возникающие при этом, на примере.

Создадим производный от класса CFigure класс CD3Figure, добавив информацию о толщине объекта и заданием его прозрачности:

Листинг 3.32 Класс *CD3Figure*, производный от класса *CFigure*

```
enum Color{red, blue, green};


class CFigure
{
      int Length, Width; //длина и ширина фигуры
```

```
        Color skin;
        char* name;
        complex *Pixel;
        int CountPixel;
        static int count;
        public:
        CFigure(int len = 100, int wid = 10);
        CFigure(char* sName);
        CFigure(Color clr);
        CFigure(CFigure& M);
        ~CFigure();
        void draw(){printf(name);printf("\n");};
        int GetLength() const {return Length;};
        int GetWidth(){return Width;};
        void SetName(char* str){name = str;};
        CFigure& GetLongly(CFigure M);
        void SetNewSize(int NewLength, int NewWidth);
        static int GetCount();
        CFigure& operator++();
        CFigure& operator++(int);
        friend CFigure& operator--(CFigure& Figure);
        void ChangeSize(int Width, int Length);
        bool operator>(const CFigure &M);
        const CFigure& operator=(CFigure &M);
};

class CD3Figure : public Figure
{
        int Heigth;
        bool Transparency;
        public:
        CD3Figure(int len=100, int wid =10, int heigth = 10) : CFigure(len, wid) { Heigth =height;};
        CD3Figure(color sk) : CFigure(sk) {Height = 10;};
        CD3Figure(char* name) : CFigure(name){};
        CD3Figure& operator=(CD3Figure& M);
        void draw(int i){printf("3D+"name);printf("\n");}
}

CD3Figure& CD3Figure::operator =(CD3Figure& M)
{
        if(&M==this)
                return *this;
        Heigth = M.Heigth;
        CFigure::operator =(M);
        Transparency = M.Transparency;
        return *this;
}
```

В классе *CD3Figure* введено поля *Heigth* и *Transparent* и метод *think*, определены собственные конструкторы и операция присваивания, а также переопределен метод отрисовки *draw*. Все поля класса *CFigure*, операции (кроме присваивания) и методы наследуются в классе *CD3Figure*, а деструктор формируется по умолчанию.

Рассмотрим правила наследования различных методов.

- Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.
- Если в конструкторе производного класса явный вызов конструктора базово го класса отсутствует, автоматически вызывается конструктор базового класса но умолчанию (то есть тот, который можно вызвать без параметров). Это использовано в первом из конструкторов класса *CD3Figure*.
- Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.
- В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации (это продемонстрировано в трех последних конструкторах).

Не наследуется и операция присваивания, поэтому ее также требуется явно определить в классе *CD3Figure*. Обратите внимание на запись функции-операции: в ее теле применен явный вызов функции-операции присваивания из базового класса. Чтобы лучше представить себе синтаксис вызова, ключевое слово *operator* вместе со знаком операции можно интерпретировать как имя функции-операции.

Вызов функций базового класса предпочтительнее копирования фрагментов кода из функций базового класса в функции производного. Кроме сокращения объема кода, этим достигается упрощение модификации программы: изменения требуется вносить только в одну точку программы, что сокращает количество возможных ошибок.

Ниже перечислены правила наследования деструкторов.

- Деструкторы не наследуются, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.
- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.
- Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Поля, унаследованные из класса *CFigure*, недоступны функциям производного класса, поскольку они определены в базовом классе как *private*. Если функциям, определенным в *CD3Figure*, требуется работать с этими полями, можно либо описать их в базовом классе как *protected*, либо обращаться к ним с помощью функций из *CFigure*, либо явно переопределить их в *CD3Figure*.

Рассматривая наследование методов, обратите внимание на то, что в классе *CD3Figure* описан метод *draw*, переопределяющий метод с тем же именем в классе *CFigure* (поскольку отрисовка различных персонажей, естественно, выполняется по-разному). Таким образом, производный класс может не только дополнять, но и корректировать поведение базового класса. Доступ к переопределенному методу базового класса для производного класса выполняется через имя, уточненное с помощью операции доступа к области видимости (::).

### 3.2.3 Виртуальные методы

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса (при открытом наследовании).

Листинг 3.33 Пример работы с указателями базового класса

```
// Описывается указатель на базовый класс:
CFigure *p;
// Указатель ссылается на объект производного класса:
p = new CD3Figure;
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается, поэтому при выполнении оператора, например.

Листинг 3.34 Вызов метода производного класса

```
pD3Figure->draw();
```

будет вызван метод класса *CFigure*, а не класса *CD3Figure*, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием.

Чтобы вызвать метод класса *CD3Figure*, можно использовать явное преобразование типа указателя.

Это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример — связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

Наряду с ранним связыванием, в C++ реализован механизм позднего связывания, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов и рассмотрен в следующем разделе.

Для определения виртуального метода используется спецификатор *virtual*.

Листинг 3.35 Определение виртуального метода

```
virtual void draw(int x. int y, int scale, int position);
```

Рассмотрим правила описания и использования виртуальных методов.

- Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров — обычным.
- Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.
- Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.
- Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.
- Если в классе вводится описание виртуального метода, он должен быть опре делен хотя бы как чисто виртуальный.

Чисто виртуальный метод содержит признак = 0 вместо тела, например:

```
virtual void f(int) = 0;
```

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Если определить метод *draw* в классе *CFigure* как виртуальный, решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель.

Листинг 3.36 Вызов определенного метода

```
CFigure* pD3Figure, pFigure;
pD3Figure = new CD3Figure("D3Figure");          // Создается объект класса CD3Figure
pFigure  = new CFigure(“2DFigure”);             // Создается объект класса CFigure
pFigure->draw();                                //Вызывается метод CFigure::draw
pD3Figure->draw();                              //Вызывается метод CD3Figure::draw
pD3Figure->CFigure::draw();                     //обход механизма виртуальный методов
```

Если объект класса *CD3Figure* будет вызывать метод *draw* не непосредственно, а косвенно (то есть из другого метода, определенного в классе *CFigure*), будет вызван метод *draw* класса *CD3Figure*.

Итак, *виртуальным* называется метод, ссылка на который разрешается на этапе выполнения программы (перевод красивого английского слова *virtual* — в данном значении всегонавсего «фактический», то есть ссылка разрешается по факту вызова).

### 3.2.3.1 Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (*vtbi*), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в *vtbl* одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное поле ссылки на *vtbl*, называемое *vptr*. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции). На этапе компиляции ссылки на виртуальные методы заменяются на обращения к *vtbl* через *vptr* объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции *delete* размер объекта, имеющий тип *size_t*. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах. Методы, которые во всей иерархии останутся неизменными или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла. С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы (особенно при проектировании библиотек классов), а объявление метода виртуальным обеспечивает гибкость и возможность расширения.

Для пояснения последнего тезиса представим себе, что вызов метода *draw* осуществляется из метода перемещения объекта. Если текст метода перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех объектов одинаков, а для отрисовки вызывается конкретный метод), переопределять этот метод в производных классах нет необходимости, и он может быть описан как невиртуальный. Если метод *draw* виртуальный, метод перемещения сможет без перекомпиляции работать с объектами любых производных классов — даже тех, о которых при его написании ничего известно не было.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*. В данном случае *полиморфизм* состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

### 3.2.3.2 Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным. Абстрактные классы* предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов — объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении.

При определении абстрактного класса необходимо иметь в виду следующее:

- абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
- допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;

- если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого типа в пределах одной иерархии.

### 3.2.4 Отличия структур и объединений от классов

Структуры (*struct*) и объединения (*union*) представляют собой частные случаи классов.

Структуры отличаются от классов тем, что доступ к элементам, а также базовый класс при наследовании по умолчанию считаются *public*. Структуры предпочтительнее использовать для классов, все элементы которых доступны.

Отличия объединений от классов перечислены ниже:

- доступ в объединениях по умолчанию public, кроме того, в них вообще нельзя явным образом использовать спецификаторы доступа;
- объединение не может участвовать в иерархии классов;
- элементами объединения не могут быть объекты, содержащие конструкторы и деструкторы;
- объединение может иметь конструктор и другие методы, только не статические;
- в анонимном объединении нельзя описывать методы.

О механизме наследования классов рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 349 - 376.

2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 336 - 374.

## 3.3 Шаблоны классов

Ранее были рассмотрены шаблоны функций, с помощью которых можно отделить алгоритм от конкретных типов данных, с которыми он работает, передавая тип в качестве параметра. Шаблоны классов предоставляют аналогичную возможность, позволяя создавать параметризованные классы.

Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра. Наиболее широкое применение шаблоны находят при создании контейнерных классов. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Стандартная библиотека *C++* содержит множество контейнерных классов для организации структур данных различного вида.

Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

### 3.3.1 Создание шаблонов классов

Рассмотрим процесс создания шаблона класса на примере. В разделе «Линейные списки» (с. 57) был описан двусвязный список и приведены алгоритмы работы с ним. Поскольку списки часто применяются для организации данных, удобно описать список в виде класса, а так как может потребоваться хранить данные различных типов, этот класс должен быть параметризованным.

Сначала рассмотрим непараметризованную версию класса «список».

Список состоит из узлов, связанных между собой с помощью указателей. Каждый узел хранит целое число, являющееся ключом списка. Опишем вспомогательный класс для представления одного узла списка.

Листинг 3.37 Класс, представляющий узел списка

```
class Node
{
    public:
    int d;                                  //Данные
    Node *next;                             //Указатель на последующий узел
    Node *prev;                             //Указатель на предыдущий узел
    Node(int dat = 0)                       //Конструктор
    {
        d = dat; next = 0; prev = 0;
    }
};
```

Поскольку этот класс будет описан внутри класса, представляющего список, поля для простоты доступа из внешнего класса сделаны доступными (*public*). Это позволяет обойтись без функций доступа и изменения полей. Назовем класс списка *List*.

Листинг 3.38 Класс списка *List*

```
class List
{
    class Node
    {
        --------
    };
    Node *pbeg, *pend;                      // Указатели на начало и конец списка
public:
    List(){pbeg = 0; pend = 0;}             //Конструктор
    ~List();                                //Деструктор
    void add(int d);                        //Добавление элемента в конец списка
    Node * find(int i);                     //Поиск элемента по ключу
    Node * insert(int key, int d);          // Вставка узла d после узла с ключом key
    bool remove(int key);                   // Удал/ение узла
    void print();                           // Печать списка в прямом направлении
    void print_back();                      // Печать списка в обратном направлении
};
```

Рассмотрим реализацию методов класса. Метод *add* выделяет память под новый объект типа Node и присоединяет его к списку, обновляя указатели на его начало и конец.

Листинг 3.39 Реализация метода *Add* класса *List*

```
void List::add(int d)
{
    Node *pv = new Node(d);                 // Выделение памяти под новый узел
    if (pbeg == 0)
        pbeg = pend = pv;                   // Первый узел списка
    else
    {                                       // Связывание нового узла с предыдущим:
        pv->prev = pend;
        pend->next = pv;
        pend = pv;
    }                                       // Обновление указателя на конец списка
}
```

При желании получить отсортированный список этот метод можно заменить на метод, аналогичный функции формирования отсортированного списка *add_sort*, приведенной в разделе «Линейные списки» на с. 61.

Метод *find* выполняет поиск узла с заданным ключом и возвращает указатель на него в случае успешного поиска и 0 в случае отсутствия такого узла в списке.

Листинг 3.40 Метод поиска узла в списке по заданному ключу

```
Node * List::find(int d )
{
    Node *pv = pbeg;
    while (pv)
    {
```

```
            if(pv->d == d)
                break;
            pv = pv->next:
        )
        return pv:
}
```

Метод *insert* вставляет в список узел после узла с ключом *key* и возвращает указатель на вставленный узел. Если такого узла в списке нет, вставка не выполняется и возвращается значение 0.

Листинг 3.41 Метод вставки узла по заданному ключу

```
Node * List::insert(int key, int d)
{
    if(Node *pkey = find(key))
    {
        // Поиск узла с ключом key
        // Выделение памяти под новый узел и его инициализация:
        Node *pv = new Node(d);
        // Установление связи нового узла с последующим:
        pv->next = pkey->next;
        // Установление связи нового узла с предыдущим:
        pv->prev = pkey;
        // Установление сзязи предыдущего узла с новым:
        pkey->next = pv;
        // Установление связи последующего узла с новым:
        if( pkey != pend)
            (pv->next)->prev = pv;      // Обновление указателя на конец списка,
                                        // если узел вставляется в конец:
        else
            pend = pv;
        return pv;
    }
    return 0;
}
```

Метод *remove* удаляет узел с заданным ключом из списка и возвращает значение *true* в случае успешного удаления и *false*, если узел с таким ключом в списке не найден:

Листинг 3.42 Метод удаления узла по заданному ключу

```
bool List::remove(int key)
{
    if(Node *pkey = find(key))
    {                                   // Удаление из начала списка
        if (pkey == pbeg)
        {
            // Удаление из конца списка
            // Удаление из середины списка
            pbeg =  pbeg->next;
            pbeg->prev =0:
        }
        else
            if (pkey == pend)
            {
                    pend = pend->prev;
                    pend->next =0;
            }
            else
            {
                    (pkey->prev)->next = pkey->next;
                    (pkey->next)->prev = pkey->prev;
            }
            delete pkey:
            return true;
    }
    return false;
}
```

Методы печати списка в прямом и обратном направлении поэлементно просматривают список, переходя по соответствующим ссылкам.

Листинг 3.43 Методы печати списка в прямом и обратном порядке

```
void List::print()
{
      Node *pv = pbeg;
      cout « endl « "list:   ";
      while (pv)
      {
            cout « pv->d « '   ';
            pv = pv->next;
      }
      cout « endl:
}

void List::print_back()
{
      Node *pv = pend;
      cout « endl « " list back:  ";
      while (pv)
      {
            cout« pv->d « '  ';
            pv « pv->prev;
      }
      cout « endl;
}
```

Деструктор списка освобождает память из-под всех его элементов.

Листинг 3.44 Деструктор класса списка

```
List::~List()
{
      if (pbeg != 0)
      {
            Node *pv = pbeg;
            while (pv)
            {
                  pv = pv->next;
                  delete pbeg;
                  pbeg = pv;
            }
      }
}
```

Ниже приведен пример программы, использующей класс *List*. Она формирует список из 5 чисел, выводит его на экран, добавляет число в список, удаляет число из списка и снова выводит его на экран.

Листинг 3.45 Программа, демонстрирующая работу с классом *List*

```
int main()
{
      List L;
      for (int 1 = 2; i<6: i++)
            L.add(i);
      L.print();
      L.pnnt_back();
      L.insert(2,  200):
      if (!L.remove(5))
            cout « "not found";
      L.print():
      L.print_back();
}
```

Класс *List* предназначен для хранения целых чисел. Чтобы хранить в нем данные любого типа, требуется описать этот класс как шаблон и передать тип в качестве параметра.

Листинг 3.46 Синтаксис описания шаблона

```
template <описание_параметров_шаблона> определение_класса;
```

Параметры шаблона перечисляются через запятую. В качестве параметров могут использоваться типы, шаблоны и переменные.

Типы могут быть как стандартными, так и определенными пользователем. Для их описания используется ключевое слово *class*. Внутри шаблона параметр типа может применяться в любом месте, где допустимо использовать спецификацию типа.

Листинг 3.47 Шаблон класса списка

```
template <class Data>
class List
{
      class Node
      {
      public:
            Data d;
            Node *next;
            Node *prev;
            Node(Data dat = 0){d = dat; next = 0; prev = 0;}
      }
      …
}
```

Класс *Data* можно рассматривать как формальный параметр, на место которого при компиляции будет подставлен конкретный тип данных.

Для любых параметров шаблона могут быть заданы значения по умолчанию.

Листинг 3.48 Задание параметров шаблона по умолчанию

```
template<class T> class myarray { /* … */ };
template<class K, class V. template<class T> class C = myarray>
class Map
{
      C<K> key;
      C<V> value;
};
```

Область действия параметра шаблона — от точки описания до конца шаблона, поэтому параметр можно использовать при описании следующих за ним, например:

```
template<class T, T* p, class U - T> class X { /* … */ };
```

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

```
template <описание__паранетров_шаблона>
      возвр_тип имя_класса <параметры_шаблона>::
            имя_функции (список_параметров функции)
```

Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса, при этом имена параметров могут не совпадать. Проще рассмотреть синтаксис описания методов шаблона на примере.

Листинг 3.49 Метод шаблона класса

```
template <class Data> void List<Data>::print()
{
      /* тело функции */
}
```

Здесь *<c1ass Data>*— описание параметра шаблона, *void* — тип возвращаемого функцией значения, *List* — имя класса, *<Data>* — параметр шаблона, *print* — имя функции без параметров.

В случае нескольких параметров порядок их следования в *описании_параметров* и *параметрах_шаблона* должен быть один и тот же.

Листинг 3.50 Порядок следования параметров

```
template<class T1,  class T2> struct A
{
      void fl();
};
template<class T2. class T1>
void A<T2, Tl>::fl(){ … }
```

Ниже перечислены правила описания шаблонов.

- Локальные классы не могут содержать шаблоны в качестве своих элементов.

- Шаблоны методов не могут быть виртуальными.
- Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
- Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.
- Внутри шаблона нельзя определять *friend*-шаблоны.

В качестве примера шаблона рассмотрим полное описание параметризованного класса двусвязного списка *List.*

Листинг 3.51 Полное описание параметризованного класса двусвязного списка *List*

```
template <class Data>
class List
{
      class Node
      {
            public: Data d;
            Node *next, *prev;
            Node(Data dat = 0){d = dat; next = 0; prev = 0;}
      }:
      Node *pbeg. *pend;
public:
      List(){pbeg = 0;  pend = 0;}
      ~List();
      void add(Data d);
      Node* find(Data i);
      Node* insert(Data key. Data d);
      bool remove(Data key);
      void print();
      void print_back();
};

template <class Data>
List <Data>::~List()
{
      If(pbeg !=0)
      {
            Node *pv = pbeg;
            while (pv)
            {
                  pv = pv->next;
                  pbeg = pv;
            }
      }
}

template <class Data>
void List <Data>::print()
{
       Node *pv = pbeg;
      cout « endl « "list: “;
      while (pv)
      {
            cout « pv->d « ' ‘;
            pv = pv->next;
      }
      cout « endl;
      }

template <class Data>
void List <Data>::print_back()
{
      Node *pv = pend;
      cout « endl « " list back;
      while (pv)
      {
            cout « pv->d « ' ';
            pv = pv->prev;
      }
      cout « endl;
}

template <class Data>
void List <Data>::add(Data d)
{
```

```
        Node *pv = new Node(d);
        if (pbeg == 0)
            pbeg = pend = pv;
        else
        {
            pv->prev = pend;
            pend->next = pv;
            pend = pv;
        }
}

template <class Data>
Node * List <Data>::find(Data d)
{
        Node *pv = pbeg;
        while (pv)
        {
            if(pv->d == d)
                break;
            pv = pv->next:
        }
        return pv;
}

template <class Data>
Node * List <Data>::insert(Data key. Data d)
{
        If(Node *pkey = find(key))
        {
            Node *pv = new Node(d);
            pv->next - pkey->next;
            pv->prev = pkey;
            pkey->next = pv;
            if( pkey != pend)
                (pv->next)->prev = pv;
            else
                pend = pv;
            return pv;
        }
        return 0;
}



template <class Data>
bool List <Data>::remove(Data key)
{
        if(Node *pkey = find(key))
        {
            if (pkey == pbeg)
            {
                pbeg = pbeg->next;
                pbeg->prev = 0;
            }
            else
            if (pkey == pend)
            {
                pend = pend->prev;
                pend->next =0;
            }
            else
            {
                (pkey->prev)->next == pkey->next;
                (pkey->next)->prev = pkey->prev;
            }
            delete pkey;
            return true;
        }
        return false;
}
```

Если требуется использовать шаблон *List* для хранения данных не встроенного, а определенного пользователем типа, в описание этого типа необходимо добавить перегрузку операции вывода в поток и сравнения на равенство, а если для его полей используется динамическое выделение памяти, то и операцию присваивания.

При определении синтаксиса шаблона было сказано, что в него, кроме типов и шаблонов, могут передаваться переменные. Они могут быть целого или перечисляемого типа, а также указателями или ссылками на объект или функцию. В теле шаблона они могут применяться в любом месте, где допустимо использовать константное выражение. В качестве примера создадим шаблон класса, содержащего блок памяти определенной длины и типа.

Листинг 3.52 Класс, содержащий блок памяти определенной длины и типа

```
template <class Type,  int kol> class Block
{
public:
      Block()
      {
            p = new Type [kol];
      }
      ~Block()
      {
            delete [] p;
      }
      operator Type *();
protected:
      Type * p;
};

template <class Type, int kol>
Block <Type,'ko1>:: operator Type *()
{
      return p;
}
```

После создания и отладки шаблоны классов удобно помещать в заголовочные файлы.

О шаблоне классов рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 377 - 407.
2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 375 - 379.

## 3.4 Обработка исключительных ситуаций

*Исключительная ситуация,* или *исключение* — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры. Например, это деление на ноль или обращение по несуществующему адресу памяти. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. *C++* дает программисту возможность восстанавливать программу и продолжать ее выполнение.

Исключения *C++* не поддерживают обработку асинхронных событий, таких, как ошибки оборудования или обработку прерываний, например, нажатие клавиш *Ctrl+C*. Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главной причиной является то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается. Это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение.

В принципе, ничто не мешает рассматривать в качестве исключений не только ошибки, но и нормальные ситуации, возникающие при обработке данных, но это не имеет преимуществ перед другими решениями и не улучшает структуру и читаемость программы.

### 3.4.1 Общий механизм обработки исключений

Место, в котором может произойти ошибка, должно входить в контролируемый блок — составной оператор, перед которым записано ключевое слово *try*. Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

- Обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение. Для этого используется ключевое слово *throw* с параметром, определяющим вид исключения. Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику.
- Отыскивается соответствующий обработчик исключения и ему передается управление.
- Если обработчик исключения не найден, вызывается стандартная функция *terminate*, которая вызывает функцию *abort*, аварийно завершающую текущий процесс. Можно установить собственную функцию завершения процесса.

Ранее говорилось о том, что при вызове каждой функции в стеке создается область памяти для хранения локальных переменных и адреса возврата в вызывающую функцию. Термин стек вызовов обозначает последовательность вызванных, но еще не завершившихся функций. *Раскручиванием стека* называется процесс освобождения памяти из-под локальных переменных и возврата управления вызывающей функции. Когда функция завершается, происходит естественное раскручивание стека. Тот же самый механизм используется и при обработке исключений. Поэтому после того, как исключение было зафиксировано, исполнение не может быть продолжено с точки генерации исключения. Подробнее этот механизм рассматривается в следующем разделе.

### 3.4.2 Синтаксис исключений

Ключевое слово try служит для обозначения контролируемого блока — кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки:

Листинг 3.53 Общий вид блока исключений

```
try
{

}
```

Все функции, прямо или косвенно вызываемые из *try*-блока, также считаются ему принадлежащими.

*Генерация (порождение)* исключения происходит по ключевому слову *throw*, которое употребляется либо с параметром, либо без него.

Листинг 3.54 Порождение исключения

```
throw [ выражение ];
```

Тип выражения, стоящего после *throw*, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не непосредственно в try-блоке, а в функциях, прямо или косвенно в него вложенных.

Не всегда исключение, возникшее во внутреннем блоке, может быть сразу правильно обработано. В этом случае используются вложенные контролируемые блоки, и исключение передается на более высокий уровень с помощью ключевого слова *throw* без параметров.

Обработчики исключений начинаются с ключевого слова *catch*, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за *try*-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабаты-

ваемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существует три формы записи.

Листинг 3.55 Формы записи обработчиков исключений

```
catch(тип имя)
{
    /* тело обработчика */
}
catch(тип)
{
    /* тело обработчика */
}
catch(...)
{
    /* тело обработчика */
}
```

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении. Вторая форма не предполагает использования информации об исключении, играет роль только его тип. Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.

Листинг 3.56 Пример реализации обработчиков исключений

```
catch(nt i)
{
    // Обработка исключений типа int
}
catch(const char *)
{
    // Обработка исключений типа const char*
}
catch(overflow)
{
    // Обработка исключений класса Overflow
}
catch(...)
{
    // Обработка всех необслуженных исключений
}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в *try*-блоке не было сгенерировано.

### 3.4.3 Перехват исключений

Когда с помощью *throw* генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- создают копию параметра *throw* в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;
- в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;
- передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

При раскручивании стека все обработчики на каждом уровне просматриваются последовательно, от внутреннего блока к внешнему, пока не будет найден подходящий обработчик. Обработчик считается найденным, если тип объекта, указанного после *throw*:

- тот же, что и указанный в параметре *catch* (параметр может быть записан в форме *T, const T, T&* или *const T&*, где *T* — тип исключения);

- является производным от указанного в параметре *catch* (если наследование производилось с ключом доступа *public*);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре *catch*.

Из вышеизложенного следует, что обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа *void* автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Листинг 3.57 Пример генерации и обработки исключительных ситуаций

```
#include <fstream,h>
class Hello
{
      // Класс, информирующий о своем создании и уничтожении
      public:
      Hello(){cout « "Hello!" « endl:}
      ~Hello(){cout « "Bye!" « endl;}
};
void fl()
{
      ifstream ifs("'WINVALIDWFILEWNAME");                  // Открываем файл
      if (!ifs)
      {
            cout « "Генерируем исключение" « endl;
            throw "Ошибка при открытии файла";
      }
}
void f2()
{
      Hello H;                                              // Создаем локальный объект
      f1();                                                 // Вызываем функцию, генерирующую исключение
}

int main()
{
      try
      {
            cout « "Входим в try-блок" « end!;
            f2();
            cout « "Выходим из try-блока" « endl:
      }
      catch(int i)
      {
            cout « "Вызван обработчик int, исключение - " « i « endl; return -1;
      }
      catch(const char * p)
      {
            cout « "Вызван обработчик const char*, исключение - " « p « end!;
            return -1;
      }
      catcht(...)
      {
            cout « "Вызван обработчик всех исключений" « end!;
            return -1;
      }
      return 0; // Все обошлось благополучно
}
```

Листинг 3.58 Результаты выполнения программы листинга 3.57

```
Входим а try-блок
Hello!
Генерируем исключение
bye!
Вызван обработчик const char *, исключение - Ошибка при открытии файла
```

Обратите внимание, что после порождения исключения был вызван деструктор локального объекта, хотя управление из функции Я было передано обработчику, находящемуся в функции main. Сообщение «Выходим из *try*-блока» не было выведено. Для работы с файлом в программе использовались потоки.

Таким образом, механизм исключений позволяет корректно уничтожать объекты при возникновении ошибочных ситуаций. Поэтому выделение и освобождение ресурсов полезно оформлять в виде классов, конструктор которых выделяет ресурс, а деструктор освобождает. В качестве примера можно привести класс для работы с файлом. Конструктор класса открывает файл, а деструктор — закрывает. В этом случае есть гарантия, что при возникновении ошибки файл будет корректно закрыт, и информация не будет утеряна.

Как уже упоминалось, исключение может быть как стандартного, так и определенного пользователем типа. При этом нет необходимости определять этот тип глобально — достаточно, чтобы он был известен в точке порождения исключения и в точке его обработки. Класс для представления исключения можно описать внутри класса, при работе с которым оно может возникать. Конструктор копирования этого класса должен быть объявлен как *public*, поскольку иначе будет невозможно создать копию объекта при генерации исключения (конструктор копирования, создаваемый по умолчанию, имеет спецификатор *public*).

### 3.4.4 Список исключений функции

В заголовке функции можно задать список исключений, которые она может прямо или косвенно порождать. Поскольку заголовок является интерфейсом функции, указание в нем списка исключений дает пользователям функции необходимую информацию для ее использо-

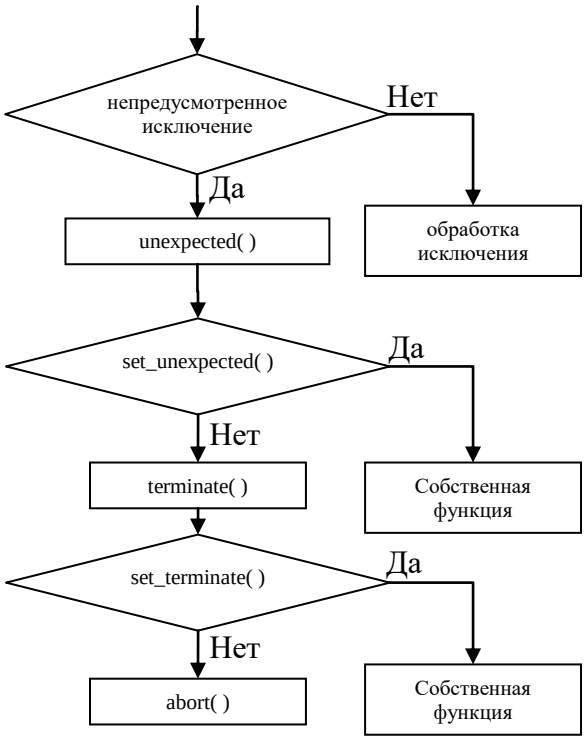


Рисунок 3.1. Алгоритм обработки исключения

вания, а также гарантию, что при возникновении непредвиденного исключения эта ситуация будет обнаружена.Алгоритм обработки исключения представлен на рис. 3.1.

Типы исключений перечисляются в скобках через запятую после ключевого слова *throw*, расположенного за списком параметров функции.

Листинг 3.59 Типы исключений

```
void fl() throw (int. const char*){ /* Тело функции */ }
void f2() throw (Oops*){ /* Тело функции */ }
```

Функция *fl* должна генерировать исключения только типов *int* и *const char**. Функция *f2* должна генерировать только исключения типа указателя на класс *Oops* или производных от него классов.

Если ключевое слово throw не указало, функция может генерировать любое исключение. Пустой список означает, что функция не должна порождать исключений.

Листинг 3.60 Функция, не порождающая исключений

```
void f() throw ()
{
    // Тело функции, не порождающей исключений
}
```

Исключения не входят в прототип функции. При переопределении в производном классе виртуальной функции можно задавать список исключений, такой же или более ограниченный, чем в соответствующей функции базового класса. Указание списка исключений ни к чему не обязывает — функция может прямо или косвенно породить исключение, которое она обещала не использовать. Эта ситуация обнаруживается во время исполнения программы и приводит к вызову стандартной функции *unexpected*, которая по умолчанию просто вызывает функцию *terminate*. С помощью функции *set_unexpected* можно установить собственную функцию, которая будет вызываться вместо *terminate* и определять действие программы при возникновении непредвиденной исключительной ситуации.

Функция *terminate* по умолчанию вызывает функцию *abort*, которая завершает выполнение программы. С помощью функции *set_terminate* можно установить собственную функцию, которая будет вызываться вместо *abort* и определять способ завершения программы. Функции *set_unexpected* и *set_terminate* описаны в заголовочном файле <*exception*>.

### 3.4.5 Исключения в конструкторах и деструкторах

Язык *C++* не позволяет возвращать значение из конструктора и деструктора. Механизм исключений дает возможность сообщить об ошибке, возникшей в конструкторе или деструкторе объекта. Для иллюстрации создадим класс *Vector*, в котором ограничивается количество запрашиваемой памяти.

Листинг 3.61 Пример порождения исключений в конструкторах и деструкторах

```
class Vector
{
public:
    class Size{};       // Класс исключения
    enum {max = 32000};   // Максимальная длина вектора
    Vector(int n)       // Конструктор
    {
        if (n<0  ||  n>max)
            throw Size(); ...
    }
    ----------
}
```

При использовании класса *Vector* можно предусмотреть перехват исключений типа *Size*:

Листинг 3.62 Порождение исключений типа *Size*

```
try
{
    Vector *p = new Vector(i);
}
catch(Vector::Size)
{
  ...  // Обработка ошибки размера вектора
}
```

В обработчике может использоваться стандартный набор основных способов выдачи сообщений об ошибке и восстановления. Внутри класса, определяющего исключение, может храниться информация об исключении, которая передается обработчику. Смысл этой техники заключается в том, чтобы обеспечить передач; информации об ошибке из точки ее обнаружения в место, где для обработки ошибки имеется достаточно возможностей.

Если в конструкторе объекта генерируется исключение, автоматически вызываются деструкторы для полностью созданных в этом блоке к текущему моменту объектов, а также для полей данных текущего объекта, являющихся объектами и для его базовых классов. Например, если исключение возникло при создании массива объектов, деструкторы будут вызваны только для успешно созданных элементов.

Если объект создается в динамической памяти с помощью операции *new* и в конструкторе возникнет исключение, память из-под объекта корректно освобождается.

### 3.4.6 Иерархии исключений

Использование собственных классов исключений предпочтительнее применения стандартных типов данных. С помощью классов можно более гибко организован передачу информации об исключении, легче дифференцировать обработку исключений, а, кроме того, появляется возможность использовать иерархии классов.

Поскольку механизм управления исключениями позволяет создать обработчиь для базового класса, родственные исключения часто можно представить в виде иерархии. Производя исключения от общего базового класса, можно в обработчике перехватывать ссылку или указатель на базовый класс, используя полиморфизм. Например, в математической библиотеке можно организовать классы следующим образом.

Листинг 3.63 Организация классов исключений в математической библиотеке

```
class Matherr{};
class Overflow: public Matherr{};        // Переполнение
class Underflow: public Matherr{}:       // Исчезновение порядка
class ZeroDivide: public Matherr{};      // Деление на ноль
```

Для представления ошибок ввода/вывода могут использоваться следующие классы:

Листинг 3.64 Представление ошибок ввода/вывода в математической библиотеке

```
class I0err{};
class Readerr: public I0err{};   // Ошибка чтения
class Writerr: public I0err{}:   // Ошибка записи
class Seekerr: public I0err{};   // Ошибка поиска
```

В зависимости от обстоятельств можно использовать либо обработчик исключений базового класса, который будет перехватывать и производные исключения либо собственные обработчики производных классов.

Существует ряд стандартных исключений, которые генерируются операциям или функциями *C++*. Все они являются производными от библиотечного класса *exception*, описанного в заголовочном файле <*stdexcept*>. Например, операция *new* при неудачном выделении памяти генерирует исключение типа *bad_alloc*.

Программист может определить собственные исключения, производные от стандартных.

Об обработке исключительных ситуаций рекомендуется прочитать в литературе:

1. Страуструп Б. Язык программирования C++. Пер. с англ. / Б. Страуструп. – М.: Радио и связь, 1991. –с. 407 - 442.

2. Подбельский В.В. Язык C++: Учебное пособие. – М.: Финансы и статистика, 1996. – стр. 445 - 488.

## Контрольные вопросы

1. Инкремент и декремент.
2. Объявление и определение функции.
3. Статические поля и методы класса
4. Условный оператор. Операции сравнения.
5. Механизм наследования. Ключи доступа.
6. Указатель на функцию, указатель на объект, указатель на void.
7. Назначение и особенности реализации рекурсивной функции.
8. Параметры функции. Виды передачи параметров
9. Класс. Описание класса. Поля и методы класса.
10. Цикл с предусловием.
11. Перегрузка унарных операций.
12. Способы инициализации указателей.
13. Перегрузка функций.
14. Директива #define.
15. Указатель this.
16. Цикл с параметром.
17. Перегрузка операции присваивания.
18. Перечисления.
19. Дружественная функция.
20. Описание объектов. Доступ к элементам и методам класса через объект и указатель на объект.
21. Условная операция.
22. Оператор switch.
23. Конструктор, его свойства.
24. Структуры.
25. Передача имен функций в качестве параметров.
26. Перегрузка оператора присваивания для класса.
27. Динамические массивы, выделение и освобождение памяти.
28. Операторы передачи управления. Безусловный переход.
29. Возвращаемое значение функции. Оператор return.
30. Абстрактные классы, их назначение.
31. Параметры функции со значениями по умолчанию.