Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информатика»

ПРОГРАММИРОВАНИЕ

ПОСОБИЕ
по одноименной дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Электронный аналог печатного издания

Гомель 2019

УДК 004.42(075.8)
ББК 32.973.22я73
К78

Рекомендовано к изданию научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 8 от 05.03.2018 г.*)

Составитель *Г. П. Косинов*

Рецензент: доц. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. физ.-мат. наук, доц. *Е. Г. Стародубцев*

К78 **Программирование** : пособие по одном. дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» днев. формы обучения / сост. Г. П. Косинов. – Гомель : ГГТУ им. П. О. Сухого, 2019. – 71 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-409-4.

Представлен материал для подготовки к лабораторным работам и экзамену, с помощью которого выполняется разработка и практическая реализация основных видов алгоритмов на языке высокого уровня C++. Рассматриваются сложные структуры данных и алгоритмы их обработки, а также алгоритмы вычислительной математики.

Для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения.

УДК 004.42(075.8)
ББК 32.973.22я73

ISBN 978-985-535-409-4

Глава 1. ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ С/С++

История создания и развития

Алгоритмический язык С был разработан в 1972 г. сотрудником фирмы AT&T Bell Laboratory Денисом Ритчи на базе языка *B* (автор – К. Томпсон). Язык продолжил развиваться и на его основе был разработан язык С++ Бьерном Страуструпом в начале 80-х гг. как дальнейшее развитие языка С. Язык С++ обладает рядом свойств, которые делают его более совершенным языком по сравнению с С.

С++ – это гибридный язык, он предоставляет возможность программировать и в стиле С, и в объектно-ориентированном стиле, и в обоих стилях сразу.

Язык С++ отражает возможности современных компьютеров и обладает следующими достоинствами: это современный, эффективный, переносимый (мобильный), мощный и гибкий, удобный и обладающий рядом присущих ассемблеру управляющих конструкций язык высокого уровня.

На основе, заложенной С и С+, появились такие языки, как С# и Java, сфера применения которых просто огромна.

Для разработки приложений на С++ чаще всего используется IDE (Integrated Development Environment или Интегрированная среда разработки) под названием Microsoft Visual Studio, которая отличается поддержкой множества различных языков программирования и удобством.

Элементы языка С++

Базовые элементы языка представлены на примере простейшей программы:

```
// Программа №1 - Первая С++-программа.
#include <iostream>
using namespace std;
int main()
{
cout << “моя специальность 1-40 04 01”;
return 0;
}
```

Первая строка этой программы – комментарий. Обычно комментарий – это текст пояснительного содержания, встраиваемый в программу, который компилятор игнорирует. Комментарии бывают одно-

строчные и многострочные. Многострочный комментарий выделяется с помощью символов /* и */. Перед однострочным ставятся символы //.

#include <iostream> – это препроцессорная директива. Препроцессорные директивы начинаются символом #, за которым следует наименование директивы, указывающее ее действие. Операция *#include* означает, что к основной программе необходимо подключить библиотеку *iostream*, которая будет отвечать за ввод и вывод данных.

using namespace std; – указывает, что компилятор должен использовать пространство имен *std*. Пространство имен (*namespace*) создает декларативную область, в которой могут размещаться различные элементы программы. Пространство имен позволяет хранить одно множество имен отдельно от другого. Другими словами, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом. Пространства имен позволяют упростить организацию больших программ. Ключевое слово *using* информирует компилятор об использовании заявленного пространства имен (в данном случае *std*). Именно в пространстве имен *std* объявлена вся библиотека стандарта C++.

int main() – любая C++-программа начинается с вызова функции *main()* и обычно заканчивается возвратом из функции *main()*. Открытая фигурная скобка на следующей (после *int main()*) строке указывает на начало кода функции *main()*. Ключевое слово *int* (сокращение от слова *integer* – целочисленный), стоящее перед именем main(), означает тип данных для значения, возвращаемого функцией main().

cout << "моя специальность 1-40 04 01";

Это функция вывода данных на консоль. При ее выполнении на консоли компьютера отобразится сообщение *"моя специальность 1-40 04 01"*. В функции cout используется оператор вывода *"<<"*. Он обеспечивает вывод выражения, стоящего с правой стороны. При этом выражений может быть несколько и все они будут разделятся этим оператором. Слово *cout* представляет собой встроенный идентификатор (составленный из частей слов *console output*), который в большинстве случаев означает консоль компьютера.

return 0;

При ее выполнении функция *main()* возвращает вызывающему процессу (в роли которого обычно выступает операционная система) значение 0. Для большинства операционных систем нулевое значение, которое возвращает эта функция, свидетельствует о нормальном за-

вершении программы. Другие значения могут означать завершение программы в связи с какой-нибудь ошибкой. Слово *return* относится к числу ключевых, используется для возврата значения из функции.

Константные выражения и операции. Объявление типов и объектов

Константами называют величины, которые не изменяют своего значения во время выполнения программы, т. е. это объекты, которым нельзя ничего присвоить, а также они не имеют адреса в памяти. Константами являются:

– самоопределенные арифметические константы целого и вещественного типов, символьные и строковые данные;

– идентификаторы массивов и функций;

– элементы перечислений.

Различают целочисленные, вещественные, символьные и строковые константы.

1. Целочисленные константы.

Общий формат записи: ±*n* (+ обычно не ставится). Например, 11 и 234 – обычные целые константы, если нужно ввести длинную целую константу, то указывается признак *L* – 234*L*. Для такой константы будет отведено – 4 байта. Обычная целая константа, которая слишком длинна для типа *int*, рассматривается как *long*. Для шестнадцатеричных констант используется последовательность цифр от 0 до 9 и букв – от *A* до *F*, начинающаяся символами 0*X*, например: 0*X*1*F* (*значение* $1F_{16}$ *соответствует* 31_{10}).

2. Вещественные константы.

Представляются в двух видах: с фиксированной и плавающей точкой. Константы с фиксированной точкой представляются в привычном нам виде, например, –2.621. Константы с плавающей точкой записываются с использованием символа E. Например, константа 0.25*E*–7 соответствует числу $2{,}5 \cdot 10^{-8}$, т. е. после E записывается степень числа 10. При использовании вещественных констант наличие так называемой десятичной точки обязательно!

3. Символьные и строковые константы.

Символьные и строковые константы представляются как последовательность символов, заключенная в кавычки. Если символ в кавычках только один, то говорят о символьной константе. Примеры символьных констант: '*A*', '*x*'. Если же говорится о строковой константе, то это любое количество символов, заключенных в кавычки. На-

пример: 'ИП41' , 'Время X'. Во внутреннем представлении к строковым константам добавляется пустой символ '\0', который не является цифрой 0, на печать не выводится (в таблице кодов *ASCII* имеет код = 0) и является признаком окончания строки. Также в строках используются специальные последовательности символов – управляющие (*escape*) последовательности:

\n – новая строка;
\t – горизонтальная табуляция;
\b – шаг назад;
\r – возврат каретки;
\v – вертикальная табуляция;
\f – переход на новую строку;
\\ – обратный слеш;
\' – апостроф;
\" – кавычки;
\0 – символ «пусто», не путать с символом ‘0’.

Типы данных

Данные в языке Си разделяются на две категории: простые (скалярные), будем их называть базовыми, и сложные (составные) типы данных.

Тип данных определяет:

– внутреннее представление данных в оперативной памяти;

– совокупность значений (диапазон), которые могут принимать данные этого типа;

– набор операций, которые допустимы над такими данными.

Основные типы базовых данных: целый – *int* (*integer*); вещественный с одинарной точностью – *float* и символьный – *char* (*character*). Также есть логический тип *bool*.

В свою очередь, данные целого типа могут быть короткими – *short*, длинными – *long* и беззнаковыми – *unsigned*, а вещественные – с удвоенной точностью – *double*.

Сложные типы данных – массивы, структуры (*struct*), объединения (*union*), перечисления (*enum*):

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

• *char*: представляет один символ в кодировке ASCII. Занимает в памяти 1 байт (8 бит). Может хранить любое значение из диапазона от –127 до 127, либо от 0 до 255;

• *signed char*: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от –127 до 127;

• *unsigned char*: представляет один символ. Занимает в памяти 1 байт (8 бит). Может хранить любой значение из диапазона от 0 до 255;

• *short*: представляет целое число в диапазоне от –32767 до 32767. Занимает в памяти 2 байта (16 бит). Данный тип также имеет синонимы *short int*, *signed short int*, *signed short*;

• *unsigned short*: представляет целое число в диапазоне от 0 до 65535. Занимает в памяти 2 байта (16 бит). Данный тип также имеет синоним *unsigned short int*;

• *int*: представляет целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита). Диапазон предельных значений, соответственно, также может варьироваться от –32767 до 32767 (при 2 байтах) или от −2 147 483 647 до 2 147 483 647 (при 4 байтах). Но в любом случае размер должен быть больше, или равен размеру типа *short* и меньше, или равен размеру типа *long*. Данный тип имеет синонимы *signed int* и *signed*;

• *unsigned int*: представляет положительное целое число. В зависимости от архитектуры процессора может занимать 2 байта (16 бит) или 4 байта (32 бита), и из-за этого диапазон предельных значений может меняться: от 0 до 65535 (для 2 байт), либо от 0 до 4 294 967 295 (для 4 байт). В качестве синонима этого типа может использоваться *unsigned*;

• *long*: представляет целое число в диапазоне от −2 147 483 647 до 2 147 483 647. Занимает в памяти 4 байта (32 бита). У данного типа также есть синонимы *long int*, *signed long int* и *signed long*;

• *unsigned long*: представляет целое число в диапазоне от 0 до 4 294 967 295. Занимает в памяти 4 байта (32 бита). Имеет синоним *unsigned long int*;

• *long long*: представляет целое число в диапазоне от −9 223 372 036 854 775 807 до +9 223 372 036 854 775 807. Занимает в памяти, как правило, 8 байт (64 бита). Имеет *синонимы long long int*, *signed long long int* и *signed long long*;

• *unsigned long long*: представляет целое число в диапазоне от 0 до 18 446 744 073 709 551 615. Занимает в памяти, как правило, 8 байт (64 бита). Имеет синоним *unsigned long long int*;

• *float*: представляет вещественное число ординарной точности с плавающей точкой в диапазоне +/– 3.4E–38 до 3.4E+38. В памяти занимает 4 байта (32 бита);

• *double*: представляет вещественное число двойной точности с плавающей точкой в диапазоне +/– 1.7E–308 до 1.7E+308. В памяти занимает 8 байт (64 бита);

• *long double*: представляет вещественное число двойной точности с плавающей точкой не менее 8 байт (64 бит). В зависимости от размера занимаемой памяти может отличаться диапазон допустимых значений;

• *void*: тип без значения.

Таким образом, все базовые типы данных за исключением void могут быть разделены на три группы: символьные (char); целочисленные (short, int, long, long long) и типы чисел с плавающей точкой (float, double, long double).

Идентификаторы

Идентификатор – имя любого объекта программы (переменной, константы, процедуры и др.).

Идентификатор может включать буквы**,** цифры и символ подчеркивания. Идентификатор не может начинаться с цифры. Прописные и строчные буквы в идентификаторах различаются, т. е. например, X и x – это разные идентификаторы. Длина идентификатора не ограничена. При выборе идентификатора необходимо иметь в виду следующее:

1) идентификатор не должен совпадать с ключевыми словами;

2) не рекомендуется начинать идентификаторы с двух символов подчеркивания, поскольку такие имена зарезервированы для служебного использования.

Объявление переменных и констант

Все переменные, используемые в программе должны быть описаны явным способом.

При описании задаются ее имя и тип:

тип имя1,[имя2, имя3, ...];

Например:

```
int dlina;
float b;
```

При описании можно инициализировать переменные:

```
int x = 1, b;
float y = 0.1;
```

При описании инициализировать переменные необязательно, но обязательно нужно их инициализировать перед вычислениями. Область действия переменной начинается в точке ее описания и распространяется до конца блока, внутри которого она описана. Блок – это код, заключенный в фигурные скобки. Имя переменной должно быть уникальным в области ее действия.

Предваряя переменную ключевым словом *const* при ее объявлении или инициализации, Вы объявляете ее как константу:

```
const int a=100;
```

Константы должны инициализироваться при объявлении и однажды присвоенные им значения не должны никогда меняться, значение константы должно быть вычислено при компиляции.

Указатели

Указатель – это адрес памяти, распределяемой для размещения идентификатора (в качестве идентификатора может выступать имя переменной, массива, структуры, строкового литерала). В том случае, если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находится скалярная величина любого типа. При объявлении переменной типа «указатель», необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек).

Пример использования:

```
#include <iostream>
using namespace std;
int main()
{
int x;
x = 10;
int *uk;
uk = &x
cout << “Это переменная x”;
cout << x;
cout << uk;
return 0;
}
```

При выполнении данной программы будет выведено число 10 и адрес в памяти, где данное число хранится.

Глава 2. ОПЕРАЦИИ И ВЫРАЖЕНИЯ

Понятие выражения, типы и значения

Выражение в С++ представляет собой последовательность операторов, операндов и знаков пунктуации, которые компилятор воспринимает как руководство к определенным действиям над данными. Тип выражения определяется типами операндов и операциями, которые участвуют в выражении. Значением выражения является результат, полученный после завершения всех операций.

Операции делятся на унарные, бинарные и тернарные – по количеству участвующих в них операндов, и выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки.

В языке Си используются четыре унарные операции, имеющие самый высокий приоритет, их часто называют первичными:

– операция доступа к полям структур и объединений при помощи идентификаторов «.» – точка;

– операция доступа к полям структур и объединений при помощи указателей «–>» – стрелка;

– операция [ ] индексации, используемая при декларации массива и обращении к его элементам;

– операция ( ) обращения к функции.

Арифметические операции:

Арифметические операции: + (сложение); – (вычитание); / (деление, для *int* операндов – с отбрасыванием остатка); * (умножение); % (остаток от деления целочисленных операндов со знаком первого операнда – деление «по модулю»).

Данные операции используются для работы с целыми и вещественными числами. Результатом операции будет число, соответствующее большему по разрядности операнду, например: сложение чисел типа *int* и *double* в результате дает число типа *double.*

Также часто используются сокращенные формы записи арифметических выражений:

```
x *= a       // полная форма записи  x=x*a
d += 2       // d=d+2
x /= y+7     //  x=x/(y+7)
--j          //  j=j-1
j--          //  j=j-1)
```

a =g++ // a=g g=g+1
a = ++g // g=g+1 a=g

Операции сравнения

Операции сравнения:
== – равно;
!= – не равно;
< – меньше;
> – больше;
<= – меньше либо равно;
>= – больше либо равно.

Операции сравнения можно применять к операндам любого типа. Значением операции является логическое значение *true* – если условие выполняется, либо *false* – в противном случае.

Примеры использования операций отношений:

a > 5 , x == z , f != 2

Логические операции

Логические операции:
&& – логическое И;
|| – логическое ИЛИ;
! – логическое НЕ.

В качестве операндов выступают логические значения, результат – также логическое значение *true* либо *false.* Старшинство операции «И» выше, чем «ИЛИ», и обе они младше операций отношения и равенства.

Примеры использования:

5 < x && x < 10 // 5<x<10
! (b < 4) // b>=4
x > 0 || x < -7 // x не входит в промежуток (-7;0)

Битовые операции

Битовые операции:
& – битовое И;
| – битовое ИЛИ;
^ – битовое ИСКЛЮЧАЮЩЕЕ ИЛИ;
~ – битовое НЕ;
<< – сдвиг влево;
>> – сдвиг вправо.

Побитовые операции применяются к целым числам и выполняются над каждым битом операнда. Результатом является целое число.

Пример:

3&1 = 1 = (битовая запись чисел) 00000011 & 00000001 = 00000001;

3|1 = 3 = 00000011 | 0000001 = 00000011;

~3 = 252 = ~00000011 = 11111100;

16 >> 4 = 1;00010000 >> 4 = 00000001;

4<< 2 = 16; 00000100 << 2 = 00010000;

Операция присваивания

Операция присваивания:

= присвоить

Операция присваивает левому операнду значение правого.

Также данную операцию можно комбинировать с алгебраическими и битовыми операциями. В результате к левому операнду сначала будет применена соответствующая операция с правым операндом, а затем присваивание.

Пример:

```
x = 6 + 4 * 2      // соответствует:  x = 14;
        x += 4          // соответствует: x  = x + 4;
        a = b = c =1;   // соответствует a =1  b =1 c =1
```

Приоритет и порядок вычислений

Порядок выполнения операций очень прост (в примерах подчеркнуты операции, которые выполняются первее):

1) операции внутри скобок; пример: a + (b + c);

2) операции обращения к функциям; пример: a + fun(i);

3) операции типа умножения: *, /, &&, & и т. д.; пример: a + b * c;

4) операции типа сложения: +, -, || и т. д.; пример: a > b + c;

5) операции отношений >, <, == и т. д.

Арифметические преобразования

Если операнды имеют разный тип, то результатом будет тип, который находится «выше» по рангу, чем остальные.

Ранги типов данных: *double > float > long > int > short > char*;

Примечание: ключевое слово unsigned повышает ранг типа к которому применяется.

Пример: *long > unsigned int > int;*

Если есть операция присваивания, то тип правого операнда будет такой же, какой тип имеет левый операнд.

Явные преобразования

Для преобразования типа данных в вычислениях, можно воспользоваться следующим способом:

int a;
(double) a или double(a);

При преобразовании из высокого ранга в более низкий возможна потеря точности. Преобразованный тип используется только в вычислениях, а сами переменные имеют тип, который описан при ее объявлении.

Пример:

```
int a = 47;
(double)a / 2      // результат  23.5;
a / 2              // результат  23;
```

Глава 3. ОПЕРАТОРЫ

Операторы языка

В программировании для решения поставленных задач часто используются различные структуры. Их бывает несколько видов.

Линейная – представляющая собой последовательное выполнение нескольких операторов. Ветвление – есть какое-то условие, исходя из которого программа выбирает, какой оператор использовать. Цикл – задает многократное выполнение одного или нескольких операторов. Операторы управления работой программы называют управляющими конструкциями программы. К ним относят:

– операторы выбора;
– операторы циклов;
– операторы перехода.

Операторы выбора

Операторы выбора – это условный оператор и переключатель. Условный оператор имеет полную и сокращенную форму.

Форма записи оператора:

```
if (выражение-условие)  оператор1; //сокращенная форма
if (выражение-условие) оператор1; else оператор2; //полная
```

В качестве выражения-условия могут использоваться арифметическое выражение, отношение и логическое выражение. Если значение выражения-условия выполняется (т. е. истинно или true), то передается управление оператору1, при невыполнении выражения-условия (т. е. ложно или false) программа переходит на выполнение оператора2.

Примеры использования:

```
    if (x<y&&x<z)min=x;
if (x<y)min=x; else min=y;
```

Переключатель определяет множественный выбор:

```
switch (выражение)
{
case константа1: оператор1;
case константа2: оператор2;
. . . . . .
[default: операторы;]
}
```

При выполнении оператора switch вычисляется выражение (или просто переменная), записанное после switch оно должно быть целочисленным. Полученное значение последовательно сравнивается с константами, которые записаны следом за case. При первом же совпадении выполняются операторы, помеченные данной меткой. Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель. Если значение выражения, записанного после switch, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой default. Метка default необязательна.

Пример использования:

```
#include <iostream.h>
void main()
{
int choice;
cout<<"\nEnter your choice";
cin>>choice;
switch(choice)
{
case 1:cout <<"\n It is one";
case 2:cout <<"\n It is two";
```

```
case 3: cout <<"\n It is three "; break; //break – выход из switch
case 4: cout << "\n It is four ";
default: cout<<"\n The end of work";
}
}
```

Операторы циклов

Различают итерационные и арифметические циклы. Группа действий, повторяющихся в цикле, называется его телом. Однократное выполнение цикла называется его шагом или итерацией. В итерационных циклах известно условие выполнения цикла.

Форма записи цикла с предусловием:

```
while (выражение-условие)
операторы;
```

В качестве <*выражения-условия*> чаще всего используется отношение или логическое выражение. Если оно истинно, т. е. *true*, то операторы внутри тела цикла выполняются до тех пор, пока выражение-условие не станет ложным.

Пример использования:

```
cin>>it;
while (it!=0)
{
summa+=it;
cin>>it;
}
```

Форма записи цикла с постусловием:

```
do
операторы
while (выражение-условие);
```

Тело цикла выполняется до тех пор, пока выражение-условие истинно. Также у этого цикла будет хотя бы одна итерация.

Пример использования:

```
do
{
cin>>it;
summa+=it;
```

```
}
while (it!=0);
```

Форма записи цикла с параметром:

```
for ( выражение1;условие;выражение3)
операторы;
```

Здесь *выражение1* и *выражение3* могут состоять из нескольких выражений, разделенных запятыми. *Выражение1* – задает начальные условия для цикла (инициализация). У*словие* определяет условие выполнения цикла, если оно не равно *false*, цикл выполняется, а затем вычисляется значение *выражения3*. *Выражение3* – задает изменение параметра цикла или других переменных (коррекция). Цикл продолжается до тех пор, пока выражение-условие не станет ложно. Любое выражение может отсутствовать, но разделяющие их «;» должны быть обязательно.

Приведем примеры использования цикла с параметром.

Уменьшение параметра цикла i:

```
for ( i=9; i>0; i--)
```

Проверяется условие отличное от условия, которое налагается на число итераций:

```
for ( j=1;j*j<100; j++)
```

Можно использовать несколько инициализирующих или корректирующих выражений:

```
for (t=1, y=0; t<9;t++;y+=t);
```

Операторы перехода

Операторы перехода выполняют безусловную передачу управления. Оператор break используется в случае, если цикл продолжать больше не нужно и найдено необходимое значение. Управление в этом случае передается на первый оператор, находящийся после тела цикла.

Пример программы, которая ищет сумму чисел, вводимых с клавиатуры, до тех пор, пока не будет введено 10 чисел или 0:

```
summa=0;
for( i=1; i<10;i++)
{
cin>>x;
if( x==0) break; // если ввели 0, то суммирование заканчивается
```

```
summa+=x;
}
```

Оператор *continue* позволяет не выполнять оставшиеся операторы в теле цикла, а перейти к следующей итерации цикла.

Пример, в котором подсчитывается количество и сумма положительных чисел; при вводе числа 0 – цикл завершается:

```
for( k=0,summa=0,x=1; x!=0;)
{
cin>>x;
if (x<=0) continue;
k++;s+=x;
}
```

Оператор *goto* имеет формат*: goto метка*. В теле той же функции должна присутствовать конструкция: *метка:оператор;*. Метка – это обычный идентификатор, областью видимости которого является функция. Оператор *goto* передает управления оператору, стоящему после метки. Использование оператора *goto* оправдано, если необходимо выполнить переход из нескольких вложенных циклов или переключателей вниз по тексту программы или перейти в одно место функции после выполнения различных действий. Применение *goto* нарушает принципы структурного и модульного программирования, по которым все блоки, составляющие программу, должны иметь только один вход и только один выход.

Оператор *return* – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора: *return [выражение];*

Глава 4. ФУНКЦИИ

Со временем код программы становится все объемнее и сложнее. Очевидным выходом является разбить программу на части. Такими частями являются функции. Таким образом, мы избавляемся от избыточного кода, ведь функцию можно вызвать несколько раз из разных частей программы. Кроме того, упрощается отладка программы. Часто используемые функции можно помещать в отдельные библиотеки.

Функция – это вспомогательный алгоритм, который может принимать аргументы. Также функция может возвращать значение-результат. Результатом может быть число, символ или объект другого типа.

Каждая функция должна быть определена, для этого в ней должны быть указаны:

– тип возвращаемого значения;

– имя функции;

– информация о формальных аргументах;

– тело функции.

Любая функция записывается следующим образом:

ТипВозвращаемогоЗначения ИмяФункции (СписокФормальныхАргументов)

```
{
  ТелоФункции;
 ...
 return (ВозвращаемоеВыражение);
}
```

Если функция не возвращает значения, то тип возвращаемого значения для нее указывается как void. При этом операция return может быть опущена.

Разбиение программ на функции дает следующие преимущества:

– функцию можно вызвать из различных мест программы, что позволяет избежать повторения программного кода;

– одну и ту же функцию можно использовать в разных программах;

– функции повышают уровень модульности программы и облегчают ее проектирование;

– использование функций облегчает чтение и понимание программы и ускоряет поиск и исправление ошибок.

Способы объявления функции:

– до функции main;

– после функции main, используя прототип функции.

Пример описания до функции main:

ТипВозвращаемого ЗначенияИмя Функции (СписокФормальныхАргументов)

```
{
  ТелоФункции;
 ...
 return (ВозвращаемоеЗначение);
}
      int main()
      {
```

```
  return 0;
}
```

Пример с использованием прототипа:

```
ТипВозвращаемогоЗначения ИмяФункции (СписокФормальныхАргументов);
int main()
{
  return 0;
}
ТипВозвращаемогоЗначения ИмяФункции (СписокФормальных Аргументов)
{
  ТелоФункции;
 ...
 return (ВозвращаемоеЗначение);
}
```

Общий вид вызова функции:

```
Переменная = ИмяФункции (СписокФактическихАргументов);
```

Фактический аргумент – это величина, которая присваивается формальному аргументу при вызове функции. Таким образом, формальный аргумент – это переменная в вызываемой функции, а фактический аргумент – это конкретное значение, присвоенное этой переменной вызывающей функцией. Фактический аргумент может быть константой, переменной или выражением.

Если в функцию требуется передать несколько значений, то они записываются через запятую. При этом формальные параметры заменяются значениями фактических параметров в порядке их следования в сигнатуре функции.

Рекурсия – определение части функции (метода) через саму себя, т. е. это функция, которая вызывает саму себя, непосредственно (в своем теле) или косвенно (через другую функцию).

Пример:

```
int a = 5;
int rec(int x)
{
  cout << x << " ";
  if(x > 0) return rec(x-1);
```

```
  cout << x << " ";
}
```

Результат работы рекурсивной функции представлен ниже:

5 4 3 2 1 0 0 1 2 3 4 5

При описании функций можно использовать одно и тоже имя, но функции должны отличаться количеством/типом параметров. При вызове функции будет вызвана та, которой будут соответствовать передаваемые значения. Объявление функций с одним именем и разным количеством параметров называется перегрузка функции.

Пример:

```
int sum(int a)
{
  return a + 7;
}
int sum(int a, int b)
{
  Return a + b + 7;
}
```

Все программы, написанные на языке С, содержат основную функцию, которая должна иметь имя *main*. Функция main служит в качестве начальной точки выполнения программы. Она обычно управляет выполнением программы, вызывая другие ее функции. Выполнение программы завершается, как правило, в конце сегмента *main*, хотя по разным причинам оно может завершаться и в других местах программы. Функция *main* может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Для передачи этих строк в функцию *main* используются два параметра, общепринятые (необязательные) идентификаторы которых – argc и argc:

```
int main (int argc, char *argv[])  ...
```

Параметр argc имеет тип int, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой функции. Параметр argv – это массив указателей на строки, каждая из которых содержит одно слово из командной строки.

Глава 5. ФУНКЦИИ РАБОТЫ С ПАМЯТЬЮ

Адресная арифметика

Под адресной арифметикой понимаются действия над указателями, связанные с использованием адресов памяти:

1. Присваивание. Указателю можно присвоить значение адреса. Любое число, присвоенное указателю, трактуется как адрес памяти:

```
int * adr1,*adr2;
int X;
adr1 = &X;            // указателю присвоен адрес переменной X
adr2 = (int *)0x00AB;  // указателю присвоен шестнадцатеричный адрес
```

2. Взятие адреса. Так как указатель является переменной, то для получения адреса памяти, где расположен указатель, можно использовать операцию взятия адреса &:

```
int *x,*y;
x= &y;  // указателю x присвоен адрес указателя y
```

3. Косвенная адресация. Для того чтобы получить значение, хранящееся по адресу, на который ссылается указатель, или послать данное по адресу, используется операция косвенной адресации *:

```
int *adr1;
int x;
int y = 2;
adr1 = &y;      // adr1 присвоен адрес переменной y
x = *adr1;      // переменная x примет значение 2
*adr1 = 3;    // переменная y примет значение 3
```

4. Преобразование типа. Указатель на объект одного типа может быть преобразован в указатель на другой тип. Операция преобразования типа указателя применяется в виде *(<тип> *)<указатель>*:

```
int i, *adr1;
i = 35;
adr1 = &i;
cout << *adr1; // печатается значение 35
cout << *((char *)adr1); // ptr преобразован к типу char
```

Преобразование типа указателя чаще всего применяется для приведения указателя на неопределенный тип данных void к типу объекта, доступ к которому будет осуществляться через этот указатель.

5. Определение размера. Для определения размера указателя можно использовать операцию «размер» в виде *sizeof(<указатель>)*. Размер памяти, отводимой компилятором под указатель, зависит от модели памяти. Для близких указателей операция *sizeof* дает значение 2, для дальних – 4.

6. Сравнение. Сравнение двух указателей любой из операций отношения имеет смысл только в том случае, если оба указателя адресуют общий для них объект, например, строку или массив.

7. Индексация. Указатель может индексироваться применением к нему операции индексации, обозначаемой в Си квадратными скобками [ ]. Индексация указателя имеет вид *<указатель>[<индекс>]*, где *<индекс>* записывается целочисленным выражением. Возвращаемым значением операции индексации является данное, находящееся по адресу *(адрес в указателе) + (значение <индекс>) * sizeof(<тип>)*, где *<тип>* – это тип указателя. Рассмотрим следующий пример:

```
int *adr1;
int x,y;
adr1 = &x; // в adr1 адрес переменной x
y = 4;
x = adr1[y];  // переменной x  присваивается значение int,
                                    // взятое из адреса на 8 (y*sizeof(int))
большего, чем
                                    // адрес переменной x;
adr1[y] = 55;  // в адрес на 8 больший, чем адрес переменной x,
                                    // записывается 55
```

8. Увеличение/уменьшение. Если к указателю применяется операция увеличения ++ или уменьшения --, то значение указателя увеличивается или уменьшается на размер объекта, который он адресует:

```
long *adr1;
long x;
adr1 = &x; // в adr1 адрес переменной x
adr1++;   //  adr1увеличивается на 4 (sizeof(long))
```

Следующие операции недопустимы с указателями:

– сложение двух указателей;
– вычитание двух указателей на различные объекты;
– сложение указателей с числом с плавающей точкой;
– вычитание из указателей числа с плавающей точкой;
– умножение указателей;
– деление указателей;
– поразрядные операции и операции сдвига;

Указатели на указатели

Указатели могут ссылаться на другие указатели. При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей. Число символов * при объявлении указателя показывает порядок указателя. Чтобы получить доступ к значению, на которое ссылается указатель, его необходимо разыменовывать соответствующее количество раз.

Пример:

```
int x = 7; // инициализация переменной x числом 7
int *adr1 = &x; // указатель на переменную adr1
int **adr2 = &adr1; // указатель на указатель на переменную x
int ***adr3 = &adr2;

cout << " x= " << x << endl;
cout << " *adr1= " << *adr1 << endl;
cout << " **adr2= " << **adr2 << endl;
cout << " ***adr3 = " << ***adr3 << endl;
```

Указатели на функции

Указатели могут ссылаться на функции. Имя функции, как и имя массива само по себе является указателем, т. е. содержит адрес входа. Формат записи:

типДанных (имяУказателя)(списокАргументовФункции);*

Тип данных определяем такой, который будет возвращать функция, на которую будет ссылаться указатель. Имя указателя берется в круглые скобки, чтобы показать, что это указатель на функцию, а не указатель на тип данных. После имени указателя идут круглые скобки, в этих скобках перечисляются все аргументы через запятую, как в объявлении прототипа функции. Аргументы наследуются от той функции, на которую будет ссылаться указатель. Это может понадобится, когда мы хотим отправить функцию как параметр другой функции.

Динамическое выделение памяти

В С работать с динамической памятью можно при помощи соответствующих функций распределения памяти (*calloc*, *malloc*, *free*), для чего необходимо подключить библиотеку:

```
#include <malloc.h>
```

C++ использует новые методы работы с динамической памятью при помощи операторов *new* и *delete*:

– *new* – для выделения памяти;

– *delete* – для освобождения памяти.

Оператор *new* используется в следующих формах:

```
new тип;                //для переменных
new тип[размер];        // для массивов
```

Память может быть распределена для одного объекта или для массива любого типа, в том числе типа, определенного пользователем. Результатом выполнения операции *new* будет указатель на отведенную память, или нулевой указатель в случае ошибки:

```
int *adr1= new int;
double *ptr_d= new double[10];
```

Память, отведенная в результате выполнения *new*, будет считаться распределенной до тех пор, пока не будет выполнена операция *delete.*

Освобождение памяти связано с тем, как выделялась память – для одного элемента или для нескольких. В соответствии с этим существуют и две формы применения *delete*:

```
delete указатель;  // для одного элемента
delete[] указатель;  // для массива
```

Например, для приведенного выше случая освободить память необходимо следующим образом:

```
delete adr1; или   delete[] adr2;
```

Освобождаться с помощью *delete* может только память, выделенная оператором *new*.

Глава 6. МАССИВЫ И СТРОКИ

Массив – это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем. Массивы используются для обработки большого количества однотипных данных. Имя массива является указателем на первый элемент массива. Отдельная ячейка данных массива называется элементом массива. Элементами массива могут быть данные любого типа. Так как элементы массива хранятся в памяти компьютера друг за другом, то мы можем обращаться к ним при помощи индексатора. Одномерный массив – массив с од-

ним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив – это массив, у которого может быть только одна строка и n-е количество столбцов.

Одномерный массив объявляется следующим образом:

ТипЭлементовМассива ИмяМассива = new ТипЭлементовМассива[РазмерМассива];

Пример объявления массива из 10 целых чисел:

int Arr = new int[10];

Массив, как и переменные, может быть инициализированным и наоборот. Инициализированный массив создается как и любой другой массив и элементы массива записываются в фигурных скобках.

Пример:

int Arr[10] = {1, 2, 3, 4, 5, 6, 7 , 8, 9, 0};

Пример ввода элементов массива:

for (int i = 0; i < n; i++) cin >> Arr[i];

Обращение к элементам массива происходит при помощи индексаторов. Например, Arr[0] – первый элемент массива, а Arr[n - 1] – последний элемент, где n – размер массива. Далее с элементами массива можно работать как с обычными переменными.

Строка – это одномерный массив типа *char*. Создается как и обычный массив:

char Arr = new char[100]; или *char Arr[100] = "String";*

Значения для строки присваиваются в двойных кавычках. Любая строка заканчивается нулевым байтом '\0' – символом, означающим конец строки. Работать со строкой можно так же как и с обычным массивом. Единственное отличие в том, что есть ряд встроенных функций, упрощающих работу со строкой:

strlen(str) – определяет длину строки без нулевого байта;

strcpy(str1, str2) – копирует строку str2 в строку str1;

strcat(str1, str2) – объединяет строки str2 и str1, результат будет находиться в строке str1;

strcmp(str1, str2) – сравнивает строку str1 со строкой str2 и возвращает результат типа int: 0 – если строки эквивалентны, >0 – если str1<str2, <0 – если str1>str2 с учетом регистра.

Глава 7. СЛОЖНЫЕ ТИПЫ ДАННЫХ

Перечисления

Перечисление – это тип данных, который включает множество именованных целочисленных констант. Объявляются перечисления следующим образом:

enum capacity { a,b,c }; // объявление типа

Здесь *enum* – ключевое слово; *capacity* – имя типа «перечисление»; *a*, *b*, *c* – сами перечисляемые константы. При объявлении типа «перечисление» его значения могут инициализироваться произвольными целочисленными константами:

enum capacity { a = 8, b = 16, c = 32 }; // объявление типа

Если инициализация отсутствует, то перечисляемым константам присваиваются последовательные значения: 0, 1, 2, …:

enum capacity { a, b, c }; // a = 0, b = 1, c = 2

Переменная типа перечисление объявляется следующим образом:

capacity d; // объявление переменной d, которая имеет тип capacity

Переменным перечисляемого типа можно присваивать только именованные значения перечислимых констант. Целочисленным переменным можно присваивать значения перечисляемых констант. Например:

int e = b;

Структуры

Структура – это составной объект языка Си, представляющий собой совокупность логически связанных данных различных типов, объединенных в группу под одним идентификатором. Данные, входящие в эту группу, называют полями. Определение объектов типа структуры производится за два шага:

– декларация структурного типа данных, не приводящая к выделению участка памяти;

– определение структурных переменных объявленного структурного типа с выделением для них памяти.

Объявляются структуры следующим образом:

struct emp
{

```
int id;
char name[20];
float  age ;
};
```

Здесь *struct* – ключевое слово; *emp* – имя типа структуры; *id*, *name*, *age* – поля структуры.

Переменная типа структура объявляется следующим образом:

emp driver; // объявление переменной driver, которая имеет тип emp

Инициализируются структуры так же как и массивы:

emp driver = { 10, "Ivan", 20 };

Так как члены структуры описываются в блоке, то их имена принадлежат локальной области видимости внутри этого блока. Для доступа к элементу структуры используется оператор точка '.':

```
driver.id = 20;
driver.name = "Ira";
driver.age = 52;
```

Членами структуры могут быть массивы и другие структуры. Для доступа к членам вложенной структуры оператор '.' используется столько раз, какова вложенность структуры:

```
struct inside
{
int b;
int c;
};
struct outside
{
int a;
struct inside d;
};
void main ()
{
Outside  x = { 0, {1,2} };
cout << x.a << x.d.b << x.d.c; // печатает:  0 1 2
}
```

Объединения

Объединение – поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента. Как и в случае структуры, члены объединения могут иметь любой тип, за исключением типов *void* и функция.

Объявляются объединения следующим образом:

```
union ads
{
int i;
float b;
};
```

Здесь *union* – это ключевое слово; *ads* – имя типа «объединение»; *i*, *b* – члены объединения.

Переменная типа «объединение» объявляется следующим образом:

```
ads d; // объявление переменной d, которая имеет тип num
```

и также называется объединением. Объявление типа «объединение» и переменной, которая имеет этот тип, может быть объединено в одну инструкцию:

```
union num
{
int n;
double f;
} d; // d – переменная типа num
```

При объявлении переменной типа «объединение» ее можно инициализировать значением, которое должно иметь тип первого члена объединения:

```
num d = { 1 }; // правильно, d = 1
num d = { 1.0 }; // ошибка
```

Как и в случае со структурами, для доступа к элементу объединения используется оператор точка ‘.’:

```
num e, g;
e.n = 1;
e.f = 1.1;
```

Битовые поля

Битовым полем называется член структуры или объединения, который определяет последовательность бит. Битовое поле может иметь один из следующих типов: int, unsigned или signed. При объявлении битового поля после его имени указывается длина поля в битах:

```
struct demo
{
unsigned a: 8; // 8 бит
signed b: 7;   // 7 бит
int c: 6;      // 6 бит
};
```

Длина битового поля должна быть неотрицательным целым числом и не должна превышать длины базового типа данных битового поля. Доступ к элементам битового поля осуществляется так же как и доступ к обычным членам структуры. Обычно битовые поля используются для установки различных флагов:

```
struct demo
{
unsigned a: 8;
signed b: 7;
int c: 6;
};

int main ()
{
demo sample;
sample.a = 15;
sample.b = -22;
sample.c = -43;
cout << sample.a << endl; // печать: 15
cout << sample.b << endl; // печать: -22
cout << sample.c << endl; // печать: -43
return 0;
}
```

Использование ключевого слова typedef для определения собственных типов данных

В язык Си введено специальное средство, позволяющее назначать имена типам данных (переименовывать). Таким средством является оператор *typedef*. Он записывается в следующем виде:

typedef тип имя;

Здесь «тип» – любой разрешенный тип данных и «имя» – любой разрешенный идентификатор. Рассмотрим пример:

typedef int HR;

После этого можно сделать объявление:

HR a, b;

Typedef можно использовать и для своих структур, объединений и т. д.

Глава 8. ВВОД-ВЫВОД

При использовании библиотеки классов Си++ необходимо подключить библиотечный файл iostream.h, в котором определены стандартные потоки ввода данных от клавиатуры cin и вывода данных на экран дисплея cout, а также соответствующие операции:

1) << – операция записи данных в поток;

2) >> – операция чтения данных из потока.

Пример:

```
cout << "Введите количество элементов: " << endl;
cin >> n;
```

Здесь *endl* – это переход на новую строку.

Передавая данные в поток, мы можем передавать их в переменные, выводить на консоль, либо записывать в файл. Для работы с файлами в Си++ необходимо подключить заголовочный файл <fstream>. Работа с файлами происходит аналогично работе с данными с использованием cin и cout. Для самой работы мы должны создать объект класса:

```
ifstream fin("file.txt");
ofstream fout("file.txt");
```

Здесь *ifstream* используется только для чтения из файла, а *ofstream* – только для записи в файл. Далее существует ряд методов

для комфортной работы с файлами. Методы применяются к объектам и для того чтобы применить их, необходимо прописать их после объекта через точку.

Методы:

• *fout.open(file.txt)* – связывает объект с файлом.

В круглых скобках указывается имя файла. Если объект создан для записи в файл, то указанный файл будет создан в текущей директории с программой. Если файл с таким именем существует, то существующий файл будет заменен новым:

• *fout.close()* – закрывает файл.

Чтение и запись в файл:

char str[100];

fin >> str; – запись строки из файла в переменную str. После записи каретка смещается в начало следующей строки, после чего этим же способом можно получить значение следующей строки.

fout << "Строка для записи в файл."; – записывает в файл строку.

Манипуляторы – специальные функции, возвращающие модифицированные данные потока. В большинстве случаев их использование позволяет форматировать данные, как при выводе, так и в оперативной памяти. Для их использования необходимо вместо файла *iostream.h* подключить заголовочный файл ***iomanip.h*** (манипуляторы для вывода потоками).

Рассмотрим работу некоторых манипуляторов на конкретном примере:

```
#include<iomanip.h>
main()
{
        int a = 789;
        double b = 4.32123;
        cout << setw(10) << a << endl;
/* Манипулятор setw(n) – устанавливает ширину поля, т. е. n позиций, для вывода объекта. На экране первые 7 позиций – пустые, а еще 3 – само число 789 (заполнение пробелами неиспользуемой части). Действует только для следующего за ним объекта. */
        cout << setw(10) << setfill('_') << a << endl;
```

```
/* Манипулятор setwfill(kod) – устанавливает заполнитель пробелов). На экране:  _____789. Действует до изменения или отмены setwfill(0) .*/
        cout << setprecision(3) << b << endl;
/* setprecision(n) – устанавливает n значащих цифр после запятой с учетом точки или без нее, в зависимости от системы программирования. На экране:
4.32  или   4.321  */
        return 0;
}
```

Глава 9. СПИСКИ ДАННЫХ

Линейные динамические списки организованы через структуры данных, элементы которых имеют тип *struct*. В каждой такой структуре имеются поля, в которых хранятся какие-то данные (информационная часть), а также указатель (адресная часть) на область размещения следующего элемента. Главным достоинством списковых конструкций является простота операций вставки и удаления элементов из списка, а также то, что каждый элемент списка может находиться в любом месте оперативной памяти (для массива должен выделяться непрерывный участок памяти). Основной недостаток списков – последовательный доступ к элементам.

Список называют однонаправленным (односвязным), если в его структуре есть указатель на следующий элемент списка. Если добавить в каждый элемент ссылку на предыдущий, получится **двунаправленный** список (двусвязный), если же последний элемент связать указателем с первым, получится **кольцевой** список.

Пример структуры простейшего однонаправленного списка:

```
struct List {
        int key;
        List *Next;
} ;
```

Над списками можно выполнять следующие операции:

– создание списка;

– удаление элемента с заданным ключом;

– вставка элемента в заданное место списка (до или после элемента с заданным ключом);

– упорядочивание списка по ключу.

Глава 10. СПИСКОВЫЕ КОНСТРУКЦИИ

Структура данных СТЕК

Стек является разновидностью линейных списков, в которых добавление новых элементов и удаление существующих производится только с одного его конца, который называют вершиной стека. Графически это можно изобразить так, как на рис. 10.1.

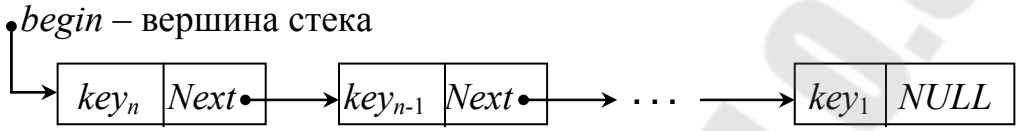

Рис. 10.1. Стек

Стек организован по принципу ***LIFO*** (*Last In, First Out*) – последним вошел, первым вышел. Стек получил свое название из-за схожести с магазином с патронами (обойма): когда в стек добавляется новый элемент, то прежний проталкивается вниз и становится недоступным. Когда же верхний элемент удаляется из стека, следующий за ним поднимается вверх и становится опять доступным.

Операции, выполняемые над стеком:

– *push* – втолкнуть элемент в стек;

– *pop* – извлечь элемент из стека;

– *peek* – прочитать значение элемента в вершине стека, не извлекая его оттуда.

Алгоритм формирования стека

1. Необходимо описать структуру для стека, содержащую информационное и адресное поля:

```
struct Stack {
        int key;
        Stack *Next;
} ;
```

2. Объявить указатели на структуру:

*Stack *begin (вершина стека), *current (текущий элемент);*

3. Для пустого стека присвоить:

begin = NULL;

4. Выделить память под первый (текущий) элемент:

current = (Stack) malloc (sizeof(Stack)); или current = new Stack;*

5. Ввести информацию для нового элемента стека:

а) формирование информационной части *current -> key (1)* (обычно вводим с клавиатуры);

б) формирование адресной части: значение адреса вершины стека записываем в адресную часть текущего элемента (там был *NULL*):

current -> Next = begin;

key (1)	*NULL*

6. Переносим вершину стека созданный элемент:

begin = current;

в результате получаем следующее:

begin →

key (1)	*NULL*

7. Захватываем память под второй элемент:

current = (Stack) malloc (sizeof(Stack)); или current = new Stack;*

Формируется конкретный адрес для второго элемента.

8. Вводим информацию для второго элемента:

а) формирование информационной части *current ->key (2)*

б) в адресную часть записываем значение адреса вершины стека:

current -> Next = begin;

9. Вершину стека устанавливаем на добавленный элемент:

begin = current;

Получается следующая цепочка:

begin →

key (2)	*Next* →

key (1)	*Next = NULL*

Функция формирования элемента стека:

```
Stack* Push(Stack *begin) {
Stack *current = new Stack;
printf("\n Введи число  ");
scanf("%d", &current -> key);
            current -> Next = begin;
```

```
return current;
        }
```

Добавление необходимого количества элементов в стек происходит в цикле:

```
    …
Stack *begin = NULL;
int choice = 1;
while(choice)   {
    begin = Push(begin);
    printf("Добавлять элемент ? 1- да, 0- нет ");
    scanf("%d", &choice);
}
```

Алгоритм извлечения элемента из стека

1. Устанавливаем текущий указатель на вершину стека:

current = begin;

2. Обрабатываем информационную часть текущего элемента:

(*current ->key);*

3. Переставляем указатель вершины стека на следующий элемент:

begin = current->Next;

4. Освобождаем память уже обработанного элемента:

free(current)

Просмотр стека

1. Устанавливаем текущий указатель на вершину СТЕКА:

current = begin;

2. Начинаем выполнять цикл до тех пор, пока текущий указатель current не станет равен *NULL*, т. е. пока не обработаем последний элемент, в адресной части которого находится значение *NULL*.

3. Обрабатываем (выводим) информационную часть текущего элемента:

cout << current->key;

4. Переставляем текущий указатель на следующий элемент:

current = current -> Next;

Функция просмотра стека:

```
void View(Stack *begin)
{
Stack *current = begin;
if(begin == NULL) {
puts(“ Стек пуст! ”);
return;
            }
while( current != NULL) {
cout << current->key;
                    current = current -> Next;
            }
      }
```

Обращение к этой функции:

```
View(begin);
```

Алгоритм удаления стека

1. Начинаем цикл, выполняющийся, пока *begin* не станет равным *NULL*.

2. Устанавливаем текущий указатель на вершину стека:

```
current = begin;
```

3. Вершину стека переставляем на следующий элемент:

```
begin = current->Next;
```

4. Уничтожаем текущий элемент, т. е. освобождаем занятую под него память *free*(*current*).

Функция освобождения памяти, занятой стеком, будет выглядеть следующим образом:

```
Stack* Delete_Stack(Stack *begin) {
Stack *current;
while( begin != NULL) {
current = begin;
begin = begin -> Next;
free(current);
            }
            return begin;
      }
```

Структура данных «ОЧЕРЕДЬ»

Очередь – структура данных, организованная по принципу ***FIFO*** (*First In, First Out* – первый вошел, первый вышел).

При работе с очередью обычно помимо текущего указателя используют еще два указателя, первый указатель устанавливается на начало очереди (удаление элемента), а второй – на ее конец (добавление элемента).

Структура очереди, информационной частью которой является целое число:

```
struct Queue {
                int key;
                Queue *Next;
        };
```

При организации очереди обычно используют три указателя:

*Queue *current, *begin, *end;*

Здесь *current* – указатель на текущий элемент очереди, а *begin* и *end* – указатели на начало и конец очереди, соответственно (рис. 10.2).

Рис. 10.2. Очередь

Основные операции с очередью следующие:
– создание очереди;
– добавление нового элемента в конец очереди;
– удаление элемента из очереди.

Создание очереди

Этот этап заключается в создании первого элемента, для которого адресная часть должна быть нулевой (*NULL*). Также на этот элемент должны ссылаться указатели *begin u end*. Порядок действий:

1) выделить память под первый элемент:

current = (Queue) malloc(sizeof(Queue)); или current = new Queue*;

в результате формируется конкретный адрес (*A*1) для первого элемента;

2) сформировать информационную часть *current -> key (1)*;

3) в адресную часть первого элемента занести *NULL*:

current -> Next = NULL;

4) указателям на начало и конец очереди присвоить значение *current*:

begin = end = current;

На этом этапе получим следующее (рис. 10.3).

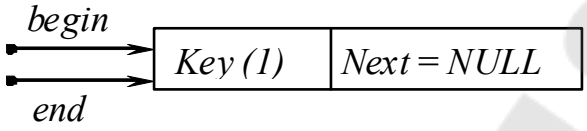

Рис. 10.3. Очередь из одного элемента

Добавление элемента в очередь

1. Выделяем память под новый элемент:

current = (Queue) malloc (sizeof(Queue)); или current = new Queue;*

2. Формируем информационную часть *current->key (2)*;

3. В адресную часть созданного элемента заносим *NULL* (он становится последним в очереди):

current -> Next = NULL;

4. Бывший последний элемент становится предпоследним (за ним становится созданный):

end -> Next = current;

5. Переставляем указатель последнего элемента на созданный:

end = current;

В результате получим следующее (рис. 10.4).

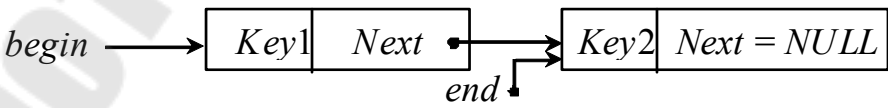

Рис. 10.4. Добавление элемента в очередь

Функция формирования очереди:

```
void Create(Queue **begin, Queue **end)  {
Queue *current = (Queue*) malloc(sizeof(Queue));
```

```
printf("\n Введи число  ");
scanf("%d",  &current -> key);
current -> Next = NULL;
if(*begin == NULL)          // Формирование первого элемента
            *begin = *end = current;
      else {
                  (*end) -> Next = current;          //  Добавление в конец
                  *end = current;
}
      }
```

Участок программы с обращением к функции *Create*:

```
…
Queue *begin = NULL, *end=NULL;
int choice = 1;
while(choice)   {
      Create(&begin, &end);
      printf(" Введите: 1 – продолжить ввод, 0 – остановить");
      scanf("%d", &choice);
}
…
```

Алгоритм удаления первого элемента из очереди

Если добавление элементов происходило в конец очереди (*end), то удаление элементов происходит с начала очереди (*begin). Порядок действий при удалении элемента:

1. Устанавливаем текущий указатель на начало очереди*:*

current = begin;

2. Обрабатываем информационную часть первого элемента очереди.

3. Указатель на начало очереди переставляем на следующий (второй) элемент:

begin = begin->Next;

4. Освобождаем память, выделенную под первый элемент:

free(current);

Двунаправленный линейный список

При построении двунаправленных (двухсвязных) списков используется структура с двумя указателями: на следующий элемент и на предыдущий. Для обработки такого списка обычно используются два дополнительных указателя – на первый и последний элементы.

Схема связи элементов двунаправленного списка:

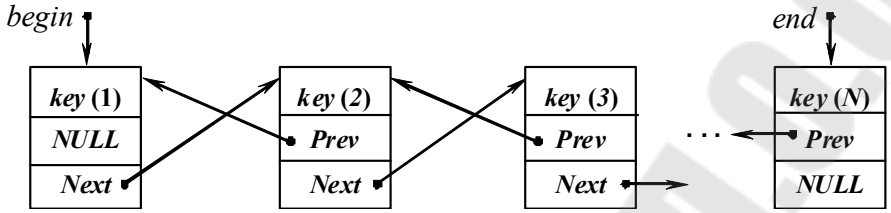

Рис. 10.5. Двунаправленный список

В структуре двунаправленного списка также выделяют информационную и адресную части:

```
struct List2    {
                int key;
                List2  *Prev, *Next;
        } ;
```

Для работы со списком декларируем *current, *begin, *end; – указатели на текущий элемент, а также на начало и конец списка. Создание двунаправленного списка проводится в два этапа – формирование первого элемента и добавление нового.

Формирование первого элемента

1. Выделение памяти под текущий элемент:

```
List2 *current = (List2*) malloc (sizeof(List2));
```

На данном этапе имеем элемент:

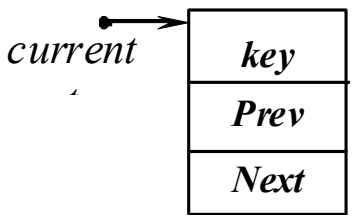

Рис. 10.6. Создание первого элемента

2. Формируем первый элемент списка:

а) формируем информационную часть:

cin >> current -> key;

б) формируем адресную часть (первоначально это *NULL*):

currentt -> Prev = current -> Next = NULL;

в) указатели начала и конца списка устанавливаем на этот элемент:

begin = end = current;

После выполнения указанных действий получили первый элемент списка (рис. 10.7).

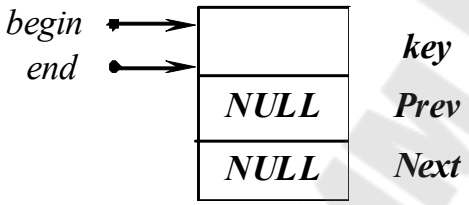

Рис. 10.7. Полученный двунаправленный список

Добавление элементов в конец списка

1. Выделение под текущий элемент:

current = (List2) malloc(sizeof(List2));*

2. Формирование информационной части:

cin >> current -> key;

3. Формирование адресных частей текущего элемента:

а) добавление выполняем в конец, следующего за current текущего элемента нет:

current -> Next = NULL;

б) указателю на предыдущий элемент *Prev* присваиваем адрес бывшего последнего элемента:

current -> Prev = end;

4. Последним элементом был *end*, а должен стать current:

end -> Next = current;

5. Переставляем указатель конца списка на созданный элемент:

end = current;

Алгоритм поиска элемента в списке по ключу

Для поиска конкретного значения myKey в списке:

1. Введем искомое значение:

cin >> myKey;

2. Установим текущий указатель на начало списка:

current = begin;

3. Начало цикла (выполнять пока *current != NULL*).

4. Сравниваем информационную часть текущего элемента с искомым:

а) если они совпадают (*current -> key = myKey*), выводим на экран элемент и завершаем поиск (*break*);

б) если иначе, переставляем текущий указатель на следующий элемент:

current = current -> Next;

Алгоритм удаления элемента в списке по ключу

1. Если найден элемент для удаления, то удаляем элемент из списка в зависимости от его местонахождения.

2. Если удаляемый элемент находится в начале списка, то создаем новый начальный элемент:

а) указатель начала списка переставляем на следующий (второй) элемент:

begin = begin -> Next;

б) указателю *Prev* элемента, который был вторым, а теперь стал первым, присваиваем значение *NULL*, т. е. предыдущего нет:

begin -> Prev = NULL;

3. Если удаляемый элемент в конце списка, то:

а) указатель конца списка переставляем на предыдущий элемент:

end = end -> Prev;

б) обнуляем указатель на следующий (*Next*) элемент нового последнего элемента:

end -> Next = NULL;

4. Если удаляемый элемент находится в середине списка, нужно обеспечить связь предыдущего и последующего элементов (рис. 10.8).

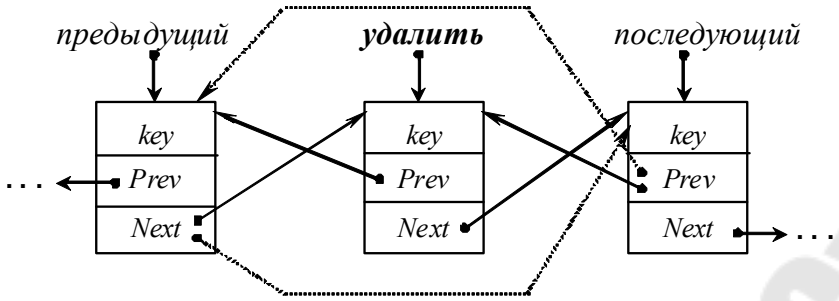

Рис. 10.8. Удаление элемента списка

Перекидываем указатели согласно схеме:

(удалить -> Prev) -> Next = удалить -> Next;
(удалить -> Next) -> Prev = удалить -> Prev;

5. Освобождаем память, занятую удаленным элементом *free* (удалить).

Глава 11. ДЕРЕВЬЯ

Деревья представляют собой одну из разновидностей нелинейных динамических структур с иерархическим представлением (рис. 11.1). Дерево в своем составе содержит один корень, множество узлов и листьев. Все дуги, соединяющие корень, узлы и листья, имеют направление.

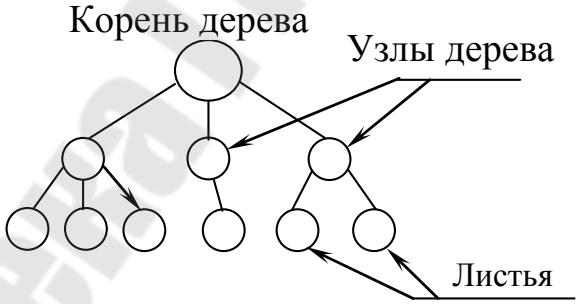

Рис. 11.1. Общее представление дерева

Корнем называется узел, у которого нет родителей (нет дуг, входящих в данный узел).

Листом называется узел, у которого есть родитель, но нет сыновей (нет дуг, выходящих из данного узла).

Узел – это такая вершина, у которой есть сыновья и родители.

Порядок узла равен количеству его сыновей.

Высота дерева – это расстояние от корня до листа, расположенного ниже всех остальных.

В структуре дерева обычно кроме полей с информационной частью содержатся указатели на сыновей данного узла (адресная часть). Если сыновья у данного узла отсутствуют, то адресная часть данного узла должна содержать значения NULL. Одной из разновидностей деревьев является бинарное (или двоичное) дерево.

Бинарное дерево – это динамическая структура данных, в которой каждый узел-родитель содержит кроме данных указатели на двух сыновей (левый и правый) (рис. 11.2).

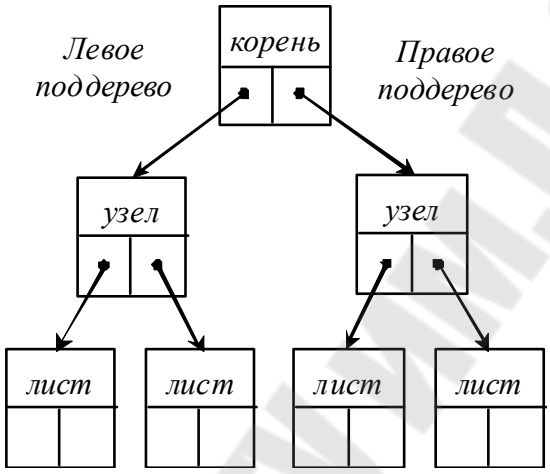

Рис. 11.2. Общий вид бинарного дерева

Бинарные деревья часто используются для поиска информации по какому-то полю (ключу). Основное правило организации такого дерева: для любого узла все ключи левого поддерева меньше ключа данного узла, а все ключи правого поддерева больше ключа данного узла. Одинаковые ключи в таких деревьях поиска не допускаются.

Сбалансированными или *AVL*-деревьями называются деревья, для которых высóты его левого и правого поддеревьев различаются не более чем на 1.

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

Рассмотрим формирование бинарного дерева на примере. Пусть имеется последовательность значений ключей 80, 86, 60, 55, 70, 99, 75, 64, 69, тогда построенное по ним дерево будет выглядеть следующим образом (рис. 11.3).

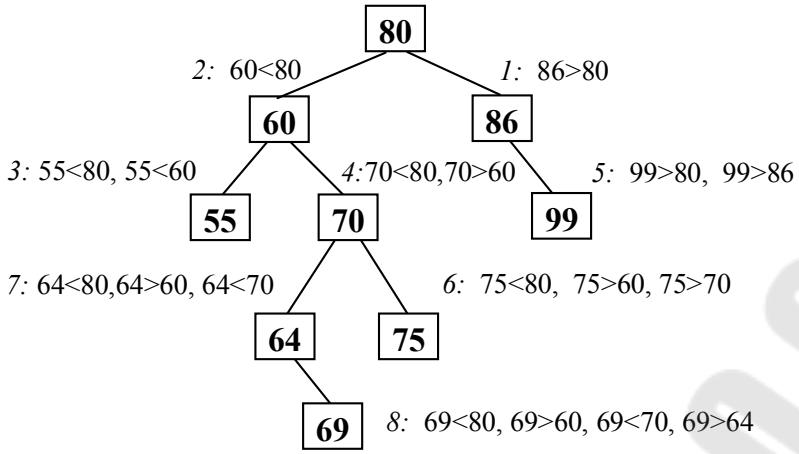

Рис. 11.3. Создание дерева

Основные операции при работе с деревьями:
– создать новый элемент дерева;
– отобразить все элементы дерева;
– выполнить поиск элемента по ключу;
– удалить заданный элемент.

Вставка нового элемента

Для того чтобы вставить новый элемент в дерево (новый элемент всегда будет листом), необходимо найти для него место. Для этого, начиная с корня, сравниваем значения узлов (*bynaryTree->key*) со значением нового элемента (*newEl*). Если *newEl* < *key*, то идем по левой ветви, в противном случае – по правой ветви. Когда дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено.

Путь поиска места в построенном дереве для числа **73** следующий (рис. 11.4).

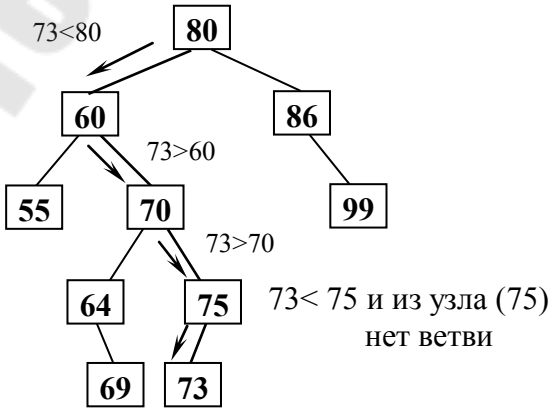

Рис. 11.4. Вставка нового элемента для дерева

Функция создания дерева, ключами которого являются целые положительные числа, представлена ниже:

```
binaryTree* MakeTree(binaryTree *treeRoot)  {
    binaryTree *prevEl, *currentEl;      // prevEl родитель текущего элемента
int newEl, find;
    if ( treeRoot == NULL )   {      // Если дерево не создано
        printf("\n Input Root :  ");
        scanf("%d", &newEl);
        treeRoot = List(newEl);          // Установили указатель на корень
    }
//================  Добавление элементов ================
    while(1) {
        printf("\n Input Info :  "); scanf("%d", &newEl);
        if (newEl<0) break;          // Признак выхода число < 0
        currentEl= treeRoot;         // Текущий указатель на корень
        find = 0;                    // Признак поиска
        while ( currentEl && ! find) {
            prevEl = currentEl;
            if( newEl == currentEl->key)
            find = 1;          // Ключи должны быть уникальны
            else
            if ( newEl < currentEl -> key ) currentEl = currentEl -> Left;
                                else  currentEl = currentEl -> Right;
        }
        if (!find) {                     // Нашли место prevEl
            currentEl = List(newEl);          // Создаем новый узел
            if ( newEl < prevEl -> key )  //  и присоединяем его, либо
            prevEl -> Left = currentEl;  // на левую ветвь,
            else   prevEl -> Right = currentEl;      // либо на правую ветвь
```

```
            }
    }
        return treeRoot;
    }
```

Функция *List* создает новый элемент – лист дерева:

```
    binaryTree* List(int i)  {
        binaryTree  *currentEl  =  (binaryTree*)  malloc
(sizeof(binaryTree));
        currentEl -> key = i;
        currentEl -> Left = currentEl -> Right = NULL;
        return currentEl;
    }
```

Участок кода с обращением к функции MakeTree будет иметь следующий вид:

```
        …
    struct binaryTree {
            int key;
            binaryTree *Left, *Right;
            };
    void main()
    {
        binaryTree *treeRoot = NULL;          // Указатель корня
        …
        treeRoot = MakeTree(treeRoot);
```

Удаление узла

При удалении узла из дерева возможны три ситуации, в зависимости от того, сколько сыновей (потомков) имеет удаляемый узел:

1. Удаляемый узел является листом – просто удаляем ссылку на него.

2. Удаляемый узел имеет только одного потомка, т. е. из удаляемого узла выходит ровно одна ветвь.

3. Удаление узла, имеющего двух сыновей, значительно сложнее рассмотренных выше. Если delKey – удаляемый узел, то его следует заменить узлом up, который содержит либо наибольший ключ (самый правый, у которого указатель *Right* равен *NULL*) в левом поддереве, либо наименьший ключ (самый левый, у которого указатель *Left* равен *NULL*) в правом поддереве.

Используя первое условие, находим узел up, который является самым правым узлом поддерева delKey, у него имеется только левый сын.

В построенном ранее дереве удалим узел Delkey (60). Используем второе условие, т. е. ищем самый левый узел в правом поддереве – это узел *w* (указатель *Left* равен *NULL*) (рис. 11.5).

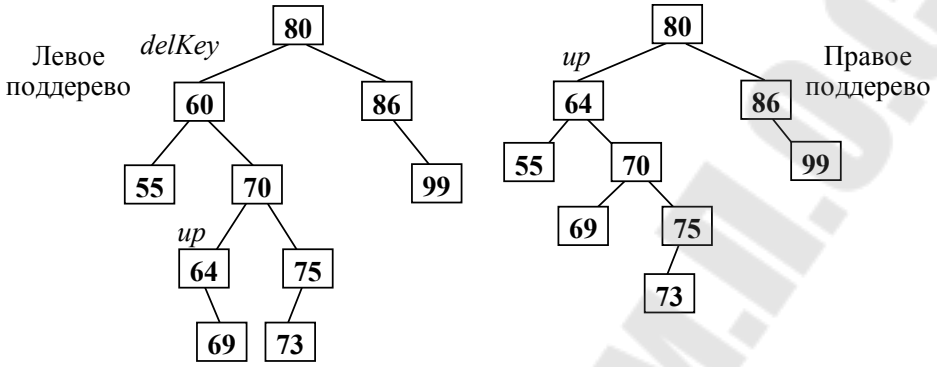

Рис. 11.5. Удаление узла дерева

Функция удаления узла по заданному ключу *delkey*

Функция удаления узла по заданному ключу *delkey*:

```
binarytree* del(binarytree *treeroot, int delkey)  {
binaryTree *delEl, *Prev_Del, *up, *Prev_R;
// delEl, Prev_Del – удаляемый элемент и его родитель;
// up, Prev_up – элемент, на который заменяется удаляемый, и его родитель;
        delEl = treeRoot;
        Prev_Del = NULL;
// ===== Поиск удаляемого элемента и его родителя по ключу delKey =====
        while (delEl != NULL && delEl -> key != delKey) {
            Prev_Del = delEl;
            if (delEl->key > delKey)  delEl = delEl->Left;
                else delEl = delEl->Right;
            }
        if (delEl == NULL)     {            // Элемент не найден
            puts("\n NO Key!");
            return treeRoot;
        }
// ============ Поиск элемента up для замены ================
```

```
        if (delEl -> Right == NULL) up = delEl->Left;
        else
                if (delEl -> Left == NULL) up = delEl->Right;
                else {
    // Ищем самый правый узел в левом поддереве
                        Prev_up = delEl;
                        up = delEl->Left;
                        while (up->Right != NULL) {
                                Prev_up = up;
                                up = up->Right;
                        }
    // Нашли элемент для замены up и его родителя Prev_up
                        if( Prev_up == delEl)
                                up->Right = delEl->Right;
                        else {
                                up->Right = delEl->Right;
                                Prev_up->Right = up->Left;
                                up->Left = Prev_up;
                        }
                }
        if (delEl== treeRoot) treeRoot = up;      // Удаляя корень, заменяем его на up
        else
    // Поддерево up присоединяем к родителю удаляемого узла
        if (delEl->key < Prev_Del->key)  Prev_Del->Left = up;    // на левую ветвь
                else   Prev_Del->Right = up;                     // на правую ветвь
        printf("\n Delete element  %d \n", delEl->key);
        free(delEl);
        return treeRoot;
    }
```

Тогда участок программы с обращением к данной функции будет иметь следующий вид:

```
        …
    printf("\n Input Del key :  ");
        scanf(“%d”, &delKey);
        treeRoot = Del(treeRoot, delKey);
        …
```

Алгоритм обхода дерева

Существуют три алгоритма обхода деревьев:

– *Left-Root-Right* (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево);

– *Left-Right-Root* (посещаем корень после поддеревьев);

– *Root-Left-Right* (посещаем корень до поддеревьев).

Для удобства просмотра дерева его часто разворачивают на 90°. Вывод дерева с использованием последнего правила можно осуществить следующей функцией:

```
void ViewTree ( binaryTree *currentEl, int depth )   {
      if ( currentEl ) {
            ViewTree ( currentEl -> Right , depth+1);      // правое поддерево
            for ( int i=0; i<depth; i++)  printf("    ");
            printf(" %d\n", currentEl -> key);
            ViewTree( currentEl -> Left , depth+1); //          левое поддерево
      }
}
```

Обращение к функции ViewTree будет иметь следующий вид:

```
ViewTree(treeRoot, 0);
```

Функция *ViewTree* рекурсивная, вторым ее параметром является переменная, определяющая, на какой глубине (*depth*) находится узел. Корень находится на глубине «0». Значения узлов выводятся по горизонтали так, что корень находится слева. Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное глубине узла.

Освобождение памяти

Для освобождения памяти, занятой элементами дерева, также понадобится функция с двойной рекурсией Del_Tree:

```
void Del_Tree(binaryTree *currentEl)  {
      if ( currentEl != NULL)  {
            Del_Tree ( currentEl -> Left);
            Del_Tree ( currentEl -> Right);
            free(currentEl);
      }
}
```

Глава 12. ПРЕПРОЦЕССОР

Препроцессором называется первая фаза компиляции, т. е. подготовка к компиляции. Директива есть инструкция препроцессора, она должна начинаться с символа #.

Директива #include<имя файла> необходима для подключения внешних файлов (обычно они имеют расширение .h). Подключаемый файл тоже может содержать директивы #include, однако не желательно в нем иметь определения для функций и данных. Если название файла записано в угловых скобках, поиск ведется в стандартных каталогах включаемых файлов. При подключении таких файлов расширение можно опускать. Если название файла записано в кавычках, тогда поиск файла ведется в текущем каталоге, а затем уже в стандартных каталогах.

Для каждого файла библиотеки С с именем <name.h> имеется соответствующий файл библиотеки С++ <cname>, в котором те же средства описываются в пространстве имен std. Например, директива #includle <cstdio> обеспечивает те же возможности, что и #include <stdio.h>, но при обращении к стандартным функциям требуется указывать имя пространства имен std.

Директива #define определяет подстановку в тексте программы.

Она используется для определения:

– символических констант;

– макросов;

– символов управляющих условной компиляцией.

Примеры:

```
#define PIVALUE 3.14
#define NAME "NAme"
#define MIN(x,y) ((x)<(y)?(x):(y))
#define UNICODE
```

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 = MIN(x, y);*, он будет заменен на *y = ((x)<(y)?(x):(y))*.

Препроцессор не оценивает вставляемый текст с точки зрения синтаксиса. Например, если к макросу *#define DIV(x) (x/x)* обратиться как *DIV(x+l)*, в результате подстановки получится выражение *(x+1/x+1)*.

Макросы и символические константы унаследованы из языка С, при написании программ на С++ их следует избегать. Вместо символических констант предпочтительнее использовать *const* или *enum*, а вместо макросов – встроенные функции или шаблоны.

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

Формат директивы *#if*:

```
# if выражение
[ #elif выражение]
[ #elif выражение]
[ #else]
#endif
```

Исключаемые блоки кода могут содержать любые другие операторы языка. Пример:

```
# if  INDEX == 1
#define FILE "N1.h"
#elif  INDEX == 2
#define FILE "N2.h"
// и так далее
#else
#define FILE "Default.h"
#endif
#include FILE
```

Можно использовать проверку, определена ли константа, с помощью defined (имя_константы).

Также директиву можно использовать, чтобы временно исключить фрагменты кода из компиляции. Иногда такой способ довольно удобен.

Кроме того, используются директивы #ifdef и #ifndef, позволяющие управлять компиляцией в зависимости от того, определена ли с помощью директивы #define указанная в них константа (хотя бы как пустая строка), например:

```
#ifdef имя
#ifndef имя
```

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

Директиву #ifndef можно использовать для обеспечения включения заголовочного файла только один раз. Однако для этого можно использовать директиву #pragma once. Директива #pragma используется для доступа к специфичным расширениям компилятора, и, соответственно, набор команд для каждого компилятора свой, уточнить наборы команд можно просмотрев документацию, однако в данный момент команда #pragma once доступна для большинства компиляторов.

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

Также препроцессор поддерживает некоторые операторы, такие, как:

– # – заключает аргумент в двойные кавычки;

– ## – соединяет две лексемы в один макрос.

Препроцессор также занимается обработкой триграфов – последовательности из трех символов, которые начинаются со знаков ??, а третий – значение. Значения представлены в табл. 12.1.

Таблица 12.1

Триграф	**??=**	**??/**	**??'**	**??(**	**??)**	**??!**	**??<**	**??>**	**??-**
Значение	#	\	^	[	]	|	{	}	~

На практике триграфы используются очень редко и были применены по причине того, что была введена таблица символов ISO/IEC 646, в которой коды данных символов значений отводились под региональные символы.

Также существуют диграфы, служащие той же цели, однако в отличие от триграфов обработка диграфов происходит после синтаксического анализа и только в том случае, если диграф есть отдельная лексема. Список диграфов можно посмотреть в табл. 12.2.

Таблица 12.2

Диграф	%:	and	bitor	or	xor	bitand	compl	and_eq	or_eq	xor_eq	not	not_eq
Значение	#	&&	|	||	^	&	~	&=	|=	^=	!	!=

Глава 13. РЕШЕНИЕ ЗАДАЧ И РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Процесс решения задачи состоит из многих этапов – раскрытие смысла задачи, разработка концептуального решения, реализация решения в виде компьютерной программы.

Решение состоит из двух компонентов: алгоритма и способов хранения данных. Алгоритмы – это пошаговое описание метода решения задачи за отрезок времени. Они часто работают со структурами данных. Например, алгоритм может вносить новые данные в структуру, удалять их, либо просматривать.

Возможно, такое описание решения формирует ложное впечатление, что вся сложность заключается лишь в разработке подходящего алгоритма, а способы хранения данных играют вспомогательную роль. Все это далеко от истины. Для решения задачи нужно не просто хранить данные, но и организовывать их таким образом, чтобы ускорить выполнение алгоритма.

Жизненный цикл программного обеспечения

Разработка хорошего программного обеспечения (ПО) должна учитывать долгий и продолжительный процесс. Этот процесс называется жизненным циклом ПО. Он начинается с первоначальной идеи, включает в себя написание и отладку программ и продолжается многие годы, в течение которых в исходное ПО вносятся изменения и улучшения.

Этап 1. Постановка задачи.

Часто люди, которые формулируют задачи, не являются программистами, поэтому исходная постановка задачи может быть неточной. Исходя из этого, на первом этапе программисты и непрограммисты совместными усилиями уточняют и детализируют исходную задачу.

Для взаимопонимания можно написать маленькие программы, которые имитируют поведение отдельных частей создаваемого ПО. Например, простая, пусть даже не эффективная, программа может демонстрировать предполагаемый пользовательский интерфейс. Лучше выявить все особые случаи, либо изменить подход к решению задачи на этом этапе, а не в процессе программирования или при эксплуатации ПО.

Этап 2. Разработка.

Эффективнее всего упростить процесс решения задачи, разбив большую задачу на несколько маленьких, которыми легче управлять.

В конечном итоге программа будет состоять из нескольких модулей, которые представляют собой самостоятельные единицы кода. Модуль может содержать как одну, так и несколько функций, а также другие блоки кода. Теоретически модули можно считать изолированными друг от друга. Каждый из них должен быть узкоспециализирован, т. е. решать только свою задачу. Стало быть, модульность – это свойство программы, которая состоит из слабо связанных и узкоспециализированных модулей. Эти модули предназначены для решения общей, точно определенной задачи. На этом этапе очень важно точно указывать не только предназначение определенных модулей, но и поток данных между этими модулями. Спецификацию можно рассматривать как договор между Вашей функцией и вызывающим ее модулем. Если Вы разрабатываете программу самостоятельно, этот договор поможет разбить исходную задачу на более мелкие части. Также он поможет разбить между командой, которая трудится над этим проектом. Программист, который разрабатывает функцию сортировки, должен выполнять спецификацию. Эта спецификация законченной функции сортировки сообщает остальным программистам, как ее вызывать и какие результаты она должна возвращать. Следует подчеркнуть, что договор не объединяет его с конкретным методом решения задачи. Не следует делать в другой части программы какие-нибудь предположения, касающиеся этого метода. Тогда, если в дальнейшем Вы будете переписывать свою функцию, вносить изменения в остальной код не потребуется совсем. Формулируя предусловие и постусловие функции, пишется ее договор, состоящий из условий, которые должны выполняться перед ее вызовом и после завершения ее работы, соответственно.

При увеличении размера программы важность хорошей документации возрастает, независимо от того, один Вы пишете программу или нет. Не рекомендуется закрывать глаза на возможность применения готовых модулей, которые решают Вашу задачу. Шансы повторного использования кода, которые предоставляются языком C++, постоянно реализуются в виде компилируемых библиотек. Библиотеки – это коллекции готовых компонентов ПО. Функцию сортировки можно применять, ничего не зная о деталях ее реализации. Помимо этого, функция может быть написана на совершенно другом языке. Окончательным результатом данного этапа должно быть модульное решение. Его легко выразить с помощью конструкций конкретного языка программирования.

Этап 3. Оценка риска.

Есть такие проблемы, которые присущи всем проектам, а есть такие, которые характерны только для определенных разработок. Некоторые можно предвидеть, а некоторые так и остаются «в тени». Данные проблемы могут влиять на график и стоимость выполнения работ, экономические успехи и даже на жизнь и здоровье людей. Кое-какие опасности можно предотвратить или смягчить, а кое-какие – нет. Для распознавания, оценки и профилактики опасностей, которые возникают при разработке ПО, существуют специальные методы. Итог оценки риска влияет на все этапы жизненного цикла ПО.

Этап 4. Верификация.

Для проверки точности алгоритмов имеются формальные методы: пред- и постусловия, представляющие собой пример простых формулировок об условиях. Эти условия выполняются в начале и в конце функции. Инвариант – это условие, которое всегда должно быть истинным в конкретной точке алгоритма. Инвариант цикла – это условие, которое должно выполняться до и после каждого выполнения цикла, являющегося частью алгоритма. Инварианты цикла на практике являются полезными для создания верных циклов. Применяя инварианты, легче выявлять ошибки. Следовательно, это сокращает время отладки и тестирования программы, т. е. инварианты экономят время. Аргументировав правильность отдельных операторов, можно доказать корректность последовательности операторов, затем функций, и в результате – всей программы. Разумеется, если в процессе верификации программы выявилась ошибка, алгоритм можно исправить, а постановку задачи изменить. Отсюда следует, что, используя инварианты, можно установить, что ошибка заключалась не в коде, а в самом алгоритме. В итоге время, израсходованное на отладку программы, значительно уменьшается. При помощи формальных методов можно установить корректность различных конструкций, а именно, операторов if, циклов и операторов присваивания. Кодирование – это сравнительно небольшая часть жизненного цикла ПО.

Этап 5. Кодирование.

Кодирование заключается в переводе алгоритма на определенный язык программирования с вытекающим корректированием синтаксических ошибок. Возможно, конкретно кодирование многие считают программированием. И все же необходимо понимать, что кодирование – это не самое важное, это лишь один из этапов жизненного цикла ПО.

Этап 6. Тестирование.

На стадии тестирования полезно выявить и внести исправления в наибольшее количество логических ошибок. Для этого разрешено воспользоваться проверкой конкретных функций, используя их к входным данным и сопоставляя их с заранее установленным итогом. Если входные данные меняются в каком-то определенном диапазоне, непременно рассмотрите их крайние значения. К примеру, если входное значение n может варьироваться от 1 до 10, невзирая ни на что, проверьте программу при значениях 1 и 10. Дополнительно необходимо протестировать, как функционирует программа, если в нее внести явно ложные данные, и может ли она определять такие ошибки. Попытайтесь ввести в программу рандомно взятые данные, а затем используйте ее для практического набора данных. Тестирование – это и наука, и искусство вместе.

Этап 7. Уточнение решения.

Итогом исполнения этапов 1–6 оказывается работающая программа, которую усиленно тестировали и отлаживали.

Предпочтительнее решать задачу при более простых предположениях, потихоньку затрудняя программу. К примеру, скажем, что входные данные имеют конкретный формат и оказываются правильными. Выполнив самый простой вариант, можно пополнять его более сложными процедурами ввода и вывода данных, снабжать специальными возможностями и средствами для нахождения ошибок.

Так, если используется подход «от простого – к сложному», этап уточнения решения оказывается необходимым. Определенное уточнение решения оказывается достаточно очевидным, особенно, если программа имеет модульную структуру. На самом деле поэтапное уточнение решения является основным преимуществом модульного подхода к разработке программ. К тому же после каждого изменения программы ее нужно опять детально протестировать. Этапы жизненного цикла ПО не оторваны друг от друга и не следуют один за другим. Выполнив реалистичные упрощающие предположения в самом начале процесса разработки программы, необходимо учесть их в дальнейшем. Тестирование программы может вынудить внести в программу изменения, однако модифицированную программу придется снова тестировать.

Этап 8. Производство.

После окончания разработки программного продукта он распространяется среди пользователей, инсталлируется на их компьютерах и используется.

Этап 9. Сопровождение.

Пользователи программ могут найти ошибки, которые остались незамеченными при тестировании. Кроме того, со временем ПО нужно совершенствовать, внедряя в него новые функциональные возможности или изменяя его компоненты. Создатели программ занимаются этим редко, тем важнее является наличие хорошей документации.

Этапы 1–7 – это компоненты процесса решения задачи. Действуя по этой стратегии, в начале необходимо разработать и реализовать решение (этапы 1–6), основываясь на некоторых начальных упрощающих предположениях. В итоге получите хорошо организованную программу, которая решает несколько упрощенную задачу. На последнем этапе эта программа усложняется и обязана полностью соответствовать начальной постановке задачи.

Глава 14. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В С++

Понятие класса и объекта

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 С для этого достаточно одного оператора.

С ростом объема программы становится невозможным удерживать в памяти все детали и необходимым структурировать информацию, выделять главное и отбрасывать несущественное.

Описание собственных типов данных, позволяющих структурировать и группировать информацию, представляя ее в более естественном виде, является одним из способов достижения этой цели. 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. Можно 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 заголовки. 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.

Описанные выше методы преследуют цель упростить структуру программы, т. е. 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, используются только через его заголовок – 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 пользовате*лем. 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. Конкретные величины классов называются экземплярами класса, или объектами.

Класс в C++ определяется как:

```
class <имя класса>{
        private:
        <сокрытые члены>
        public:
        <открытые члены>
};
```

Идея классов является основой объектно-ориентированного программирования (ООП).

Основными свойствами ООП являются:

– инкапсуляция – объединение данных с функциями их обработки с сокрытием ненужной для использования этих данных информации;

– наследование – возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять или создавать новые;

– 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 конструктора и деструктора

Конструктор – особый метод, вызываемый при создании объекта для его инициализации.

Конструкторы обладают следующими свойствами:

– конструктор не возвращает никакого значения;

– класс может иметь несколько конструкторов с разными параметрами;

– параметром конструктора не может быть тип, определяемый этим классом;

– если не написать конструктор, будет создан конструктор по умолчанию – сильно урезанная форма конструктора;

– конструкторы не наследуются.

Конструктор вызывается при упоминании объекта, вне зависимости от того, в каком виде, кроме создания через оператор new (потому что в этом случае создается указатель на объект, а не сам объект).

Конструктор описывается как метод с именем, совпадающим с именем класса:

```
class example{
        public:
        example(int exampleValue); //конструктор
}
```

Деструктор – аналогичный конструктору метод, вызываемый во время уничтожения объекта для освобождения памяти, однако в отличие от конструктора не принимают параметров.

Описывается так же как и конструкторы: со знаком ~ в начале:

```
class example{
        public:
        example(int exampleValue); //конструктор
        ~example(); //деструктор
}
```

Стоит заметить, что деструктор необходим только в том случае, если объект использует динамическое выделение памяти, в противном случае память не будет помечена как свободная, и возникнет «утечка памяти».

Дружественные функции

Дружественные функции применяются для доступа к скрытым нолям класса и представляют собой альтернативу методам. Метод, как правило, используется для реализации свойств объекта, дружественная функция представляет собой действие, концептуально входящее в класс и нуждающиеся в доступе к его скрытым полям, например, переопределенные операции вывода объектов.

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 не имеет значения.

– одна функция может быть дружественной сразу нескольким классам.

Примечание: дружественные функции нарушают принцип инкапсуляции, а также затрудняют отладку и модификацию программы, поэтому на практике применяются очень редко.

Глава 15. РЕАЛИЗАЦИЯ НАСЛЕДОВАНИЯ В ЯЗЫКЕ С++

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, т. е. 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.

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, например:

```
class parent1 { ... };
class parent2 { ... };
class parent3 { ... };
class child: parent1, parent2, public parent3 { ... };
```

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

Разница между спецификаторами доступа определена в табл. 15.1.

Таблица 15.1

Ключ доступа	**Спецификатор в базовом классе**	**Доступ в производном классе**
private	private	нет
	protected	private
	public	private
protected	private	нет
	protected	protected
	public	protected
public	private	нет
	protected	protected
	public	public

Глава 16. ПОЛИМОРФИЗМ

Полиморфизм можно также определить как способность объектов-родственников по-разному осуществлять однотипные действия, т. е. однотипные действия у множества классов. Осуществляется при помощи виртуальных и абстрактных функций.

Виртуальные и абстрактные функции

Для определения виртуальной функции используется спецификатор virtual, например:

virtual void outputInt(int item);

Правила описания и использования виртуальных функций следующие:

1. Если в базовом классе функция определена как виртуальная, функция, определенная в производном классе с тем же именем и набором параметров, автоматически становится виртуальной, а с отличающимся набором параметров – обычной.

2. Виртуальные функции стоит переопределять только при необходимости задать отличающиеся действия. Права доступа изменить нельзя.

3. Если виртуальная функция переопределена в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости.

4. Виртуальная функция не может объявляться с модификатором static, но может быть объявлена как дружественная.

5. Если в классе вводится описание виртуальной функции, она должна быть определена хотя бы как абстрактная.

Абстрактная функция содержит признак =0 вместо тела, например:

virtual void f(int x) = 0;

Абстрактная функция должна переопределяться в производном классе (возможно, опять как абстрактная).

Правила, по которому функцию следует делать виртуальной, не существует. Можно только дать рекомендацию объявлять виртуальными функции, для которых есть вероятность, что они будут переопределены в производных классах. Функции, 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 функции виртуальной обеспечивает гибкость и возможность расширения.

Для пояснения представим себе, что вызов функции draw осуществляется из функции перемещения объекта. Если текст функции перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех объектов одинаков, а для отрисовки вызывается конкретная функция), переопределять эту функцию в производных классах нет необходимости, и она может быть описана как невиртуальная. Если функция draw виртуальная, функция 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 функции, 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.

Глава 17. ОСНОВЫ ПРОГРАММИРОВАНИЯ ПОД WINDOWS

Окно (window) – один из наиболее важных объектов в операционной системе Windows. Окно представляет собой прямоугольную область, в которой приложение отображает выводимую информацию и из которой получает вводимую. Для любого окна Windows определяет дескриптор (handle), который идентифицирует каждое окно.

Класс окна (Window Class) определяет общие свойства и особенности поведения группы окон, созданных на основе класса окна. С помощью класса окна определяются такие характеристики, как пиктограмма, заголовок, кисть для закрашивания, стиль окна и т. д. Одно из полей класса связывает окно с его оконной процедурой, которое обрабатывает сообщения. После прихода сообщения для заданного окна оконная процедура ищет обработчик данного сообщения или использует стандартный Windows-обработчик. Классы окон создаются и регистрируются

функциями RegisterClass() и RegisterClassEx(). Создание окон любых видов обеспечивается функциями CreateWindow() и CreateWindowEx().

Оконная процедура (Window Procedure или WndProc) является получателем всех сообщений, поступающих окну. Данная процедура имеет спецификатор CALLBACK, который означает, что вызываться данная функция будет напрямую операционной системой. Все сообщения становятся в очередь и выбираются с помощью функции GetMessage. Чаще всего оконная процедура представляет собой конструкцию вида switch–case, распознающую поступившие сообщения и выполняющую их обработку.

Сообщения (Window Message) лежат в основе механизмов событийного управления приложениями. Они представляют собой структуры, передаваемые между приложениями и несущими информацию о событиях, команды, данные и т. д. Для представления сообщений служит структура MSG, поля которой содержат:

typedef struct tagMSG {
HWND hwnd; – дескриптор окна, которому адресовано сообщение
UINT message; – идентификатор сообщения
WPARAM wparam; – дополнительная информация
LPARAM lparam; – дополнительная информация
DWORD time; – время отправки сообщения
POINT pt; – экранные координаты курсора мыши в момент отправки сообщения
} MSG;

Минимальное WinAPI-приложение должно содержать как минимум две функции:

– WinMain – главную функцию, в которой создается основное окно программы и запускается цикл обработки сообщений;

– WndProc – оконную процедуру, обеспечивающую обработку сообщений для основного окна программы.

WinMain является точкой входа в программу и выполняет следующие действия:

– определение класса окна;
– регистрация класса окна;
– создание окна;
– отображение окна;
– запуск цикла обработки сообщений.

Пример такого приложения приведен ниже:

```
#include <windows.h>
#include <tchar.h>
LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM,
LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");
INT WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE
hPrevInst,
LPTSTR lpszCmdLine, int nCmdShow)
{
HWND hWnd;
MSG Msg;
WNDCLASSEX wcl;
wcl.cbSize = sizeof (wcl);
wcl.style = CS_HREDRAW | CS_VREDRAW;
wcl.lpfnWndProc = WindowProc;
wcl.cbClsExtra = 0;
wcl.cbWndExtra = 0;
wcl.hInstance = hInst;
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
wcl.hbrBackground = (HBRUSH) GetStockOb-
ject(WHITE_BRUSH);
wcl.lpszMenuName = NULL;
wcl.lpszClassName = szClassWindow;
wcl.hIconSm = NULL;

if (!RegisterClassEx(&wcl))
return 0;
hWnd=CreateWindowEx(0,
szClassWindow,
TEXT("Каркас Windows приложения"),
WS_OVERLAPPEDWINDOW,

CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
CW_USEDEFAULT,
NULL,
```

```
NULL,
hInst,
NULL);

ShowWindow(hWnd, nCmdShow);

UpdateWindow(hWnd);

while(GetMessage(&Msg, NULL, 0, 0))
{
TranslateMessage(&Msg);
DispatchMessage(&Msg);
}
return Msg.wParam;
}
LRESULT CALLBACK WindowProc(HWND hWnd, UINT uMes-
sage, WPARAM wParam,
LPARAM lParam)
{
switch(uMessage)
{
case WM_DESTROY;
PostQuitMessage(0);
break;
default:
return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

Глава 18. ОБРАБОТКА СОБЫТИЙ

При возникновении любого события операционная система генерирует некоторое сообщение, которое будет обрабатываться оконной процедурой активного окна. Например, когда WinMain вызывает функцию CreateWindow, Windows создает окно и отправляет оконной процедуре сообщение WM_CREATE. Когда WinMain вызывает ShowWindow, Windows отправляет оконной процедуре сообщения WM_SIZE и WM_SHOWWINDOW. Когда WinMain вызывает UpdateWindow, Windows отправляет оконной процедуре сообщение WM_PAINT (перерисовка клиентской части окна).

При завершении работы приложения генерируется сообщение о выходе из программы (WM_QUIT).

При обработке событий клавиатуры генерируются сообщения WM_KEYDOWN (клавиша нажимается) и сообщение WM_KEYUP (клавиша отпускается). В сообщении WM_KEYDOWN содержится также информация о скан-коде нажатой клавиши. Функция API TranslateMessage преобразует пару сообщений WM_KEYDOWN и WM_KEYUP в символьное сообщение WM_CHAR, которое содержит ASCII-код символа (wParam). Сообщение WM_CHAR помещается в очередь, а на следующей стадии цикла функция GetMessage извлекает его для последующей обработки.

При обработке событий мыши генерируются следующие сообщения:

– WM_MOUSEMOVE – перемещение мыши по клиентской области окна;

– WM_LBUTTONDOWN – нажата левая кнопка мыши;

– WM_MBUTTONDOWN – нажата средняя кнопка мыши;

– WM_RBUTTONDOWN – нажата правая кнопка мыши;

– WM_LBUTTONUP – отпущена левая кнопка мыши;

– WM_MBUTTONUP – отпущена средняя кнопка мыши;

– WM_RBUTTONUP – отпущена правая кнопка мыши;

– WM_LBUTTONDBLCLK – двойной щелчок левой кнопкой мыши;

– WM_MBUTTONDBLCLK – двойной щелчок средней кнопкой мыши;

– WM_MOUSEWHEEL – прокрутка колесика мыши;

– WM_RBUTTONDBLCLK – двойной щелчок правой кнопкой мыши.

Для всех этих сообщений значение параметра lParam содержит положение мыши. При этом в младшем слове (младшие 2 байта) находится значение координаты X, а в старшем слове (старшие 2 байта) – значение координаты Y. Отсчет координат ведется от левого верхнего угла клиентской области окна. Эти значения можно извлечь из lParam при помощи макросов LOWORD и HIWORD.

Глава 19. ГРАФИЧЕСКИЕ ПРИЛОЖЕНИЯ WINDOWS

У программ, написанных для Windows, нет прямого доступа к аппаратной части таких устройств отображения информации, как, например, экран и принтер. Вместо этого они вызывают функции гра-

фической подсистемы WinAPI, являющейся графическим интерфейсом устройства (Graphics Device Interface, GDI). Функции GDI реализуют основные графические команды при обращении к программным драйверам соответствующих аппаратных устройств. Одна и та же команда (например, LineTo – нарисовать линию) может иметь различную реализацию в разных драйверах. Эта реализация скрыта от программиста, использующего WinAPI, что упрощает разработку приложений.

GDI (Graphics Device Interface) представляет собой единый унифицированный интерфейс устройств (средств) отображения графической информации в Windows. Работа GDI базируется на понятии контекста устройства (device context – DC), который абстрагирует свойства реальных устройств: экран (окно на экране), принтер, битовый образ в памяти и т. д. Контекст идентифицируется его описателем, тип HDC (handle DC).

Для получения контекста служат следующие функции: GetDC(), GetWindowDC(), GetDCEx(). Они применимы для оконных (экранных) контекстов. Освобождение контекстов выполняется функциями ReleaseDC() для оконных и DeleteDC() для остальных.

Для формирования изображения в контексте служат функции графических примитивов (например, LineTo(), DrawText() и т. д.) и графические инструменты. Основными объектами являются: «перо» (Pen), «кисть» (Brush) и «шрифт» (Font).

Для создания инструментов служат соответствующие функции GDI API, например, CreatePen(), CreateBrush() и т. д. Для сложных объектов может быть определено несколько функций, различающихся параметрами и получаемым эффектом.

Общая схема создания изображения следующая:

– получение контекста;

– установка набора инструментов;

– формирование изображения из примитивов;

– освобождение контекста.

В типичном случае перерисовка содержимого окна инициируется сообщением WM_PAINT.

Установка таймера

Использование таймера является хорошим способом время от времени «будить» программу. Для установки таймера необходимо использовать функцию API SetTimer:

```
UINT SetTimer(
HWND hwnd, UINT nID,
UINT wLength,
TIMEPROC lpTFunc
);
```

Если значение lpTFunc равно NULL, то для обработки сообщений таймера будет вызываться оконная процедура главного окна приложения. В этом случае каждый раз по истечении заданного временного интервала в очередь сообщений программы будет помещаться сообщение WM_TIMER, а оконная процедура программы должна будет обрабатывать его так же как и остальные сообщения. Функция SetTimer в случае успешного завершения возвращает значение идентификатора таймера, в противном случае возвращается 0.

При поступлении сообщения WM_TIMER параметр wParam содержит идентификатор таймера (приложение может установить несколько таймеров), а lParam – адрес функции таймера

Будучи установленным, таймер станет посылать сообщения до тех пор, пока программа не завершится или не вызовет функцию API KillTimer:

```
BOOL KillTimer(
HWND hwnd,
UINT nID
);
```

Содержание

Учебное электронное издание комбинированного распространения

Учебное издание

ПРОГРАММИРОВАНИЕ

Пособие
по одноименной дисциплине для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Составитель **Косинов** Геннадий Петрович

Электронный аналог печатного издания

Редактор *Т. Н. Мисюрова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 04.04.19.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 4,18. Уч.-изд. л. 4,50.
Изд. № 21.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель