МИНИСТЕРСТВО ПО РАЗВИТИЮ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ФАКУЛЬТЕТ КОМПЬЮТЕР ИНЖИНИРИНГ

КАФЕДРА «ОСНОВЫ ИНФОРМАТИКИ»

СБОРНИК ЛЕКЦИЙ ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ на С/С++»

Ташкент-2016

ЛЕКЦИЯ № 1

Тема: Строки и операции над строками **(2ч.)**

Цель занятия. *Ознакомить с понятием строка, совершением операций над строками, удаление, слияние, вставка в строку, функциями обработки строк.*

Ключевые слова. Строка, одинарные кавычки, операции над строками, прототип функции, описание функции.

План лекции:

1. Строки
2.Операции над строками.
3.Функции обработки строк.

1. Строки

Строка — последовательность (массив) символов. Если в выражении встречается одиночный символ, он должен быть заключен в *одинарные кавычки*. При использовании в выражениях строка заключается в *двойные кавычки.* Признаком конца строки является нулевой символ **\0**. В C++ строки можно описать с помощью массива символов (массив элементов типа **char**), в котором следует предусмотреть место для хранения признака конца строки.

Например, описание строки из 25 символов должно выглядеть так:

```
1 char s[25];
```

Здесь элемент **s[24]** предназначен для хранения символа конца строки.

```
1 char s[7] = "Привет";
```

Можно описать и массив строк:

```
1 char s[3][25] = {"Пример", "использования", "строк"};
```

Определен массив из 3 строк по 25 байт в каждой.

Для работы с указателями можно использовать (**char ***). Адрес первого символа будет начальным значением указателя.

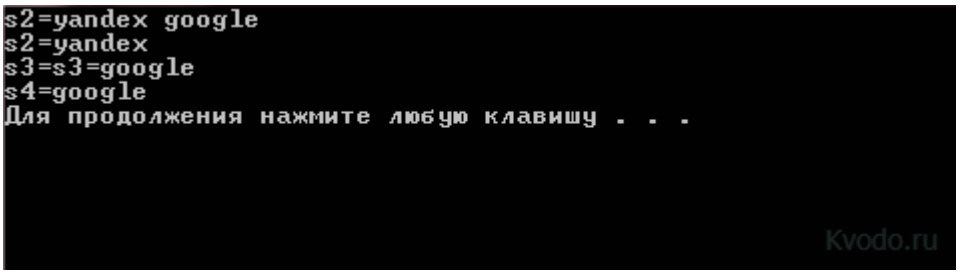
Рассмотрим пример объявления и вывода строк.

```
1 #include "stdafx.h"
2 #include <iostream>
3 using namespace std;
4 int main()
```

```
{
setlocale(LC_ALL,"Rus");
//описываем 3 строки, s3- указатель
char s2[20], *s3, s4[30];
cout<<"s2="; cin>>s2; //ввод строки s2
cout<<"s2="<<s2<<endl;
//запись в s3 адреса строки, где хранится s4. Теперь в переменных
//(указателях) s3 и s4 хранится значение одного и того же адреса
s3=s4;
cout<<"s3="; cin>>s3; //ввод строки s3
//вывод на экран строк s3 и s4, хотя в результате присваивния s3=s4;
//теперь s3 и s4 - это одно и тоже
cout<<"s3="<<s3<<endl;
cout<<"s4="<<s4<<endl;
system("pause");
return 0;
}
```

Результат работы программы:

Но следует отметить, что если пользователь введет в одну переменную слова разделенные пробелом, то программа будет работать иначе:



Все дело в том, что функция **cin** вводит строки до встретившегося пробела. Более универсальной функцией является **getline**.

cin.getline(char *s, int n);

Функция предназначена для ввода с клавиатуры строки **s** с пробелами, в строке не должно быть более **n** символов. Следовательно, для корректного ввода строк, содержащих пробел, необходимо в нашей программе заменить **cin>>s** на**cin.getline(s, 80)**.

2.Операции над строками

Строку можно обрабатывать как массив символов, используя алгоритмы обработки массивов или с помощью специальных функций обработки строк, некоторые из которых приведены ниже. Для работы с этими строками необходимо подключить библиотеку **cstring**.

Для преобразования числа в строку можно воспользоваться функцией **sprintf** из библиотеки **stdio.h**.

Некоторые функции работы со строками:

ссчс	**Описание функции**
size_t strlen(const char *s)	вычисляет длину строки s в байтах.
char *strcat(char *dest, const char *scr)	присоединяет строку src в конец строки dest, полученная срока возвращается в качестве результата
char *strcpy(char *dest, const char *scr)	копирует строку scr в место памяти, на которое указывает dest
char strncat(char *dest, const char *dest, size_t maxlen)	присоединяет строку maxlen символов строки src в конец строки dest
char *strncpy(char *dest, const char *scr, size_t maxlen)	копирует maxlen символов строки src в место памяти, на которое указывает dest
int ctrcmp(const char *s1, const char *s2)	сравнивает две строки в лексикографическом порядке с учетом различия прописных и строчных букв, функция возвращает 0, если строки совпадают, возвращает — 1, если s1 располагается в упорядоченном по алфавиту порядке раньше, чем s2, и 1 — в противоположном случае.
int strncmp(const char *s1, const char *s2, size_t maxlen)	сравнивает maxlen символов двух строк в лексикографическом порядке, функция возвращает 0, если строки совпадают, возвращает — 1, если s1 располагается в упорядоченном по алфавиту порядке раньше, чем s2, и 1 — в противоположном случае.
double atof(const char *s)	преобразует строку в вещественное число, в случае неудачного преобразования возвращается число 0
long atol(const char *s)	преобразует строку в длинное целое число, в случае неудачного преобразования возвращается 0
char *strchr(const char *s, int c);	возвращает указатель на первое вхождение символа **c** в строку, на которую указывает **s**. Если символ **c** не найден, возвращается NULL

char *strupr(char *s)	преобразует символы строки, на которую указывает s, в символы верхнего регистра, после чего возвращает ее

3.Функции обработки строк

Ниже приведены Функции обработки строк в языке Си++

char *strcpy(char *s1,const char *s2) ;

Копирует строку s2 в массив символов s1. Возвращает значение s1.

char *strncpy(char *s1,const char *s2,size_t n) ;

Копирует не более n символов из строки s2 в массив символов s1. Возвращает значение s1.

char *strcat(char *s1,const char *s2);

Добавляет строку s2 к строке s1. Первый символ строки s2 записывается поверх завершающего нулевого символа строки s1. Возвращает значение s1.

char *strncat(char *s1,const char *s2, size_t n) ;

Добавляет не более n символов строки s2 в строку s1. Первый символ строки s2 записывается поверх завершающего нулевого символа строки s1. Возвращаетзначение1.

int strcmp(const char *s1,const char *s2) ;

Сравнивает строки s1 и s2. Функция возвращает 0, если строки равны; значение меньше 0, если s1 меньше s2 и значение больше 0, если s1 больше s2.

int strncmp(const char *s1,const char *s2, size_t n);

Сравнивает до n символов строк s1 и s2. Функция возвращает 0, если строки равны; значение меньше 0, если s1 меньше s2 и значение больше 0, если s1 больше s2.

char *strtok(char *s1,const char *s2) ;

Последовательность вызовов strtok разбивает строку s1 на лексемы – логические части, такие как слова, разделенные символами, содержащимися в строке s2. Первый вызов содержит в качестве первого аргумента s1, а последующие вызовы для той же строки, содержат в качестве первого аргумента null. При каждом вызове

возвращается указатель на текущую лексему. Если лексем больше нет возвращается null.

size_t strlen(const char *s) ;

Определяет длину строки s. Возвращает количество символов, предшествующих завершающему нулевому символу.

char *strchr(const char *s,int c) ;

Находит позицию первого вхождения символа c в строку s. Если c найден, функция возвращает указатель на c в строке s, иначе возвращается NULL.

size_t strcspn(const char *s1, const char *s2) ;

Определяет и возвращает длину начального сегмента строки s1, содержащего только те символы, которые не входят в s2.

char * strpbrk(const char *s1, const char *s2);

Находит в строке s1 позицию первого вхождения любого из символов строки s2. Если символ из строки найден, возвращается указатель на этот символов строке s1, иначе возвращается NULL.

char * strrchr(const char *s, int c) ;

Находит позицию последнего вхождения символа c в строку s. Если c найден, функция возвращает указатель на этот символ, иначе возвращается NULL.

char * strstr(const char *s1, const char *s2) ;

Находит позицию первого вхождения строки s2 в строку s1. Если подстрока найдена, функция возвращает указатель подстроки в строке s1, иначе возвращается NULL.

ЛЕКЦИЯ № 2

Тема: Структуры и объединения. Структуры и объединения. Битовые поля. **(2ч.)**

Цель занятия. *Ознакомится с понятием структура, член структуры, из каких переменных состоит объединение, битовые поля структур.*

Ключевые слова. Структура, член структуры, имя структуры, объединения, анонимные объединения, битовое поле

План лекции:

1.Структура.
2.Объединения.
3 Битовые поля.
4.Контрольные вопросы.

1. Структура - это группа связанных переменных.

В C++ структура представляет собой коллекцию объединенных общим именем переменных, которая обеспечивает удобное средство хранения родственных данных в одном месте. Структуры - этосовокупные типы данных, поскольку они состоят из нескольких различных, но логически связанных переменных. По тем же причинам их иногда называют составными или конгломератными типами данных.

Прежде чем будет создан объект структуры, должен быть определен ее формат. Это делается посредством объявления структуры. Объявление структуры позволяет понять, переменные какого типа она содержит. Переменные, составляющие структуру, называются еечленами. Члены структуры также называют элементами или полями.

Член структуры - это переменная, которая является частью структуры.

В общем случае все члены структуры должны быть логически связаны друг с другом. Например, структуры обычно используются для хранения такой информации, как почтовые адреса, таблицы имен компилятора, элементы карточного каталога и т.п. Безусловно, отношения между членами структуры совершенно субъективны и определяются программистом. Компилятор "ничего о них не знает" (или "не хочет знать").

Начнем рассмотрение структуры с примера. Определим структуру, которая может содержать информацию о товарах, хранимых на складе компании. Одна запись инвентарной ведомости обычно состоит из нескольких данных, например: наименование товара, стоимость и количество, имеющееся в наличии. Поэтому для управления подобной информацией удобно использовать именно структуру. В следующем фрагменте кода объявляется структура, которая определяет следующие элементы: наименование товара, стоимость, розничная цена,имеющееся в наличии количество и время до пополнения запасов. Этих данных часто вполне достаточно для управления складским хозяйством. О начале объявления структуры компилятору сообщает ключевое словоstruct.

```
struct inv_type {
char item[40]; // наименование товара
```

```
double cost; // стоимость
double retail; // розничная цена
int on_hand; // имеющееся в наличии количество
int lead_time; // число дней до пополнения запасов
};
```

Имя структуры - это ее спецификатор типа.

Обратите внимание на то, что объявление завершается точкой с запятой. Дело в том, что объявление структуры представляет собой инструкцию. Именем типа структуры здесь является inv_type. Другими словами, имя inv_typeидентифицирует конкретную структуру данных и является ее спецификатором типа.

В предыдущем объявлении в действительности не было создано ни одной переменной. Был определен лишь формат данных. Чтобы с помощью этой структуры объявить реальную переменную (т.е. физический объект), нужно записать инструкцию, подобную следующей.

```
inv_type inv_var;
```

Вот теперь объявляется структурная переменная типа inv_type с именем inv_var. Помните: определяя структуру, вы определяете новый тип данных, но он не будет реализован до тех пор, пока вы не объявите переменную того типа, который уже реально существует.

При объявлении структурной переменной C++ автоматически выделит объем памяти, достаточный для хранения всех членов структуры. На рис. 10.1 показано, как переменная inv_var будет размещена в памяти компьютера (в предположении, что double-значение занимает 8 байт, а int-значение - 4).

Одновременно с определением структуры можно объявить одну или несколько переменных, как показано в этом примере.

```
struct inv_type {
char item[40]; // наименование товара
double cost; // стоимость
double retail; // розничная цена
int on_hand; // имеющееся в наличии количество
int lead_time; // число дней до пополнения запасов
} inv_varA, inv_varB, inv_varC;
```

Этот фрагмент кода определяет структурный тип inv_type и объявляет переменные inv_varA, inv_varB и inv_varC этого типа. Важно понимать, что каждая структурная переменная содержит собственные копии членов структуры. Например, поле cost структуры inv_varA изолировано от поля cost структуры inv_varB. Следовательно, изменения, вносимые в одно поле, никак не влияют на содержимое другого поля.

Если для программы достаточно только одной структурной переменной, в ее определение необязательно включать имя структурного типа. Рассмотрим следующий пример:

```
struct {
char item[40]; // наименование товара
double cost; // стоимость
```

```
double retail; // розничная цена
int on_hand; // имеющееся в наличии количество
int lead_time; // число дней до пополнения запасов
} temp;
```
Этот фрагмент кода объявляет одну переменную temp в соответствии с предваряющим ее определением структуры.
Ключевое слово struct означает начало объявления структуры.
Общий формат объявления структуры выглядит так.
```
struct имя_типа_структуры {
тип имя_элемента1;
тип имя_элемента2;
тип имя_элемента3;
тип имя_элементаN;
} структурные_переменные;
```
2.Объединения
Объединение состоит из нескольких переменных, которые разделяют одну и ту же область памяти.
Объединение состоит из нескольких переменных, которые разделяют одну и ту же область памяти. Следовательно, объединение обеспечивает возможность интерпретации одной и той же конфигурации битов двумя (или более) различными способами. Объявление объединения, как нетрудно убедиться на следующем примере, подобно объявлению структуры.
```
union utype {
short int i;
char ch;
};
```
Объявление объединения начинается с ключевого слова union.
Здесь объявляется объединение, в котором значение типа short int и значение типа char разделяют одну и ту же область памяти. Необходимо сразу же прояснить один момент: невозможно сделать так, чтобы это объединение хранило и целочисленное значение, и символ одновременно, поскольку переменные i и ch накладываются (в памяти) друг на друга. Но программа в любой момент может обрабатывать информацию, содержащуюся в этом объединении, как целочисленное значение или как символ. Следовательно, объединение обеспечивает два (или больше) способа представления одной и той же порции данных. Как видно из этого примера, объединение объявляется с помощью ключевого слова union.
Как и при использовании структур, при объявлении объединения не определяется ни одна переменная. Переменную можно объявить, разместив ее имя в конце объявления либо воспользовавшись отдельной инструкцией объявления. Чтобы объявить переменную объединения именем u_var типа utype, достаточно записать следующее:
```
utype u_var;
```
В переменной объединения u_var как переменная i типа short int, так и символьная переменная ch разделяют одну и ту же область памяти.

(Безусловно, переменная i занимает два байта, а символьная переменная ch использует только один.) Как переменные i и chразделяют одну область памяти, показано на рис. 10.2.

При объявлении объединения компилятор автоматически выделяет область памяти, достаточную для хранения в объединении переменных самого большого по объему типа.
Чтобы получить доступ к элементу объединения, используйте тот же синтаксис, который применяется и для структур: операторы "точка" и "стрелка". При непосредственном обращении к объединению (или посредством ссылки) используется оператор "точка". Если же доступ к переменной объединения осуществляется через указатель, используется оператор "стрелка". Например, чтобы присвоить букву'A' элементу ch объединения u_var, достаточно использовать такую запись.

```
u_var.ch = 'A';
```

В следующем примере функции передается указатель на объединение u_var. В теле этой функции с помощью указателя переменной i присваивается значение 10.

```
// ...
func1(&u_var); // Передаем функции func1() указатель  на объединение u_var.
// ...
}
void fund (utype *un)
{
un->i = 10; /* Присваиваем число 10 члену объединения u_var с помощью указателя. */
}
```

Поскольку объединения позволяют вашей программе интерпретировать одни и те же данные по-разному, они часто используются в случаях, когда требуется необычное преобразование типов. Например, следующая программа использует объединение для перестановки двух байтов, которые составляют короткое целочисленное значение. Здесь для отображения содержимого целочисленных переменных используется функция disp_binary(), разработанная в главе 9. (Эта программа написана в предположении, что короткие целочисленные значения имеют длину два байта.)

```
// Использование объединения для перестановки двух байтов в рамках короткого целочисленного значения.
#include <iostream>
using namespace std;
void disp_binary(unsigned u);
union swap_bytes {
short int num;
char ch[2];
```

```
};
int main()
{
swap_bytes sb;
char temp;
sb.num = 15; // двоичный код: 0000 0000 0000 1111
cout << "Исходные байты: ";
disp_binary(sb.ch[1]);
cout << " ";
disp_binary(sb.ch[0]);
cout << "\n\n";
// Обмен байтов.
temp = sb.ch[0];
sb.ch[0] = sb.ch[1];
sb.ch[1] = temp;
cout << "Байты после перестановки: ";
disp_binary(sb.ch[1]);
cout << " ";
disp_binary(sb.ch[0]);
cout << "\n\n";
return 0;
}
// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
register int t;
for(t=128; t>0; t=t/2)
if(u & t) cout << "1 ";
else cout << "0 ";
}
```

При выполнении программа генерирует такие результаты.

Исходные байты: 0000 0000 0000 1111

Байты после перестановки: 0000 1111 0000 0000

В этой программе целочисленной переменной sb.num присваивается число 15. Перестановка двух байтов, составляющих это значение, выполняется путем обмена двух символов, которые образуют массив ch. В результате старший и младший байты целочисленной переменнойnum меняются местами. Эта операция возможна лишь потому, что как переменная num, так и массив ch разделяют одну и ту же область памяти.

В следующей программе демонстрируется еще один пример использования объединения. Здесь объединения связываются с битовыми полями, используемыми для отображения в двоичной системе счисления ASCII-кода, генерируемого при нажатии любой клавиши. Эта программа также демонстрирует альтернативный способ отображения отдельных битов,

составляющих байт. Объединение позволяет присвоить значение нажатой клавиши символьной переменной, а битовые поля используются для отображения отдельных битов.

```
// Отображение ASCII-кода символов в двоичной системе счисления.
#include <iostream>
#include <conio.h>
using namespace std;
// Битовые поля, которые будут расшифрованы.
struct byte {
unsigned a : 1;
unsigned b : 1;
unsigned c : 1;
unsigned d : 1;
unsigned e : 1;
unsigned f : 1;
unsigned g : 1;
unsigned h : 1;
};
union bits {
char ch;
struct byte bit;
}ascii;
void disp_bits(bits b);
int main()
{
do {
cin >> ascii.ch;
cout << ":";
disp_bits(ascii);
}while(ascii.ch!='q'); // Выход при вводе буквы "q".
return 0;
}
// Отображение конфигурации битов для каждого символа.
void disp_bits(bits b)
{
if(b.bit.h) cout << "1";
else cout << "0";
if(b.bit.g) cout << "1";
else cout << "0";
if(b.bit.f) cout << "1";
else cout << "0 ";
if(b.bit.e) cout << "1";
else cout << "0";
if(b.bit.d) cout << "1";
else cout << "0";
```

```
if(b.bit.c) cout << "1";
else cout << "0";
if(b.bit.b) cout << "1";
else cout << "0";
if(b.bit.a) cout << "1";
else cout << "0";
cout << "\n";
}
```

Вот как выглядит один из возможных вариантов выполнения этой программы.

```
a: 0 1 1 0 0 0 0 1
b: 0 1 1 0 0 0 1 0
c: 0 1 1 0 0 0 1 1
d: 0 1 1 0 0 1 0 0
e: 0 1 1 0 0 1 0 1
f: 0 1 1 0 0 1 1 0
g: 0 1 1 0 0 1 1 1
h: 0 1 1 0 1 0 0 0
i: 0 1 1 0 1 0 0 1
j: 0 1 1 0 1 0 1 0
k: 0 1 1 0 1 0 1 1
1: 0 1 1 0 1 1 0 0
m: 0 1 1 0 1 1 0 1
n: 0 1 1 0 1 1 1 0
o: 0 1 1 0 1 1 1 1
p: 0 1 1 1 0 0 0 0
q: 0 1 1 1 0 0 0 1
```

Важно! Поскольку объединение предполагает, что несколько переменных разделяют одну и ту же область памяти, это средство предоставляет программисту возможность хранить информацию, которая (в зависимости от ситуации) может содержать различные типы данных, и получать доступ к этой информации. По сути, объединения обеспечивают низкоуровневую поддержку принципов полиморфизма. Другими словами, объединение обеспечивает единый интерфейс для нескольких различных типов данных, воплощая таким образом концепцию "один интерфейс - множество методов" в своей самой простой форме.

3. Битовое поле - это бит-ориентированный член структуры.

В отличие от многих других компьютерных языков, в C++ предусмотрен встроенный способ доступа к конкретному разряду байта. Побитовый доступ возможен путем использования битовых полей. Битовые поля могут оказаться полезными в различных ситуациях. Приведем всего три примера. Во-первых, если вы имеете дело с ограниченным объемом памяти, можно хранить несколько булевых (логических) значений в одном байте. Во-вторых, некоторые интерфейсы устройств передают информацию, закодированную

именно в битах. И, в-третьих, существуют подпрограммы кодирования, которым нужен доступ к отдельным битам в рамках байта. Реализация всех этих функций возможна с помощью поразрядных операторов, как было показано в предыдущей главе, но битовое поле может сделать вашу программу более прозрачной и читабельной, а также повысить ее переносимость.

Метод, который использован в языке C++ для доступа к битам, основан на применении структур. Битовое поле - это в действительности специальный тип члена структуры, который определяет свой размер в битах. Общий формат определения битовых полей таков.

```
struct имя_типа_структуры {
тип имя1 : длина;
тип имя2 : длина;
тип имяN : длина;
};
```

Здесь элемент тип означает тип битового поля, а элемент длина - количество битов в этом поле. Битовое поле должно быть объявлено как значение целочисленного типа или перечисления. Битовые поля длиной 1 бит объявляются как значения типа без знака (unsigned), поскольку единственный бит не может иметь знакового разряда.

Битовые поля обычно используются для анализа входных данных, принимаемых от устройств, входящих в состав оборудования системы. Например, порт состояний последовательного адаптера связи может возвращать байт состояния, организованный таким образом.

Для представления информации, которая содержится в байте состояний, можно использовать следующие битовые поля.

```
struct status_type {
unsigned delta_cts: 1;
unsigned delta_dsr: 1;
unsigned tr_edge: 1;
unsigned delta_rec: 1;
unsigned cts: 1;
unsigned dsr: 1;
unsigned ring: 1;
unsigned rec_line: 1;
} status;
```

Чтобы определить, когда можно отправить или получить данные, используется код, подобный следующему.

```
status = get_port_status();
if(status.cts) cout << "Установка в исходное состояние";
if(status.dsr) cout << "Данные готовы";
```

Чтобы присвоить битовому полю значение, достаточно использовать такую же форму, которая обычно применяется для элемента структуры любого другого типа. Например, следующая инструкция очищает битовое поле ring:

```
status.ring = 0;
```

Как видно из этих примеров, доступ к каждому битовому полю можно получить с помощью оператора"точка". Но если общий доступ к структуре осуществляется через указатель, необходимо использовать оператор "->".
Следует иметь в виду, что совсем необязательно присваивать имя каждому битовому полю. Это позволяет обращаться только к нужным битам, "обходя" остальные. Например, если вас интересуют только биты cts и dsr, вы могли бы объявить структуру status_type следующим образом.

```
struct status_type {
unsigned : 4;
unsigned cts: 1;
unsigned dsr: 1;
} status;
```

Обратите здесь внимание на то, что биты после последнего именованного dsr нет необходимости вообще упоминать.
В структуре можно смешивать "обычные" члены с битовыми полями. Вот пример.

```
struct emp {
struct addr address;
float pay;
unsigned lay_off: 1; // работает или нет
unsigned hourly: 1: // почасовая оплата или оклад
unsigned deductions: 3: // удержание налога
};
```

Эта структура определяет запись по каждому служащему, в которой используется только один байт для хранения трех элементов информации: статус служащего, характер оплаты его труда (почасовая оплата или твердый оклад) и налоговая ставка. Без использования битовых полей для хранения этой информации пришлось бы занять три байта.
Использование битовых полей имеет определенные ограничения. Программист не может получить адрес битового поля или ссылку на него. Битовые поля нельзя хранить в массивах. Их нельзя объявлять статическими. При переходе от одного компьютера к другому невозможно знать наверняка порядок следования битовых полей: справа налево или слева направо. Это означает, что любая программа, в которой используются битовые поля, может страдать определенной зависимостью от марки компьютера. Возможны и другие ограничения, связанные с особенностями реализации компилятора C++, поэтому имеет смысл прояснить этот вопрос в соответствующей документации.
В следующем разделе представлена программа, в которой используются битовые поля для отображения символьных ASCII-кодов в двоичной системе счисления.

Контрольные вопросы

1.Для чего и в каком случае используется тип структура?
2.Ограничено ли количество полей структуры?

3.Назначение полей структуры?
4. Чем отличаются объединение и структура?
5.В каком случае используется typedef?

Тесты:

1) структура это..
а) вид класса
б) прототип класса
в) тип переменной
2) для чего используется typedef
а) для переименования структуры
б) это ключевое слово и с помощью него можно задать новое имя
в) функция для определения типа структуры
3) Элементы структуры называются
а) поля
б) строки
в) указатель

ЛЕКЦИЯ № 3

Тема: Структуры и объединения. Доступ к членам структуры. Присваивание структур. **(2ч.)**

Цель занятия. Ознакомиться с методом доступа к членам структуры и присваиваю структур.

План лекции:

1.Доступ к членам структуры.
2.Структура и объединение.

1.Доступ к членам структуры

Доступ к отдельным членам структуры осуществляется с помощью оператора. (обычно называется «точкой») Например, следующий фрагмент кода присваивает члену zip структурной переменой addr_info значение 12345:

```
addr_info.zip = 12345;
```

За именем структурной переменной следует точка, а за ней имя члена, к которому происходит обращение. Ко всем членам структуры доступ осуществляется точно таким же способом. Стандартный вид доступа следующий:

имя_структуры.имя_члена

Следовательно, для вывода поля zip на экран надо написать:

```
printf("%ld", addr_info.zip);
```

Данная строка выводит на экран содержимое поля zip структурной переменной addr_nfo. Таким же образом массив символов addr_info.name может использоваться в gets():

```
gets (addr_info.name);
```

Данная команда передает указатель на символ, указывающий на начало name.

Для доступа к отдельным элементам addr_info.name можно использовать индекс name. Например, можно вывести содержимое addr_info.name посимвольно с помощью следующего кода:

```
register int t;
for(t=0; addr_info.name [t]; ++t) putchar (addr_info.name [t]);
```

2. Присваивание структур

Содержимое одной структуры можно присвоить другой, если обе эти структуры имеют одинаковый тип. Например, следующая программа присваивает значение структурной переменной svar1переменной svar2.

```
// Демонстрация присваивания значений структур.
```

```
#include <iostream>
using namespace std;
struct stype {
int a, b;
};
int main()
{
stype svar1, svar2;
svar1.a = svar1.b = 10;
svar2.a = svar2.b = 20;
cout << "Структуры до присваивания.\n";
cout << "svar1: " << svar1.a << ' ' << svar1.b;
cout <<'\n';
cout << "svar2: " << svar2.a << ' ' << svar2.b;
cout <<"\n\n";
svar2 = svar1; // присваивание структур
cout << "Структуры после присваивания.\n";
cout << "svar1: " << svar1.a << ' ' << svar1.b;
cout << '\n';
cout << "svar2: " << svar2.a << ' ' << svar2.b;
return 0;
}
```

Эта программа генерирует следующие результаты.

```
Структуры до присваивания.
svar1: 10 10
svar2: 20 20
Структуры после присваивания,
svar1: 10 10
svar2: 10 10
```

В C++ каждое новое объявление структуры определяет новый тип. Следовательно, даже если две структуры физически одинаковы, но имеют разные имена типов, компилятор будет считать их разными и не позволит присвоить значение одной из них другой. Рассмотрим следующий фрагмент кода. Он некорректен и поэтому не скомпилируется.

```
struct stype1 {
int a, b;
};
struct stype2 {
int a, b;
};
stype1 svar1;
stype2 svar2;
svar2 = svar1; // Ошибка из-за несоответствия типов.
```

Несмотря на то что структуры stype1 и stype2 физически одинаковы, с точки зрения компилятора они являются отдельными типами.

ЛЕКЦИЯ № 4

Тема: Структуры и объединения. Массивы структур. Передача структур функциям. Передача членов структур. Передача целых структур. **(2ч.)**

Цель занятия. *Ознакомится с массивами структур, с передачей структур функциям, членов структур и передача целых структур.*

План лекции:

1. *Массив структур.*
2. *Передача структур функциям.*
3. *Передача членов структур.*
4. *Передача целых структур.*
5.

1. Массивы структур

Возможно, наиболее часто структуры используются в виде массивов структур. Для объявления массива структур следует сначала определить структуру, а затем объявить массив переменных данного типа. Например, для объявления 100-элементного массива структур типа addr следует написать:

```
struct addr addr_info[100];
```

В результате получаем набор из 100 переменных, устроенных, как объявлено в типе структуры addr.

Для доступа к отдельным структурам массива adar_info следует проиндексировать имя массива. Например, для вывода содержимого поля zip третьей структуры, следует написать:

```
printf("%ld", addr_info[2].zip);
```

Как и массивы переменных, массивы структур индексируются с нуля.

2.Передача структур в функции

Все структуры и массивы структур, используемые в примерах, или являются глобальными, или определяются в функциях, их использующих. Здесь подробно здесь подробно будет рассмотрена передача структур и их членов в функции.

3.Передача членов структур

При передаче членов структур в функции фактически передается значение члена. Следовательно, передается обычная переменная. Рассмотрим для примера следующую структуру:

```
struct fred {
char x;
int y;
float z;
```

```
char s[10];
} mike;
```

Ниже приведены примеры передачи каждого члена в функцию:

```
func(mike.x); /* передача символьного значения x */
func2(mike.y); /* передача целочисленного значения y */
func3(mike.z); /* передача вещественного значения z */
func4(mike.s); /* передача адреса строки s */
func(mike.s[2]); /* передача символьного значения s [2] */
```

Тем не менее, если необходимо передать адрес отдельного члена структуры, следует поместить оператор & перед именем структуры. Например, для передачи адреса элементов структуры mike следует написать:

```
func(&mxke.x) ; /* передача адреса символа x */
func2(&mike.y); /* передача адреса целого y */
func3(&mike.z); /* передача адреса вещественного z */
func4(mike.s) ; /* передача адреса строки s */
func(&mike.s [2]); /* передача адреса символа s[2] */
```

Обратим внимание, что оператор & стоит перед именем структуры, а не перед именем члена. Помимо этого, массив s сам по себе является адресом, поэтому не требуется оператора &. Тем не менее, когда осуществляется доступ к отдельному символу строки s, как показано в последнем примере, оператор & необходим.

4.Передача целых структуры

Когда структура используется как аргумент функции, передается вся структура с помощью стандартной передачи по значению. Это означает, что любые изменения, внесенные в содержимое структуры внутри функции, не повлияют на структуру, используемую в качестве аргумента.

Когда структура используется как параметр, самое важное - это запомнить, что тип аргумента должен соответствовать типу параметра. Лучший способ сделать это - определить структуру глобально, а затем использовать ее ярлык для объявления необходимых структурных переменных и параметров. Например:

```
#include <stdio.h>

/* объявление типа структуры */
struct struct_type {
int a, b;
char ch;
};
```

```
void f1(struct struct_type parm);

int main(void)
{
struct struct_type arg;    /* объявление arg */
arg.a = 1000;
f1(arg);
return 0;
}

void f1(struct struct_type parm) {
printf("%d", parm.a);
}
```

Данная программа выводит число 1000 на экран. Можно видеть, что как arg, так и parm объявлены как структуры типа struct_type.

ЛЕКЦИЯ № 5

Тема: Структуры и объединения. Указатели на структуры. Объявление указателей на структуры. **(2ч.)**

Цель занятия. *Понять принцип указателя на структуры, объявления указателей на структуры.*

План лекции:

1. Указатели на структуры.
2. Объявление указателя на структуру.
3. Использование указателей на структуру

1.Указатели на структуры

В языке С указатели на структуры также официально признаны, как и указатели на любой другой вид объектов. Однако указатели на структуры имеют некоторые особенности, о которых и пойдет речь.

2.Объявление указателя на структуру

Как и другие указатели, указатель на структуру объявляется с помощью звездочки *, которую помещают перед именем переменной структуры. Например, для ранее определенной структуры addr следующее выражение объявляет addr_pointer указателем на данные этого типа (то есть на данные типа addr): struct addr *addr_pointer;

3.Использование указателей на структуры

Указатели на структуры используются главным образом в двух случаях: когда структура передается функции с помощью вызова по ссылке, и когда создаются связанные друг с другом списки и другие структуры с динамическими данными, работающие на основе динамического размещения. В этой главе рассматривается первый случай.

У такого способа, как передача любых (кроме самых простых) структур функциям, имеется один большой недостаток: при выполнении вызова функции, чтобы поместить структуру в стек, необходимы существенные ресурсы. (Вспомните, что аргументы передаются функциям через стек.) Впрочем, для простых структур с несколькими членами эти ресурсы являются не такими уж большими. Но если в структуре имеется большое количество членов или некоторые члены сами являются массивами, то при передаче структур функциям производительность может упасть до недопустимо низкого уровня. Как же решить эту проблему? Надо передавать не саму структуру, а указатель на нее.

Когда функции передается указатель на структуру, то в стек попадает только адрес структуры. В результате вызовы функции выполняются очень быстро. В некоторых случаях этот способ имеет еще и второе преимущество: передача указателя позволяет функции модифицировать содержимое структуры, используемой в качестве аргумента.

Чтобы получить адрес переменной-структуры, необходимо перед ее именем поместить оператор &. Например, в следующем фрагменте кода

```
struct bal {

  float balance;

  char name[80];

} person;



struct bal *p;  /* объявление указателя на структуру */
```

адрес структуры person можно присвоить указателю p:

```
p = &person;
```

Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка ->. Вот, например, как можно сослаться на полebalance:

```
p->balance
```

Оператор ->, который обычно называют оператором стрелки, состоит из знака "минус", за которым следует знак "больше". Стрелка применяется вместо оператора точки тогда, когда для доступа к члену структуры используется указатель на структуру.

Чтобы увидеть, как можно использовать указатель на структуру, проанализируйте следующую простую программу, которая имитирует таймер, выводящий значения часов, минут и секунд:

```
/* Программа-имитатор таймера. */

#include <stdio.h>



#define DELAY 128000
```

```
struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
    display(&systime);
  }
  return 0;
```

```
}
void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }
 if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }
if(t->hours==24) t->hours = 0;
  delay();
}
void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}
void delay(void)
{
  long int t;
```

```
 /* если надо, можно изменять константу DELAY (задержка) */

  for(t=1; t<DELAY; ++t) ;

}
```

Эту программу можно настраивать, меняя определение DELAY.

В этой программе объявлена глобальная структура my_time, но при этом не объявлены никакие другие переменные программы. Внутри же main() объявлена структурasystime и она инициализируется значением 00:00:00. Это значит, что systime непосредственно видна только в функции main().

Функциям update() (которая изменяет значения времени) и display() (которая выводит эти значения) передается адрес структуры systime. Аргументы в обеих функциях объявляются как указатель на структуру my_time.

Внутри update() и display() доступ к каждому члену systime осуществляется с помощью указателя. Так как функция update() принимает указатель на структуру systime, то она в состоянии обновлять значение этой структуры. Например, необходимо "в полночь", когда значение переменной, в которой хранится количество часов, станет равным 24, сбросить отсчет и снова сделать значение этой переменной равным 0. Для этого в update() имеется следующая строка:

```
if(t->hours==24) t->hours = 0;
```

Таким образом, компилятору дается указание взять адрес t (этот адрес указывает на переменную systime из main()) и сбросить значение hours в нуль.

Лекция № 6

Тема: Файлы и работа с файлами. Потоковый ввод/вывод. Файлы и потоки. Стандартные файлы и функции для работы ввод/вывод. Манипуляторы потоков. **(2ч.)**

Цель занятия. *Ознакомиться с понятием файл, работой с файлами, с функциями для работы ввода и вывода.*

План лекции:

1.Файл.

2. Форматированный файловый ввод-вывод.

3.Файловый ввод-вывод с использованием потоков

4.Использование манипуляторов

1.Файлом называют способ хранения информации на физическом устройстве. Файл — это понятие, которое применимо ко всему — от файла на диске до терминала.

В C++ отсутствуют операторы для работы с файлами. Все необходимые действия выполняются с помощью функций, включенных в стандартную библиотеку. Они позволяют работать с различными устройствами, такими, как диски, принтер, коммуникационные каналы и т.д. Эти устройства сильно отличаются друг от друга. Однако файловая система преобразует их вединое абстрактное логическое устройство, называемое потоком.

Текстовый поток — это последовательность символов. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток — это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Организация работы с файлами средствами C++

Объявление файла

FILE *идентификатор;

Пример

FILE *f;

Открытие файла:

fopen(имя физического файла, режим доступа)

Режим доступа — строка, указывающая режим открытия файла файла и тип файла

Типы файла: бинарный (b); текстовый (t)

Значение	Описание
R	Файл открывается только для чтения
W	Файл открывается только для записи. Если соответствующий физический файл существует, он будет перезаписан
A	Файл открывается для записи в конец (для дозаписи) или создается, если не существует
r+	Файл открывается для чтения и записи.

W+	Файл открывается для записи и чтения. Если соответствующий физический файл существует, он будет перезаписан
a+	Файл открывается для записи в конец (для дозаписи) или создается, если не существует

Например
f = fopen(s, "wb");
k = fopen("h:\ex.dat", "rb");
Неформатированные файловый ввод-вывод
Запись в файл

fwrite(адрес записываемой величины, размер одного экземпляра, количество записываемых величин, имя логического файла);

Например,

fwrite(&dat, sizeof(int), 1, f);

Чтение из файла

fread(адрес величины, размер одного экземпляра, количество считываемых величин, имя логического файла);

Например,

fread(&dat, sizeof(int), 1, f);

Закрытие файла

fclose(имя логического файла);

Пример 1. Заполнить файл некоторым количеством целых случайных чисел.

```
/* Заполнить файл некоторым количеством целых случайных чисел. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
 FILE *f; int dat;
 srand(time(0));
 int n=rand()%30 + 1;
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "wb");
 for (int i=1; i<=n; i++)
 { dat = rand()%101 - 50;
   cout << dat << " ";
   fwrite(&dat, sizeof(int), 1, f);
 }
 cout << endl;
```

```
 fclose(f);
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Пример 2. Найти сумму и количество целых чисел, записанных в бинарный файл.

```
/* Найти сумму и количество целых чисел, записанных в бинарный файл. */
/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
 FILE *f;
 int dat, n=0, sum=0;

 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "rb");
 while (fread(&dat, sizeof(int), 1, f))
       {n++;
        cout << dat << " ";
        sum+=dat;
       }
 cout << endl;
 cout << "sum: " << sum << "; number: " << n << endl;
 fclose(f);
 system("PAUSE");
 return EXIT_SUCCESS;
}
```

Пример 3. Поместить в файл n записей, содержащих сведения о кроликах, содержащихся в хозяйстве: пол (m/f), возраст (в мес.), масса.

```
/* Поместить в файл n записей, содержащих сведения о кроликах, содержащихся в хозяйстве:
   пол (m/f), возраст (в мес.), масса. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;
```

```
struct krolik {char pol; int vozrast; double massa;};

int main()
{
 FILE *f; krolik dat; int n;
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "wb");
 cout << "How many rabbits? "; cin >> n;
 for (int i=1; i<=n; i++)
 { cout << "What sex " << i << "th rabbit? "; cin >> dat.pol;
   cout << "How old " << i << "th rabbit? "; cin >> dat.vozrast;
   cout << "What is the mass of the " << i << "th rabbit? "; cin >> dat.massa;
   fwrite(&dat, sizeof(krolik), 1, f);
 }
 fclose(f);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Пример 3 (продолжение). В бинарном файле хранятся сведения о кроликах, содержащихся в хозяйстве: пол (m/f), возраст (в мес.), масса. Найти наиболее старого кролика. Если таких несколько, то вывести информацию о том из них, масса которого больше.

```
/* В бинарном файле хранятся сведения о  кроликах, содержащихся в хозяйстве: пол (m/f),  возраст (в мес.), масса.
  Найти наиболее старого кролика. Если таких несколько, то вывести информацию о том из них, масса которого больше. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>

using namespace std;

struct krolik {char pol; int vozrast; double massa;};

int main()
{
 FILE *f; krolik dat, max; int n;
 cout << "File name? ";
 char s[20];
 cin.getline(s, 20);
 f=fopen(s, "rb");
 fread(&dat, sizeof(krolik), 1, f);
```

```
 max=dat;
 while (fread(&dat, sizeof(krolik), 1, f))
 {if (dat.vozrast>max.vozrast) max=dat;
  else if (dat.vozrast==max.vozrast&&dat.massa>max.massa) max=dat;}
 cout << "The oldest rabbit has a sex " << max.pol << ", age " << max.vozrast << "
and mass " << max.massa << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.Форматированный файловый ввод-вывод

1) Функции fgetc() и fputc() позволяют соттветственно осуществить ввод-вывод символа.

2) Функции fgets() и fputs() позволяют соттветственно осуществить ввод-вывод строки.

3) Функции fscanf() и fprintf() позволяют соттветственно осуществить форматированный ввод-вывод и аналогичный соответствующим функиям форматированного ввода-вывода, только делают это применительно к файлу.

Организация работы с файлами средствами C++

3.Файловый ввод-вывод с использованием потоков

Библиотека потокового ввода-вывода

fstream

Связь файла с потоком вывода

ofstream имя логического файла;

Связь файла с потоком ввода

ifstream имя логического файла;

Открытие файла

имя логического файла.open(имя физического файла);

Закрытие файла

имя логического файла.close();

Пример 4. Заполнить файл значениями функции y = x * cos x.

```
/* Заполнить файл значениями функции y = x * cos x. */

/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <cmath>

using namespace std;

double fun(double x);

int main()
{double a, b, h, x; char s[20];
 cout << "Enter the beginning and end of the segment, step-tabulation: ";
```

```
 cin >> a >> b >> h;
 cout << "File name? "; cin >> s;
 ofstream f;
 f.open(s);
 for (x=a; x<=b; x+=h)
  {f.width(10);   f << x;
   f.width(15);   f << fun(x) << endl;  }
 f.close();

    system("PAUSE");
    return EXIT_SUCCESS;
}
double fun(double x)
{ return x*cos(x); }
```

Пример 5. Файл содержит несколько строк, в каждой из которых записано единственное выражение вида a#b (без ошибок), где a, b - целочисленные величины, # - операция +, -, /, *. Вывести каждое из выражений и их значения.

```
/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>

using namespace std;
int main()
{
long a, b;  char s[256], c;  int i;
  cout << "File name? "; cin >> s;
  ifstream f;  f.open(s);
  while (!f.eof())
   {    f.getline(s, 256);
    i=0; a=0;
    while (s[i]>='0'&&s[i]<='9')
     {
          a=a*10+s[i]-'0';
           i++;
     }
    c=s[i++];    b=0;
    while (s[i]>='0'&&s[i]<='9')
     {
          b=b*10+s[i]-'0';
           i++;
     }
    switch (c){
    case '+': a+=b; break;
```

```
   case '-': a-=b; break;
   case '/': a/=b; break;
   case '*': a*=b; break;}
   cout << s << " = " << a << endl;   }
   f.close();
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Пример 6. В заданном файле целых чисел посчитать количество компонент, кратных 3.

```
/* В заданном файле целых чисел посчитать количество компонент, кратных 3. */
/* Dev-C++ */
#include <cstdlib>
#include <iostream>
#include <fstream>

using namespace std;
int main()
{int r,ch;
 ifstream f;
 f.open("CH_Z.TXT");
 ch=0;
 for (;f.peek()!=EOF;)
   {f>>r;
    cout << r << " ";
    if (r%3==0) ch++ ;
    }
 f.close();
 cout << endl << "Answer: " << ch;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

4.Использование манипуляторов

Система ввода/вывода C++ включает второй способ изменения параметров форматирования потока. Для этого используются специальные функции, называемые манипуляторами (manipulators), которые могут включаться в выражения ввода/вывода. Стандартные манипуляторы показаны в таблице.

Манипулятор	**Назначение**	**Ввод/вывод**
dec	Ввод/вывод данных в десятичной форме	ввод и вывод
endl	Вывод символа новой строки с передачей в поток всех данных из буфера	вывод

Манипулятор	Назначение	Ввод/вывод
ends	Вывод нулевого символа	вывод
flush	Передача в поток содержимого буфера	вывод
hex	Ввод/вывод данных в шестнадцатиричной системе	ввод и вывод
oct	Ввод/вывод данных в восьмеричной форме	ввод и вывод
resetiosflags(long f)	Сбрасывает флаги, указанные в f	ввод и вывод
setbase(int base)	Устанавливает базу счисления равной параметру base	вывод
setfill(int ch)	Устанавливает символ заполнения равным ch	вывод
setiosflags(long f)	Устанавливает флаги, указанные в f	ввод и вывод
setprecision(int p)	Устанавливает число цифр после запятой	вывод
setw(int w)	Устанавливает ширину поля равной w	вывод
ws	Пропускает начальный символ-разделитель	ввод

Таблица: Манипуляторы ввода/вывода C++

Для использования манипуляторов с параметрами в программу необходимо включить заголовочный файл iomanip.h. Манипуляторы могут использоваться в составе выражений ввода/вывода. Ниже представлен пример программы, использующей манипуляторы для изменения формата вывода:

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
cout << setiosflags(ios::fixed);
cout << setprecision (2) << 1000.243 << endl;
cout << setw (20) << "Hello there.";
return 0;
}
```

Программа выводит следующие данные:

```
1000                                                .24
                Hello there.
```

Обратим внимание, как манипуляторы появляются в последовательности операторов ввода/вывода. Когда манипуляторы не имеют аргументов, как манипулятор endl в этой программе, за ними не следуют скобки. Причина этого в том, что оператору << передается адрес манипулятора.

Следующая программа использует функции setiosflags() для установки флагов scientific и showpos потока cout:

```
#include <iostream.h>
#include <iomanip.h>
main ()
{
cout << setiosflags(ios::showpos);
cout << setiosflags(ios::scientific);
cout << 123 << " " << 123.23;
return 0;
}
```

Следующая программа использует манипулятор ws для пропуска идущих вначале символов- разделителей при вводе строки в переменную s:

```
#include <iostream.h>
int main()
{
char s [80];
cin >> ws >> s;
cout << s;
}
```

ЛЕКЦИЯ № 7

Тема: Файлы и работа с файлами. Текстовые файлы. Двоичные файлы. Файловый ввод/вывод с помощью потоков в стиле C++.**(2ч.)**

Цель занятия. *Ознакомится с текстовыми файлами, познакомится со способами для записи двоичных данных в файл*

План лекции.

1. Текстовые файлы.
2. Двоичные файлы.
3. Файловые операции ввода/вывода в C++.
4. Вывод в файловый поток.
5. Определение конца файла.
6. Проверка ошибок при выполнении файловых операций.
7. Управление открытием файла.

1.Текстовые файлы

Для того, чтобы писать в текстовые файлы или читать из них, достаточно воспользоваться операторами << и >> для открытого потока. Например, следующая программа записывает целое число, число с плавающей запятой и строку в файл TEST:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
ofstream out("test");
if (!out) {
cout << "Cannot open file.\n";
return 1;
}
out << 10 << " " << 123.23 << "\n";
out << "This is a short text file.\n";
out.close ();
return 0;
}
```

Следующая программа читает целое число, число с плавающей запятой, символ и строку из файла, созданного предыдущей программой:

```
#include <iostream.h>
#include <fstream.h>
int main()
```

```
{
char ch;
int i;
float f;
char str[80];
ifstream in("test");
if (!in) {
cout << "Cannot open file.\n";
return 1;
}
in >> i;
in >> f;
in >> ch;
in >> str;
cout << i << " " << f << " " << ch << "\n";
cout << str;
in.close();
return 0;
}
```

При использовании оператора >> для чтения текстовых файлов надо иметь в виду, что происходит определенное преобразование символов. Например, символы-разделители опускаются. Если нужно предотвратить какое-либо преобразование символов, то необходимо использовать функции двоичного ввода/вывода C++, которые рассматриваются в следующем разделе.

2.Двоичные файлы

Имеется несколько способов для записи двоичных данных в файл и чтения из файла. В этом разделе мы рассмотрим два из них. В первую очередь, можно записать байт с помощью функции-члена put() и прочитать байт, используя функцию-член get(). Функция get() имеет много форм, но наиболее употребительна показанная ниже версия, где приведена также функция put():

istream &get(char &ch);
ostream &put(char ch);

Функция get() читает единственный символ из ассоциированного потока и помещает его значение в ch. Она возвращает ссылку на поток. Функция put() пишет ch в поток и возвращает ссылку на этот поток.

Следующая программа выводит содержимое любого файла на экран. Она использует функцию get().

```
#include <iostream.h>
#include <fstream.h>
int main(int argc, char *argv[])
{
```

```
char ch;
if (argc!=2) {
cout << "Usage: PR <filename>\n";
return 1;
}
ifstream in(argv[1], ios::in | ios::binary);
if (!in) {
cout << "Cannot open file.\n";
return 1;
}
while (in) { // in будет нулем при достижении конца файла
in.get (ch);
cout << ch;
}
in.close();
return 0;
}
```

Когда in достигает конца файла, то принимает значение NULL, в результате чего цикл while заканчивается.

Имеется более компактная запись кода для этого цикла, как показано ниже:

```
while (in.get(ch))
cout << ch;
```

Такая запись работает, поскольку функция get() возвращает поток in, обращающийся в нуль, когда достигается конец файла.

Следующая программа использует функцию put() для записи строки, содержащей не ASCII- символы:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
char *p = "hello there\n\r\xfe\xff";
ofstream out("test", ios::out | ios::binary );
if (!out) {
cout << "Cannot open file.\n";
return 1;
}
while (*p) out.put (*p++);
out.close ();
return 0;
}
```

Второй способ чтения и записи двоичных данных состоит в использовании функций read() и write(). Наиболее обычный способ использования этих функций соответствует прототипу:

*istream &read(unsigned char *buf, int num);*
*ostream &write(const unsigned char *buf, int num);*

Функция read() читает num байт из ассоциированного потока и посылает их в буфер buf. Функция write() пишет num байт в ассоциированный поток из буфера buf.

Следующая программа пишет и потом читает массив целых чисел:

```
#include <iostream.h>
#include <fstream.h>
int main()
{
int n [5] = {1, 2, 3, 4, 5};
register int i;
ofstream out ("test", ios::out | ios::binary);
if (!out) {
cout << "Cannot open file.\n";
return 1;
}
out.write((unsigned char *) &n, sizeof n);
out.close();
for (i=0; i<5; i++) // очистка массива
n[i] = 0;
ifstream in ("test", ios::in | ios::binary);
in.read((unsigned char *) &n, sizeof n);
for (i=0; i<5; i++) // вывести значения, прочитанные из файла
cout << n[i] << " ";
in.close();
return 0;
}
```

Следует обратить внимание, что приведение типов в вызовах read() и write() необходимо для работы с буфером, который не определен как массив символов.

Если конец файла достигается до того, как будет прочитано заданное число символов, функция read() просто прекращает работу и буфер содержит столько символов, сколько было прочитано. Можно определить, сколько символов было прочитано, используя другую функцию-член gcount(), имеющую следующий прототип:

int gcount();

Она возвращает число символов, прочитанных последним оператором двоичного ввода.

3.ФАЙЛОВЫЕ ОПЕРАЦИИ В/В В С++

По мере усложнения ваших программ они будут сохранять и получать информацию, используя файлы. Если вы знакомы с файловыми манипуляциями в языке С, вы сможете использовать подобные методы и в С++. Кроме того, как вы узнаете из этого урока, С++ предоставляет набор классов файловых потоков, с помощью которых можно очень легко выполнять операции ввода и вывода (В/В) с файлами. К концу данного урока вы освоите следующие основные концепции:

Используя выходной файловый поток, вы можете писать информацию в файл с помощью оператора вставки (<<).

Используя входной файловый поток, вы можете читать хранимую в файле информацию с помощью оператора извлечения (>>).

Для открытия и закрытия файла вы используете методы файловых классов.

Для чтения и записи файловых данных вы можете использовать операторы вставки и извлечения, а также некоторые методы файловых классов.

Многие программы, которые вы создадите в будущем, будут интенсивно использовать файлы. Выберите время для экспериментов с программами, представленными в данном уроке. И вы обнаружите, что в С++ выполнять файловые операции очень просто.

4.ВЫВОД В ФАЙЛОВЫЙ ПОТОК

Из урока 33 вы узнали, что cout представляет собой объект типа ostream(выходной поток). Используя класс ostream, ваши программы могут выполнять вывод в cout с использованием оператора вставки или различных методов класса, например cout.put. Заголовочный файлiostream.h определяет выходной поток cout. Аналогично, заголовочный файл f stream.h определяет класс выходного файлового потока с именемofstream. Используя объекты класса ofstream, ваши программы могут выполнять вывод в файл. Для начала вы должны объявить объект типаofstream, указав имя требуемого выходного файла как символьную строку, что показано ниже:

```
ofstream file_object("FILENAME.EXT");
```

Если вы указываете имя файла при объявлении объекта типа ofstream, С++ создаст новый файл на вашем диске, используя указанное имя, или перезапишет файл с таким же именем, если он уже существует на вашем диске Следующая программа OUT_FILE.CPP создает объект типа ofstreamи затем использует оператор вставки для вывода нескольких строк текста в файл BOOKINFO.DAT:

```
#include <fstream.h>
void main(void)
{
  ofstream book_file("BOOKINFO.DAT");
  book_file << "Учимся программировать на языке С++, " << "Вторая
```

```
редакция" << endl;
  book_file << "Jamsa Press" << endl;
  book_file << "22.95" << endl;
}
```

В данном случае программа открывает файл BOOKINFO.DAT и затем записывает три строки в файл, используя оператор вставки. Откомпилируйте и запустите эту программу. Если вы работаете в среде MS-DOS, можете использовать команду TYPE для вывода содержимого этого файла на экран:

```
C:\> TYPE BOOKINFO.DAT <ENTER>
```

Как видите, в C++ достаточно просто выполнить операцию вывода в файл.

ЧТЕНИЕ ИЗ ВХОДНОГО ФАЙЛОВОГО ПОТОКА

Только что вы узнали, что, используя класс ofstream, ваши программы могут быстро выполнить операции вывода в файл. Подобным образом ваши программы могут выполнить операции ввода из файла, используя объекты типа ifstream. Опять же, вы просто создаете объект, передавая ему в качестве параметра требуемое имя файла:

```
ifstream input_file("filename.EXT");
```

Следующая программа FILE_IN.CPP открывает файл BOOKINFO.DAT, который вы создали с помощью предыдущей программы, и читает, а затем отображает первые три элемента файла:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  ifstream input_file("BOOKINFO.DAT") ;
  char one[64], two[64], three[64];
  input_file >> one;
  input_file >> two;
  input_file >> three;
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

Если вы откомпилируете и запустите эту программу, то, вероятно, предположите, что она отобразит первые три строки файла. Однако, подобно cin, входные файловые потоки используют пустые символы, чтобы определить, где заканчивается одно значение и начинается другое. В результате при запуске предыдущей программы на дисплее появится следующий вывод:

```
C:\> FILE_IN <ENTER>
учимся
программировать
на
```

Чтение целой строки файлового ввода

Из урока 33 вы узнали, что ваши программы могут использовать cin.getlineдля чтения целой строки с клавиатуры. Подобным образом объекты типаifstream могут использовать getline для чтения строки файлового ввода. Следующая программа FILELINE.CPP использует функцию getline для чтения всех трех строк файла BOOKINFO.DAT:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  ifstream input_file("BOOKINFO.DAT");
  char one[64], two[64], three [64] ;
  input_file.getline(one, sizeof(one)) ;
  input_file.get line(two, sizeof(two));
  input_file.getline(three, sizeof(three)) ;
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

В данном случае программа успешно читает содержимое файла, потому что она знает, что файл содержит три строки. Однако во многих случаях ваша программа не будет знать, сколько строк содержится в файле. В таких случаях ваши программы будут просто продолжать чтение содержимого файла пока не встретят конец файла.

5.ОПРЕДЕЛЕНИЕ КОНЦА ФАЙЛА

Обычной файловой операцией в ваших программах является чтение содержимого файла, пока не встретится конец файла. Чтобы определить конец файла, ваши программы могут использовать функцию eof потокового объекта. Эта функция возвращает значение 0, если конец файла еще не встретился, и 1, если встретился конец файла. Используя цикл while, ваши программы могут непрерывно читать содержимое файла, пока не найдут конец файла, как показано ниже:

```
while (! input_file.eof())
{
  // Операторы
}
```

В данном случае программа будет продолжать выполнять цикл, пока функция eof возвращает ложь (0). Следующая программа TEST_EOF.CPP использует функцию eof для чтения содержимого файла BOOKINFO.DAT, пока не достигнет конца файла:

```
#include <iostream.h>
#include <fstream.h>
void main (void)
{
  ifstream input_file("BOOKINFO.DAT");
```

```
  char line[64];
  while (! input_file.eof())
  {
    input_file.getline(line, sizeof(line));
    cout << line << endl;
  }
}
```

Аналогично, следующая программа WORD_EOF.CPP читает содержимое файла по одному слову за один раз, пока не встретится конец файла:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  ifstream input_file("BOOKINFO.DAT");
  char word[64] ;
  while (! input_file.eof())
  {
    input_file >> word;
    cout << word << endl;
  }
}
```

И наконец, следующая программа CHAR_EOF.CPP читает содержимое файла по одному символу за один раз, используя функцию get, пока не встретит конец файла:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  ifstream input_file("BOOKINFO.DAT");
  char letter;
  while (! input_file.eof())
  {
    letter = input_file.get();
    cout << letter;
  }
}
```

6.ПРОВЕРКА ОШИБОК ПРИ ВЫПОЛНЕНИИ ФАЙЛОВЫХ ОПЕРАЦИЙ

Программы, представленные до настоящего момента, предполагали, что во время файловых операций В/В не происходят ошибки. К сожалению, это сбывается не всегда. Например, если вы открываете файл для ввода, ваши программы должны проверить, что файл существует. Аналогично, если ваша программа пишет данные в файл, вам необходимо убедиться, что операция прошла успешно (к примеру, отсутствие места на диске, скорее всего, помешает записи данных). Чтобы помочь вашим программам следить за

ошибками, вы можете использовать функцию fail файлового объекта. Если в процессе файловой операции ошибок не было, функция возвратит ложь (0). Однако, если встретилась ошибка, функция fail возвратит истину. Например, если программа открывает файл, ей следует использовать функцию fail, чтобы определить, произошла ли ошибка, как это показано ниже:

```
ifstream input_file("FILENAME.DAT");
if (input_file.fail())
{
  cerr << "Ошибка открытия FILENAME.EXT" << endl;
  exit(1);
}
```

Таким образом, программы должны убедиться, что операции чтения и записи прошли успешно. Следующая программа TEST_ALL.CPP использует функцию fail для проверки различных ошибочных ситуаций:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  char line[256] ;
  ifstream input_file("BOOKINFO.DAT") ;
  if (input_file.fail()) cerr << "Ошибка открытия BOOKINFO.DAT"<< endl;
  else
  {
    while ((! input_file.eof()) && (! input_file.fail()))
    {
       input_file.getline(line, sizeof(line)) ;
      if (! input_file.fail()) cout << line << endl;
    }
  }
}
```

ЗАКРЫТИЕ ФАЙЛА, ЕСЛИ ОН БОЛЬШЕ НЕ НУЖЕН

При завершении вашей программы операционная система закроет открытые ею файлы. Однако, как правило, если вашей программе файл больше не нужен, она должна его закрыть. Для закрытия файла ваша программа должна использовать функцию close, как показано ниже:

```
input_file.close ();
```

Когда вы закрываете файл, все данные, которые ваша программа писала в этот файл, сбрасываются на диск, и обновляется запись каталога для этого файла.

7.УПРАВЛЕНИЕ ОТКРЫТИЕМ ФАЙЛА

В примерах программ, представленных в данном уроке, файловые операции ввода и вывода выполнялись с начала файла. Однако, когда вы записываете данные в выходной файл, вероятно, вы захотите, чтобы программа добавляла информацию в конец существующего файла. Для открытия файла в режиме

добавления вы должны при его открытии указать второй параметр, как показано ниже:

```
ifstream output_file("FILENAME.EXT", ios::app);
```

В данном случае параметр ios::app указывает режим открытия файла. По мере усложнения ваших программ они будут использовать сочетание значений для режима открытия файла, которые перечислены в табл. 34.

Таблица 34. Значения режимов открытия.

Режим открытия	Назначение
ios::app	Открывает файл в режиме добавления, располагая файловый указатель в конце файла.
ios::ate	Располагает файловый указатель в конце файла.
ios::in	Указывает открыть файл для ввода .
ios::nocreate	Если указанный файл не существует, не создавать файл и возвратить ошибку.
ios::noreplace	Если файл существует, операция открытия должна быть прервана и должна возвратить ошибку.
ios::out	Указывает открыть файл для вывода.
ios::trunc	Сбрасывает (перезаписывает) содержим, з существующего файла.

Следующая операция открытия файла открывает файл для вывода, используя режим ios::noreplace, чтобы предотвратить перезапись существующего файла:

```
ifstream output_file("FIlename.EXT", ios::out | ios::noreplace);
```

ВЫПОЛНЕНИЕ ОПЕРАЦИЙ ЧТЕНИЯ И ЗАПИСИ

Все программы, представленные в данном уроке, выполняли файловые операции над символьными строками. По мере усложнения ваших программ, возможно, вам понадобится читать и писать массивы и структуры. Для этого ваши программы могут использовать функции read иwrite. При использовании функций read и write вы должны указать буфер данных, в который данные будут читаться или из которого они будут записываться, а также длину буфера в байтах, как показано ниже:

```
input_file.read(buffer, sizeof(buffer)) ;
output_file.write(buffer, sizeof(buffer));
```

Например, следующая программа STRU_OUT.CPP использует функциюwrite для вывода содержимого структуры в файл EMPLOYEE.DAT:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  struct employee
  {
    char name[64];
```

```
    int age;
    float salary;
  } worker = { "Джон Дой", 33, 25000.0 };
  ofstream emp_file("EMPLOYEE.DAT") ;
  emp_file.write((char *) &worker, sizeof(employee));
}
```

Функция write обычно получает указатель на символьную строку. Символы (char *) представляют собой оператор приведения типов, который информирует компилятор, что вы передаете указатель на другой тип. Подобным образом следующая программа STRU_IN.CPP использует метод read для чтения из файла информации о служащем:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  struct employee
  {
    char name [6 4] ;
    int age;
    float salary;
  } worker = { "Джон Дой", 33, 25000.0 };
  ifstream emp_file("EMPLOYEE.DAT");
  emp_file.read((char *) &worker, sizeof(employee));
  cout << worker.name << endl;
  cout << worker.age << endl;
  cout << worker.salary << endl;
}
```

ЛЕКЦИЯ №8

Тема: Файлы и работа с файлами. Ввод и вывод файлов прямого доступа. Обработка исключительных ситуаций. Генерирование исключеный. Перехватывание исключений. Использование вложенных блоков try/catch. **(2ч.)**

Цель занятия. *Ознакомится с вводом и выводом файлов прямого доступа, обработкой исключений, перехватыванием исключений.*

План лекции:

1. Ввод/вывод при прямом доступе
2. Генерирование исключений
3. Перехват всех исключений
4. Работа с конструкцией TRY…CATCH

1.Ввод/вывод при прямом доступе

При прямом доступе можно выполнять операции ввода/вывода, используя систему ввода/вывода языка С и функцию `fseek()`, которая устанавливает указатель текущей позиции в файле. Вот прототип этой функции:

int fseek(FILE *_уф_, long int _колич_байт_, int _начало_отсчета_);

Здесь _уф_ — это указатель файла, возвращаемый в результате вызова функции `fopen()`, _колич_байт_ — количество байтов, считая от _начало_отсчета_, оно определяет новое значение указателя текущей позиции, а _начало отсчёта_ — это один из следующих макросов:

Начало отсчета	Макрос
Начало файла	SEEK_SET
Текущая позиция	SEEK_CUR
Конец файла	SEEK_END

Поэтому, чтобы получить в файле доступ на расстоянии _колич_байт_ байтов от начала файла, _начало_отсчета_ должно равняться `SEEK_SET`. Чтобы при доступе расстояние отсчитывалось от текущей позиции, используйте макрос `SEEK_CUR`, а чтобы при доступе расстояние отсчитывалось от конца файла, нужно указывать макрос `SEEK_END`. При успешном завершении своей работы функция `fseek()` возвращает нуль, а в случае ошибки — ненулевое значение.

В следующей программе показано, как используется `fseek()`. Данная программа в определенном файле отыскивает некоторый байт, а затем отображает его. В командной строке нужно указать имя файла, а затем нужный байт, то есть его расстояние в байтах от начала файла.

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
  FILE *fp;

  if(argc!=3) {
    printf("Синтаксис: SEEK <имя_файла> <байт>\n");
    exit(1);
  }

  if((fp = fopen(argv[1], "rb"))==NULL) {
    printf("Ошибка при открытии файла.\n");
    exit(1);
  }

  if(fseek(fp, atol(argv[2]), SEEK_SET)) {
    printf("Seek error.\n");
    exit(1);
  }

  printf("В %ld-м байте содержится %c.\n", atol(argv[2]), getc(fp));
  fclose(fp);

  return 0;
}
```

Функцию `fseek()` можно использовать для доступа внутри многих значений одного типа, просто умножая размер данных на номер элемента, который вам нужен. Например, предположим, имеется список рассылки, который состоит из структур типа `addr` (определенных ранее). Чтобы получить доступ к десятому адресу в файле, в котором хранятся адреса, используйте следующий оператор:

```
fseek(fp, 9*sizeof(struct addr), SEEK_SET);
```

Текущее значение указателя текущей позиции в файле можно определить с помощью функции `ftell()`. Вот ее прототип:

```
long int ftell(FILE *уф);
```

Функция возвращает текущее значение указателя текущей позиции в файле, связанном с указателем файла *уф*. При неудачном исходе она возвращает -1.

Обычно прямой доступ может потребоваться лишь для двоичных файлов. Причина тут простая — так как в текстовых файлах могут выполняться

преобразования символов, то может и не быть прямого соответствия между тем, что находится в файле и тем байтом, к которому нужен доступ. Единственный случай, когда надо использовать `fseek()` для текстового файла — это доступ к той позиции, которая была уже найдена с помощью `ftell()`; такой доступ выполняется с помощью макроса `SEEK_SET`, используемого в качестве начала отсчета.

Хорошо помните следующее: даже если в файле находится один только текст, все равно этот файл при необходимости можно открыть и в двоичном режиме. Никакие ограничения, связанные с тем, что файлы содержат текст, к операциям прямого доступа не относятся. Эти ограничения относятся только к файлам, открытым *в текстовом режиме*.

2.Генерирование исключеный

Если возникает необходимость снова сгенерировать исключения из блока, который обрабатывает исключения, можно сделать это путем вызова throw без указания исключения. В результате текущее исключение будет передано во внешнюю последовательность try/catch обработки исключений. Причиной для этого может послужить желание обрабатывать исключения несколькими обработчиками. Например, один обработчик может заниматься одним аспектом исключения, а второй обработчик — другим. Исключение может быть снова сгенерировано или изнутри блока catch, или из функции, вызванной в этом блоке. Когда повторно генерируется исключение, оно не будет перехвачено той же самой инструкцией catch. Оно будет распространяться до следующей внешней инструкции catch. Следующая программа иллюстрирует повторную генерацию исключения. В ней повторно генерируется исключение типа char*.

```
// пример повторной генерации исключения
#include <iostream.h>
void Xhandler()
{
try {
throw "hello"; // генерация char *
}
catch (char *) { // перехват char *
cout << "Caught char * inside Xhandler\n";
throw; // повторная генерация char * извне функции
}
}
int main()
{
cout << "Start\n";
try{
```

```
Xhandler();
}
catch(char *) {
cout << "Caught char * inside main\n";
}
cout << "End";
return 0;
}
```

Эта программа выдаст на экран следующий текст:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

3.Перехват всех исключений

В определенных обстоятельствах может потребоваться перехватывать все исключения, а не какой- то конкретный тип. Для этого достаточно использовать следующую форму инструкции catch:

```
catch (...) {
// обработка всех исключений
}
```

Здесь многоточие соответствует любому типу данных.

Следующая программа иллюстрирует использование catch (...):

```
// данный пример перехватывает все исключения
#include <iostream.h>
void Xhandler(int test)
{
try{
if(test==0) throw test; // генерация int
if(test==1) throw 'a'; // генерация char
if(test==2) throw 123.23; // генерация double
}
catch (...) { // перехват всех исключений
cout << "Caught One!\n";
}
}
int main()
{
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
```

```
cout << "End";
return 0;
}
```

Программа выведет на экран следующий текст:

```
Start
Caught One!
Caught One!
Caught One!
End
```

Как можно видеть, три инструкции throw были перехвачены с использованием одной инструкции catch.

4.Работа с конструкцией TRY…CATCH

При работе с конструкцией TRY…CATCH пользуйтесь следующими правилами и предложениями.

- Каждая конструкция TRY…CATCH должна находиться в одном пакете, хранимой процедуре или триггере. Например, если блок TRY размещен в одном пакете, размещение блока CATCH в другом пакете не допускается. При выполнении следующего скрипта будет выдано сообщение об ошибке:
- BEGIN TRY
- SELECT *
- FROM sys.messages
- WHERE message_id = 21;
- END TRY
- GO
- -- The previous GO breaks the script into two batches,
- -- generating syntax errors. The script runs if this GO
- -- is removed.
- BEGIN CATCH
- SELECT ERROR_NUMBER() AS ErrorNumber;
- END CATCH;
- GO
- За блоком TRY сразу же должен следовать блок CATCH.
- Конструкция TRY...CATCH может быть вложенной. Это означает, что конструкция TRY…CATCH может быть размещена внутри других блоков TRY и CATCH. При возникновении ошибки внутри вложенного блока TRY управление программой передается блоку CATCH, связанному с вложенным блоком TRY.
- Для обработки ошибки внутри данного блока CATCH нужно написать конструкцию TRY...CATCH внутри этого блока CATCH.
- Ошибки с уровнем серьезности, большим или равным 20, заставляющие компонент Database Engine закрыть соединение, не

могут быть обработаны конструкцией TRY…CATCH. Однако конструкция TRY…CATCH обеспечивает обработку ошибок с уровнем серьезности, большим или равным 20, если соединение не закрывается.

- Ошибки с уровнем серьезности, меньшим или равным 10, рассматриваются как предупреждения или информационные сообщения и не обрабатываются с помощью конструкции TRY…CATCH.
- Сообщения об ошибке завершают пакет, даже если этот пакет находится внутри области конструкции TRY…CATCH. Это относится и к вызовам, направляемым координатором распределенных транзакций Майкрософт (MS DTC) при ошибке распределенной транзакции. Координатор распределенных транзакций (Microsoft) осуществляет управление распределенными транзакциями.

Функции ошибок

В конструкции TRY…CATCH для сбора сведений об ошибках используются следующие функции.

- ERROR_NUMBER() возвращает номер ошибки.
- Функция ERROR_MESSAGE() возвращает полный текст сообщения об ошибке. Текст содержит значения подставляемых параметров, таких как длина, имена объектов или время.
- ERROR_SEVERITY() возвращает уровень серьезности ошибки.
- Функция ERROR_STATE() возвращает код состояния ошибки.
- Функция ERROR_LINE() возвращает номер строки, которая вызвала ошибку, внутри подпрограммы.
- Функция ERROR_PROCEDURE() возвращает имя хранимой процедуры или триггера, в котором произошла ошибка.

Сведения об ошибках извлекаются с помощью этих функций из любого источника в области блока CATCH конструкции TRY…CATCH. Функции ошибок возвращают значение NULL, если они вызываются вне области блока CATCH. Ссылки на функции ошибок могут содержаться внутри хранимой процедуры. Эти функции можно использовать для получения сведений об ошибках при выполнении хранимой процедуры внутри блока CATCH. В этом случае повторять код обработки ошибок в каждом блоке CATCH нет необходимости. В следующем примере кода инструкция SELECT в блоке TRY формирует ошибку деления на 0. Ошибка обрабатывается блоком CATCH, который возвращает сведения об ошибке с помощью хранимой процедуры.

```
USE AdventureWorks2008R2;
GO

-- Verify that the stored procedure does not exist.
IF OBJECT_ID ('usp_GetErrorInfo', 'P') IS NOT NULL
   DROP PROCEDURE usp_GetErrorInfo;
GO
```

```
-- Create a procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_SEVERITY() AS ErrorSeverity,
      ERROR_STATE() as ErrorState,
      ERROR_PROCEDURE() as ErrorProcedure,
      ERROR_LINE() as ErrorLine,
      ERROR_MESSAGE() as ErrorMessage;
GO

BEGIN TRY
   -- Generate divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   -- Execute the error retrieval routine.
   EXECUTE usp_GetErrorInfo;
END CATCH;
GO
```

Ошибки компиляции и ошибки повторной компиляции уровня инструкции

Существует два типа ошибок, не обрабатываемых конструкцией TRY…CATCH в случае, если ошибка происходит на том же уровне выполнения, что и конструкция TRY…CATCH.

- Ошибки компиляции, такие как синтаксические ошибки, препятствующие выполнению пакета.
- Ошибки, происходящие во время повторной компиляции уровня инструкций, такие как ошибки разрешения имен объектов, которые происходят после компиляции из-за отложенного разрешения имен.

Если содержащий конструкцию TRY…CATCH пакет, хранимая процедура или триггер формирует одну из данных ошибок, конструкция TRY…CATCH не обрабатывает эти ошибки. Данные ошибки возвращаются приложению или пакету, который вызвал завершившуюся ошибкой процедуру. Например, в следующем примере кода показана инструкция SELECT, вызывающая синтаксическую ошибку. При выполнении этого примера в редакторе запросов среды Среда SQL Server Management Studio выполнение не будет начато из-за невозможности компиляции пакета. Ошибка будет возвращена в редактор запросов и не будет выявлена конструкцией TRY…CATCH.

```
USE AdventureWorks2008R2;
GO

BEGIN TRY
```

```
   -- This PRINT statement will not run because the batch
   -- does not begin execution.
   PRINT N'Starting execution';

   -- This SELECT statement contains a syntax error that
   -- stops the batch from compiling successfully.
   SELECT ** FROM HumanResources.Employee;
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

В отличие от синтаксической ошибки, приведенной в предыдущем примере, ошибка, происходящая при повторной компиляции на уровне инструкции, не будет препятствовать компиляции пакета, но она завершит обработку пакета, как только произойдет сбой повторной компиляции данной инструкции. Например, если пакет содержит две инструкции и вторая инструкция ссылается на несуществующую таблицу, благодаря отложенному разрешению имен компиляция пакета завершится успешно и начнется выполнение без привязки несуществующей таблицы к плану запроса до повторной компиляции данной инструкции. При передаче управления инструкции, содержащей ссылку на несуществующую таблицу, выполнение пакета прекращается с ошибкой. Этот тип ошибок не обрабатывается конструкцией TRY…CATCH на том же уровне выполнения, на котором произошла ошибка. Следующий пример демонстрирует эту ситуацию.

```
USE AdventureWorks2008R2;
GO

BEGIN TRY
   -- This PRINT statement will run because the error
   -- occurs at the SELECT statement.
   PRINT N'Starting execution';

   -- This SELECT statement will generate an object name
   -- resolution error because the table does not exist.
   SELECT * FROM NonExistentTable;
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

Возможна обработка ошибок средствами конструкции TRY…CATCH в ходе компиляции или повторной компиляции на уровне инструкций. Для этого вызывающий ошибку код необходимо выполнить в отдельном пакете внутри блока TRY. Например, это можно сделать, поместив данный код в хранимую процедуру или выполнив динамическую инструкцию Transact-SQL с помощью процедуры sp_executesql. В таком случае конструкция TRY…CATCH сможет перехватить ошибку на более высоком уровне исполнения, чем уровень, на котором произошла ошибка. Например, в следующем примере кода включена хранимая процедура, создающая ошибку разрешения имен объектов. Содержащий конструкцию TRY…CATCH пакет выполняется на более высоком уровне, чем хранимая процедура, а ошибка, совершенная на более низком уровне, перехватывается.

```
USE AdventureWorks2008R2;
GO

-- Verify that the stored procedure does not already exist.
IF OBJECT_ID ('usp_MyError', 'P') IS NOT NULL
   DROP PROCEDURE usp_MyError;
GO

CREATE PROCEDURE usp_MyError
AS
   -- This SELECT statement will generate
   -- an object name resolution error.
   SELECT * FROM NonExistentTable;
GO

BEGIN TRY
   -- Run the stored procedure.
   EXECUTE usp_MyError;
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

Нефиксируемые транзакции

Внутри конструкции TRY…CATCH транзакции могут переходить в состояние, в котором транзакция остается открытой, но не может быть зафиксирована. Транзакция не может выполнять действия, влекущие за собой запись в журнал транзакций, такие как изменение данных или попытка отката до точки сохранения. Однако в этом состоянии полученные транзакцией блокировки сохраняются, а соединение тоже остается открытым. Результаты выполнения транзакции не будут отменены до тех

пор, пока не будет выполнена инструкция ROLLBACK или пока не завершится выполнение пакета, после чего компонентом Database Engine будет выполнен автоматический откат транзакции. Если при переходе транзакции в нефиксируемое состояние не было отправлено сообщения об ошибке, оно будет отправлено клиентскому приложению после завершения выполнения пакета. В сообщении будет указано, что была обнаружена нефиксируемая транзакция и выполнен ее откат.
Транзакция переходит в нефиксируемое состояние, если внутри блока TRY происходит ошибка, которая в других обстоятельствах завершила бы эту транзакцию. Например, в результате большинства ошибок, вызываемых инструкциями языка DDL, такими как CREATE TABLE, и большинства ошибок, возникающих, когда значение параметра SET XACT_ABORT установлено в ON, транзакции вне блока TRY завершаются, а внутри блока TRY переходят в нефиксируемое состояние.
Код внутри блока CATCH должен проверять состояние транзакции с помощью функции XACT_STATE. Функция XACT_STATE возвращает значение -1, если в сеансе имеется нефиксируемая транзакция. Блок CATCH не должен выполнять действий, которые приведут к записям в журнал, если параметр XACT_STATE возвращает значение -1. В следующем примере кода формируется ошибка инструкции DDL и выполняется проверка состояния транзакции с помощью параметра XACT_STATE для определения наиболее приемлемых действий.

```
USE AdventureWorks2008R2;
GO

-- Verify that the table does not exist.
IF OBJECT_ID (N'my_books', N'U') IS NOT NULL
   DROP TABLE my_books;
GO

-- Create table my_books.
CREATE TABLE my_books
   (
   Isbn       int PRIMARY KEY,
   Title      NVARCHAR(100)
   );
GO

BEGIN TRY
   BEGIN TRANSACTION;
      -- This statement will generate an error because the
      -- column author does not exist in the table.
      ALTER TABLE my_books
         DROP COLUMN author;
   -- If the DDL statement succeeds, commit the transaction.
```

```
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() as ErrorNumber,
        ERROR_MESSAGE() as ErrorMessage;

    -- Test XACT_STATE for 1 or -1.
    -- XACT_STATE = 0 means there is no transaction and
    -- a commit or rollback operation would generate an error.

    -- Test whether the transaction is uncommittable.
    IF (XACT_STATE()) = -1
    BEGIN
        PRINT
            N'The transaction is in an uncommittable state. ' +
            'Rolling back transaction.'
        ROLLBACK TRANSACTION;
    END;

    -- Test whether the transaction is active and valid.
    IF (XACT_STATE()) = 1
    BEGIN
        PRINT
            N'The transaction is committable. ' +
            'Committing transaction.'
        COMMIT TRANSACTION;
    END;
END CATCH;
GO
```

Обработка взаимоблокировок

Конструкция TRY…CATCH может быть использована для обработки взаимоблокировок. Ошибка 1205 жертвы взаимоблокировки может быть перехвачена блоком CATCH, и для данной транзакции может быть выполнен откат до момента разблокирования потоков. Дополнительные сведения о взаимоблокировках см. в разделе Взаимоблокировка.

В следующем примере показано, как конструкция TRY…CATCH может использоваться для обработки взаимоблокировок. В этом первом разделе создается таблица, которая будет использоваться для демонстрации состояния взаимоблокировки, и хранимая процедура, с помощью которой будут выводиться на печать сведения об ошибке.

```
USE AdventureWorks2008R2;
GO

-- Verify that the table does not exist.
```

```
IF OBJECT_ID (N'my_sales',N'U') IS NOT NULL
   DROP TABLE my_sales;
GO

-- Create and populate the table for deadlock simulation.
CREATE TABLE my_sales
   (
   Itemid      INT PRIMARY KEY,
   Sales       INT not null
   );
GO

INSERT my_sales (itemid, sales) VALUES (1, 1);
INSERT my_sales (itemid, sales) VALUES (2, 1);
GO

-- Verify that the stored procedure for error printing
-- does not exist.
IF OBJECT_ID (N'usp_MyErrorLog',N'P') IS NOT NULL
   DROP PROCEDURE usp_MyErrorLog;
GO

-- Create a stored procedure for printing error information.
CREATE PROCEDURE usp_MyErrorLog
AS
   PRINT
      'Error ' + CONVERT(VARCHAR(50), ERROR_NUMBER()) +
      ', Severity ' + CONVERT(VARCHAR(5), ERROR_SEVERITY()) +
      ', State ' + CONVERT(VARCHAR(5), ERROR_STATE()) +
      ', Line ' + CONVERT(VARCHAR(5), ERROR_LINE());
   PRINT
      ERROR_MESSAGE();
GO
```

Следующие сценарии кода для сеанса 1 и сеанса 2 выполняются одновременно с использованием двух отдельных соединений среды Среда SQL Server Management Studio. В ходе обоих сеансов предпринимаются попытки обновить одни и те же строки таблицы. Один из сеансов успешно проведет операцию обновления с первой попытки, а второй будет выбран жертвой взаимоблокировки. Ошибка жертвы взаимоблокировки приведет к тому, что управление внезапно будет передано блоку CATCH и транзакция перейдет в нефиксируемое состояние. Внутри блока CATCH жертва взаимоблокировки может выполнить откат транзакции и вновь пытаться обновить таблицу до тех пор, пока процесс обновления не завершится успешно или пока не будет исчерпан лимит попыток. К прекращению

попыток обновления таблицы приведет то из перечисленных событий, которое наступит первым.

Сеанс 1	**Сеанс 2**

USE AdventureWorks2008R2;
GO
 	
-- Declare and set variable
-- to track number of retries
-- to try before exiting.
DECLARE @retry INT;
SET @retry = 5;
 	
-- Keep trying to update
-- table if this task is
-- selected as the deadlock
-- victim.
WHILE (@retry > 0)
BEGIN
  BEGIN TRY
  BEGIN TRANSACTION;
 	
  UPDATE my_sales
  SET sales = sales + 1
  WHERE itemid = 1;
 	
  WAITFOR DELAY '00:00:13';
 	
  UPDATE my_sales
  SET sales = sales + 1
  WHERE itemid = 2;
 	
  SET @retry = 0;
 	
  COMMIT TRANSACTION;
  END TRY
  BEGIN CATCH
  -- Check error number.
  -- If deadlock victim error,
  -- then reduce retry count
  -- for next update retry.
  -- If some other error
  -- occurred, then exit
  -- retry WHILE loop.
	USE AdventureWorks2008R2;
	GO
	
	-- Declare and set variable
	-- to track number of retries
	-- to try before exiting.
	DECLARE @retry INT;
	SET @retry = 5;
	
	--Keep trying to update
	-- table if this task is
	-- selected as the deadlock
	-- victim.
	WHILE (@retry > 0)
	BEGIN
	  BEGIN TRY
	  BEGIN TRANSACTION;
	
	  UPDATE my_sales
	  SET sales = sales + 1
	  WHERE itemid = 2;
	
	  WAITFOR DELAY '00:00:07';
	
	  UPDATE my_sales
	  SET sales = sales + 1
	  WHERE itemid = 1;
	
	  SET @retry = 0;
	
	  COMMIT TRANSACTION;
	  END TRY
	  BEGIN CATCH
	  -- Check error number.
	  -- If deadlock victim error,
	  -- then reduce retry count
	  -- for next update retry.
	  -- If some other error
	  -- occurred, then exit
	  -- retry WHILE loop.

```
      IF (ERROR_NUMBER() =
1205)
         SET @retry = @retry - 1;
      ELSE
         SET @retry = -1;

      -- Print error information.
      EXECUTE usp_MyErrorLog;

      IF XACT_STATE() <> 0
         ROLLBACK
TRANSACTION;
   END CATCH;
END; -- End WHILE loop.
GO
```

```
      IF (ERROR_NUMBER() =
1205)
         SET @retry = @retry - 1;
      ELSE
         SET @retry = -1;

      -- Print error information.
      EXECUTE usp_MyErrorLog;

      IF XACT_STATE() <> 0
         ROLLBACK
TRANSACTION;
   END CATCH;
END; -- End WHILE loop.
GO
```

Использование конструкции TRY…CATCH с инструкцией RAISERROR

Инструкция RAISERROR может быть использована как в блоке TRY, так и в блоке CATCH конструкции TRY…CATCH для управления процессом обработки ошибок.

При выполнении инструкции RAISERROR с уровнем серьезности от 11 до 19 внутри блока TRY управление переходит к связанному с ним блоку CATCH.

При выполнении инструкции RAISERROR с уровнем серьезности от 11 до 19 внутри блока CATCH инструкция возвращает вызывающему приложению или пакету сообщение об ошибке. Таким образом, инструкция RAISERROR может быть использована для возвращения вызывающему объекту сведений об ошибке, приведшей к выполнению блока CATCH. Сведения об ошибках, предоставляемые функциями ошибок конструкции TRY…CATCH, могут быть включены в сообщение инструкции RAISERROR. В них может быть включен исходный номер ошибки, однако номер ошибки для инструкции RAISERROR должен быть >= 50000.

При выполнении инструкции RAISERROR с уровнем серьезности, меньшим или равным 10, она возвращает вызывающему пакету или приложению информационное сообщение без вызова блока CATCH.

Инструкция RAISERROR с уровнем серьезности, большим или равным 20, закрывает соединение с базой данных без вызова блока CATCH.

Следующий пример кода иллюстрирует возможное использование инструкции RAISERROR внутри блокаCATCH для возвращения исходных сведений об ошибке вызывающему приложению или пакету. Инструкция DELETE, выполняемая хранимой процедурой usp_GenerateError внутри блока TRY, вызывает ошибку нарушения ограничения. Эта ошибка вызывает передачу управления связанному блоку CATCHвнутри процедуры usp_GenerateError, в которой выполняется хранимая процедура usp_RethrowError, с целью вывода сведений об ошибке нарушения ограничения при помощи

инструкции RAISERROR. Эта созданная инструкцией RAISERROR ошибка возвращается вызывающему пакету, в котором выполнялась хранимая процедура usp_GenerateError, и передает управление связанному блоку CATCH в вызывающем пакете.

Примечание
Инструкция RAISERROR может формировать только ошибки с состоянием от 1 до 127 включительно. Поскольку компонентом Database Engine могут выводиться ошибки, имеющие состояние 0, рекомендуется проверять состояние ошибки с помощью функции ERROR_STATE перед его передачей в качестве значения параметру состояния инструкции RAISERROR.

```
USE AdventureWorks2008R2;
GO

-- Verify that stored procedure does not exist.
IF OBJECT_ID (N'usp_RethrowError',N'P') IS NOT NULL
   DROP PROCEDURE usp_RethrowError;
GO

-- Create the stored procedure to generate an error using
-- RAISERROR. The original error information is used to
-- construct the msg_str for RAISERROR.
CREATE PROCEDURE usp_RethrowError AS
   -- Return if there is no error information to retrieve.
   IF ERROR_NUMBER() IS NULL
      RETURN;

   DECLARE
      @ErrorMessage    NVARCHAR(4000),
      @ErrorNumber     INT,
      @ErrorSeverity   INT,
      @ErrorState      INT,
      @ErrorLine       INT,
      @ErrorProcedure  NVARCHAR(200);

   -- Assign variables to error-handling functions that
   -- capture information for RAISERROR.
   SELECT
      @ErrorNumber = ERROR_NUMBER(),
      @ErrorSeverity = ERROR_SEVERITY(),
      @ErrorState = ERROR_STATE(),
      @ErrorLine = ERROR_LINE(),
```

```
        @ErrorProcedure = ISNULL(ERROR_PROCEDURE(), '-');

    -- Build the message string that will contain original
    -- error information.
    SELECT @ErrorMessage =
        N'Error %d, Level %d, State %d, Procedure %s, Line %d, ' +
            'Message: '+ ERROR_MESSAGE();

    -- Raise an error: msg_str parameter of RAISERROR will contain
    -- the original error information.
    RAISERROR
        (
        @ErrorMessage,
        @ErrorSeverity,
        1,
        @ErrorNumber,    -- parameter: original error number.
        @ErrorSeverity,  -- parameter: original error severity.
        @ErrorState,     -- parameter: original error state.
        @ErrorProcedure, -- parameter: original error procedure name.
        @ErrorLine       -- parameter: original error line number.
        );
GO

-- Verify that stored procedure does not exist.
IF OBJECT_ID (N'usp_GenerateError',N'P') IS NOT NULL
    DROP PROCEDURE usp_GenerateError;
GO

-- Create a stored procedure that generates a constraint violation
-- error. The error is caught by the CATCH block where it is
-- raised again by executing usp_RethrowError.
CREATE PROCEDURE usp_GenerateError
AS
    BEGIN TRY
        -- A FOREIGN KEY constraint exists on the table. This
        -- statement will generate a constraint violation error.
        DELETE FROM Production.Product
            WHERE ProductID = 980;
    END TRY
    BEGIN CATCH
        -- Call the procedure to raise the original error.
        EXEC usp_RethrowError;
    END CATCH;
GO
```

```
-- In the following batch, an error occurs inside
-- usp_GenerateError that invokes the CATCH block in
-- usp_GenerateError. RAISERROR inside this CATCH block
-- generates an error that invokes the outer CATCH
-- block in the calling batch.
BEGIN TRY  -- outer TRY
   -- Call the procedure to generate an error.
   EXECUTE usp_GenerateError;
END TRY
BEGIN CATCH  -- Outer CATCH
   SELECT
      ERROR_NUMBER() as ErrorNumber,
      ERROR_MESSAGE() as ErrorMessage;
END CATCH;
GO
```

Изменение хода выполнения

Для изменения хода выполнения можно использовать инструкцию GOTO внутри блока TRY или блока CATCH. Инструкцию GOTO можно также использовать для выхода из блока TRY или из блока CATCH; однако GOTO нельзя использовать для входа в блок TRY или в блок CATCH.

Решение обработки ошибок в образце базы данных AdventureWorks2008R2

В состав образца базы данных База данных AdventureWorks2008R2 входит решение обработки ошибок, предназначенное для регистрации сведений об ошибках, перехваченных блоком CATCH конструкции TRY…CATCH, в отношении которых в дальнейшем могут выполняться запросы и осуществляться анализ.

Таблица dbo.ErrorLog

В таблицу ErrorLog заносятся сведения об ошибках, в том числе номер ошибки, уровень серьезности ошибки, состояние ошибки, имя хранимой процедуры или триггера, в котором произошла ошибка, номер строки, в которой произошла ошибка, а также полный текст сообщения об ошибке. Кроме того, в нее записываются дата и время, в которое была совершена ошибка, а также имя пользователя, который выполнял вызвавшую ошибку процедуру. Эта таблица заполняется в ходе выполнения хранимой процедуры uspLogError внутри области действия блока CATCH конструкции TRY…CATCH.

dbo.uspLogError

Хранимая процедура uspLogError заносит в таблицу ErrorLog сведения об ошибке, вызвавшей передачу управления блоку CATCH конструкции TRY…CATCH. Чтобы процедура uspLogError могла занести в таблицу ErrorLog сведения об ошибке, должны быть выполнены следующие условия.

- Процедура uspLogError должна выполняться в области блока CATCH.
- Если текущая транзакция находится в нефиксируемом состоянии, необходимо выполнить ее откат до выполнения процедуры uspLogError.

Выходной параметр @ErrorLogID процедуры uspLogError должен возвращать идентификатор ErrorLogID строки, введенной процедурой uspLogError в таблицу ErrorLog. Значение параметра @ErrorLogID по умолчанию равно 0. В следующем примере показан код процедуры uspLogError.

```
CREATE PROCEDURE [dbo].[uspLogError]
   @ErrorLogID [int] = 0 OUTPUT  -- Contains the ErrorLogID of the row inserted
                                 -- by uspLogError in the ErrorLog table.

AS
BEGIN
   SET NOCOUNT ON;

   -- Output parameter value of 0 indicates that error
   -- information was not logged.
   SET @ErrorLogID = 0;

   BEGIN TRY
      -- Return if there is no error information to log.
      IF ERROR_NUMBER() IS NULL
         RETURN;

      -- Return if inside an uncommittable transaction.
      -- Data insertion/modification is not allowed when
      -- a transaction is in an uncommittable state.
      IF XACT_STATE() = -1
      BEGIN
         PRINT 'Cannot log error since the current transaction is in an uncommittable state. '
            + 'Rollback the transaction before executing uspLogError in order to successfully log error information.';
         RETURN;
      END;

      INSERT [dbo].[ErrorLog]
         (
         [UserName],
         [ErrorNumber],
         [ErrorSeverity],
         [ErrorState],
         [ErrorProcedure],
         [ErrorLine],
         [ErrorMessage]
         )
```

```
        VALUES
          (
          CONVERT(sysname, CURRENT_USER),
          ERROR_NUMBER(),
          ERROR_SEVERITY(),
          ERROR_STATE(),
          ERROR_PROCEDURE(),
          ERROR_LINE(),
          ERROR_MESSAGE()
          );

        -- Pass back the ErrorLogID of the row inserted
        SELECT @ErrorLogID = @@IDENTITY;
    END TRY
    BEGIN CATCH
        PRINT 'An error occurred in stored procedure uspLogError: ';
        EXECUTE [dbo].[uspPrintError];
        RETURN -1;
    END CATCH
END;
```

dbo.uspPrintError

Хранимая процедура uspPrintError выводит на печать сведения об ошибке, вызвавшей передачу управления блоку CATCH конструкции TRY…CATCH. Процедуру **uspPrintError** следует выполнять в области блока CATCH, иначе она не выполняет распечатку сведений об ошибке. В следующем примере показан код процедуры uspPrintError.

```
CREATE PROCEDURE [dbo].[uspPrintError]
AS
BEGIN
    SET NOCOUNT ON;

    -- Print error information.
    PRINT 'Error ' + CONVERT(varchar(50), ERROR_NUMBER()) +
          ', Severity ' + CONVERT(varchar(5), ERROR_SEVERITY()) +
          ', State ' + CONVERT(varchar(5), ERROR_STATE()) +
          ', Procedure ' + ISNULL(ERROR_PROCEDURE(), '-') +
          ', Line ' + CONVERT(varchar(5), ERROR_LINE());
    PRINT ERROR_MESSAGE();
END;
```

Пример обработки ошибки

Следующий пример демонстрирует решение обработки ошибок База данных AdventureWorks2008R2. Код внутри блока TRY пытается удалить из таблицы Production.Product запись с кодом ProductID 980. Ограничение FOREIGN KEY для таблицы препятствует успешному выполнению операции DELETE. Возникает ошибка нарушения ограничения. В результате

этой ошибки управление передается блокуCATCH. Внутри блока CATCH выполняются следующие действия.

- Процедура uspPrintError выводит на печать сведения об ошибке.
- После выполнения отката транзакции процедура uspLogError вводит сведения об ошибке в таблицу ErrorLog и возвращает идентификатор ErrorLogID вставленной строки в параметр@ErrorLogID OUTPUT.

```
USE AdventureWorks2008R2;
GO

-- Variable to store ErrorLogID value of the row
-- inserted in the ErrorLog table by uspLogError
DECLARE @ErrorLogID INT;

BEGIN TRY
   BEGIN TRANSACTION;

   -- A FOREIGN KEY constraint exists on this table. This
   -- statement will generate a constraint violation error.
   DELETE FROM Production.Product
      WHERE ProductID = 980;

   -- If the delete operation succeeds, commit the transaction.
   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   -- Call procedure to print error information.
   EXECUTE dbo.uspPrintError;

   -- Roll back any active or uncommittable transactions before
   -- inserting information in the ErrorLog.
   IF XACT_STATE() <> 0
   BEGIN
      ROLLBACK TRANSACTION;
   END

   EXECUTE dbo.uspLogError @ErrorLogID = @ErrorLogID OUTPUT;
END CATCH;

-- Retrieve logged error information.
SELECT * FROM dbo.ErrorLog WHERE ErrorLogID = @ErrorLogID;
GO
```

Пример вложенной обработки ошибок

Следующий пример демонстрирует использование вложенной конструкции TRY…CATCH.

```
BEGIN TRY
   BEGIN TRY
      SELECT CAST('invalid_date' AS datetime)
   END TRY
   BEGIN CATCH
      PRINT 'Inner TRY error number: ' +
         CONVERT(varchar,ERROR_NUMBER()) + ' on line: ' +
         CONVERT(varchar, ERROR_LINE())
   END CATCH
   SELECT CAST('invalid_int' AS int)
END TRY
BEGIN CATCH
   PRINT 'Outer TRY error mumber: ' +
CONVERT(varchar,ERROR_NUMBER())+
         ' on line: ' + CONVERT(varchar, ERROR_LINE())
END CATCH
```

ЛЕКЦИЯ №9

Тема: Динамические информационные структуры данных*:* списки, стеки, очереди.**(2ч.)**

Цель занятия. Ознакомится со списками, стеками, понятием бинарные деревья.

План лекции:

1. Линейные списки
2. Стеки
3. Очереди
4. Бинарные деревья

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память следует распределять во время выполнения программы по мере необходимости отдельными блоками. Блоки связываются друг с другом с помощью указателей. Такой способ организации данных называется *динамической структурой данных*, поскольку она размещается в динамической памяти и ее размер изменяется во время выполнения программы.

Из динамических структур в программах чаще всего используются *линейные списки, стеки, очереди и бинарные деревья.* Они различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура, в отличие от массива или записи, может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки.

Элемент любой динамической структуры состоит из двух частей: *информационной,* ради хранения которой и создается структура, и *указателей,* обеспечивающих связь элементов друг с другом.

Элемент динамической структуры описывается в виде записи, например:

```
type
  pnode =  ^node;
  node = record
           d : word;                { информационная }
           s : string;              { часть }
           p : pnode;               { указатель на следующий элемент }
   end;
```

ПРИМЕЧАНИЕ

Обратите внимание, что тип указателя pnode на запись node определен раньше, чем сама запись. Это не противоречит принципу «использование
только после описания», поскольку для описания переменной типа pnode
информации вполне достаточно.

Рассмотрим принципы работы с основными динамическими структурами.

1.Линейные списки

В линейном списке каждый элемент связан со следующим и, возможно, с предыдущим. В первом случае список называется *односвязным,* во втором — *двусвязным*. Если последний элемент связать указателем с первым, получится*кольцевой список.*

Каждый элемент списка содержит *ключ,* идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять следующие операции:

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

Для работы со списком в программе требуется определить указатель на его начало. Чтобы упростить добавление новых элементов в конец списка, можно также завести указатель на конец списка.

Рассмотрим программу, которая формирует односвязный список из пяти элементов, содержащих число и его текстовое представление, а затем выполняет вставку и удаление заданного элемента. В качестве ключа используется число.

```
program linked_list;
const n = 5;
```

```
type        pnode = ^node;
            node = record
{ элемент списка }
                        d : word;
                        s : string;
                        p : pnode;
            end;
var         beg                 : pnode;                        { указатель на
начало списка }
  i, key    : word;
  s                     : string;
  option    : word;
const       text: array [1 .. n] of string =
                        ('one', 'two', 'three', 'four', 'five');

{ ---------- добавление элемента в конец списка ----- }
procedure add(var beg : pnode; d : word; const s : string);
var p : pnode;                          { указатель на создаваемый элемент }
  t : pnode;                                    { указатель для просмотра
списка }
begin
  new(p);                                                                   {
создание элемента }
  p^.d := d; p^.s := s;                                 { заполнение элемента
}
  p^.p := nil;
  if beg = nil then beg := p                                    { список был
пуст }
  else begin
{ список не пуст }
            t := beg;
            while t^.p <> nil do    { проход по списку до конца }
                        t := t^.p;
            t^.p := p;          { привязка нового элемента к последнему }
  end
end;

{ ------------------------ поиск элемента по ключу ----- }
function find(beg : pnode; key : word; var p, pp : pnode) : boolean;
begin
  p := beg;
  while p <> nil do begin
            if p^.d = key then begin
                        find := true; exit end;
            pp := p;
```

```
            p := p^.p;
  end;
  find := false;
end;

{ ------------------------------- вставка элемента ----- }
procedure insert(beg : pnode; key, d : word; const s : string);
var         p          : pnode;                    { указатель на создаваемый
элемент }
  pkey : pnode;              { указатель на искомый элемент }
  pp         : pnode;                  { указатель на предыдущий элемент }
begin
  if not find(beg, key, pkey, pp) then begin
            writeln(' вставка не выполнена'); exit; end;
  new(p);
  p^.d := d; p^.s := s;
  p^.p := pkey^.p;
  pkey^.p := p;
end;

{ ------------------------------- удаление элемента ----- }
procedure del(var beg : pnode; key : word);
var p       : pnode;                    { указатель на удаляемый элемент }
  pp : pnode;                { указатель на предыдущий элемент }
begin
  if not find(beg, key, p, pp) then begin
            writeln(' удаление не выполнено'); exit; end;
  if p = beg then beg := beg^.p      { удаление первого элемента }
  else pp^.p := p^.p;
  dispose(p);
end;

{ ---------------------------------- вывод списка ----- }
procedure print(beg : pnode);
var p : pnode;                          { указатель для просмотра списка }
begin
  p := beg;
  while p <> nil do begin                                    { цикл по
списку }
            writeln(p^.d:3, p^.s);                                { вывод
элемента }
            p := p^.p   { переход к следующему элементу списка }
  end;
end;
```

```
{ ------------------------------ главная программа ----- }
begin
  for i := 1 to 5 do add(beg, i, text[i]);
  while true do begin
        writeln('1 - вставка, 2 - удаление, 3 - вывод, 4 - выход');
        readln(option);
        case option of
              1: begin
                    writeln('Ключ для вставки?');
                    readln(key);
                    writeln('Вставляемый элемент?');
                    readln(i); readln(s);
                    insert(beg, key, i, s);
              end;
              2: begin
                    writeln('Ключ для удаления?');
                    readln(key);
                    del(beg, key);
              end;
              3: begin
                    writeln('Вывод списка:');
                    print(beg);
              end;
              4: exit;
        end
        writeln;
  end
end.
```

Функция поиска элемента find возвращает true, если искомый элемент найден, и false в противном случае. Поскольку одного факта отыскания элемента недостаточно, функция также возвращает через список параметров два указателя: на найденный элемент p и на предшествующий ему pp. Последний требуется при удалении элемента из списка.

Если элемента с заданным ключом в списке нет, цикл просмотра списка завершится естественным образом, поскольку последний элемент списка содержит nil в поле p указателя на следующий элемент.

Вставка элемента выполняется после элемента с заданным ключом (процедура insert). Под новый элемент выделяется место в динамической памяти (оператор 1), и информационные поля элемента заполняются переданными в процедуру значениями (оператор 2). Новый элемент p вставляется между элементами pkey и следующим за ним (его адрес хранится в pkey^.p). Для этого в операторах 3 и 4 устанавливаются две связи (рис. 5.8).

2.Стеки

Стек является простейшей динамической структурой. Добавление элементов в стек и выборка из него выполняются из одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека.

Говорят, что стек реализует принцип обслуживания LIFO (last in — first out, последним пришел — первым обслужен). Кстати, сегмент стека назван так именно потому, что память под локальные переменные выделяется по принципу LIFO. Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

3.Очереди

Очередь — это динамическая структура данных, добавление элементов в которую выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (first in — first out, первым пришел — первым обслужен). В программировании очереди применяются очень широко — например, при моделировании, буферизованном вводе-выводе или диспетчеризации задач в операционной системе.

4.Бинарные деревья

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

Пример бинарного дерева приведен на рис. 5.9 (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками*. *Высота дерева*определяется количеством уровней, на которых располагаются его узлы.

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

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью рекурсивных алгоритмов. Например, *процедуру обхода всех узлов дерева* можно в общем виде описать так:

```
procedure print_tree( дерево );
begin
  print_tree( левое_поддерево )
  посещение корня
  print_tree( правое_поддерево )
end;
```

Приведенная функция позволяет получить последовательность ключей, отсортированную по возрастанию:

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

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

Для бинарных деревьев определены операции:

- включения узла в дерево;
- поиска по дереву;
- чтение элемента с заданным ключом;
- обхода дерева;

ЛЕКЦИЯ № 10.

Тема: Компоненты библиотек Borland C++ Builder 6: свойства и события компонентов; методы обработки событий. Страницы палитры компонентов: характеристика страниц Standart, Additional, System.(2ч.)

Цель занятия. Познакомить с интерфейсом среды.

План лекции:

1.Интерфейс среды,
2.группы команд меню,
3.Общая характеристика.
4.Палитра визуальных компонентов VCL , характеристика.

Ключевые слова: RAD-среды, окна, форма, интерфейс, визуальные компоненты, standart, additional, Win32,

Среда Borland Bilder 6, панель компонентов, общая характеристика
Цель занятия. Познакомить с интерфейсом среды.
План лекции.
Интерфейс среды, группы команд меню, общая характеристика.
Палитра визуальных компонентов VCL , характеристика.
Ключевые слова: RAD-среды, окна, форма, интерфейс, визуальные компоненты, standart, additional, Win32,

Объектно-ориентированное программирование является основным методом среды.Разработку программы, используя метод ООП, можно разбить на отдельные этапы, а затем соединить их вместе для получения конечной программы. Именно такой метод разработки имеет среда Borland C++ Builder 6 и ему подобные пакеты для создания программ под Windows.
ООП дает возможность писать программу постепенно, шаг за шагом, создавая отдельные небольшие программы (функции-методы) для обработки действий (событий), вызываемых объектами (кнопками и окнами программы, кнопками клавиатуры и т. п.)
Интерфейс среды
После запуска среды разработки окно загрузки сменяется несколькими новыми окнами, за которыми просматривается рабочий стол компьютера. Все эти окна представляют собой интерфейс программы
Borland C++ Builder 6, который изображен на рис. 6.1.

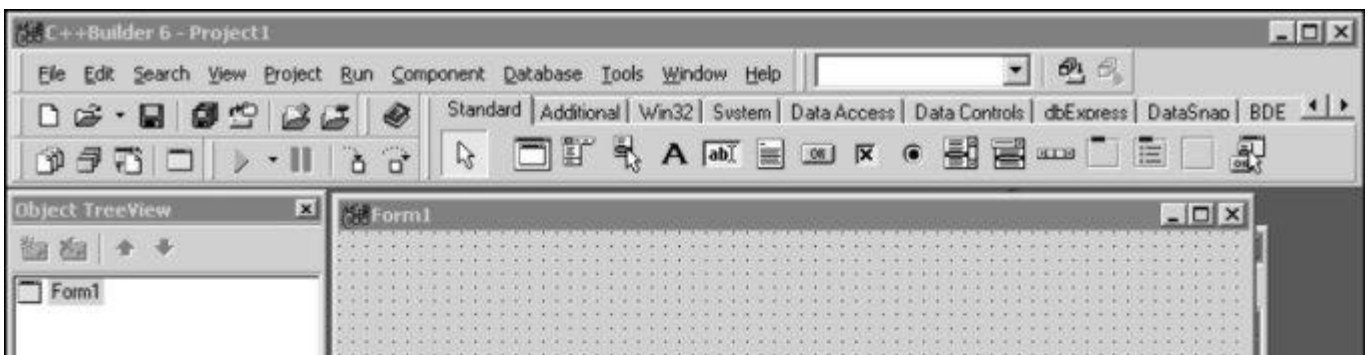


Рис. 3.1. Интерфейс программы Borland C++Builder 6

В терминологии программистов этот интерфейс называется средой быстрой разработки приложений RAD (RapidApplicationDevelopment). Такое название он получил за то, что создание программы в нем сводится к простому конструированию внешнего вида будущей программы из готовых кубиков, а большую часть стандартных и рутинных операций за человека выполняет компьютер. Например, компьютер автоматически создает заготовку текста программы для функций обработки событий.

Самое верхнее главное окно интерфейса (рис. 6.2) имеет заголовок C++Builder 6 - Project 1, который отражает название среды разработки и имя нового проекта, из которого будет получена работающая программа.

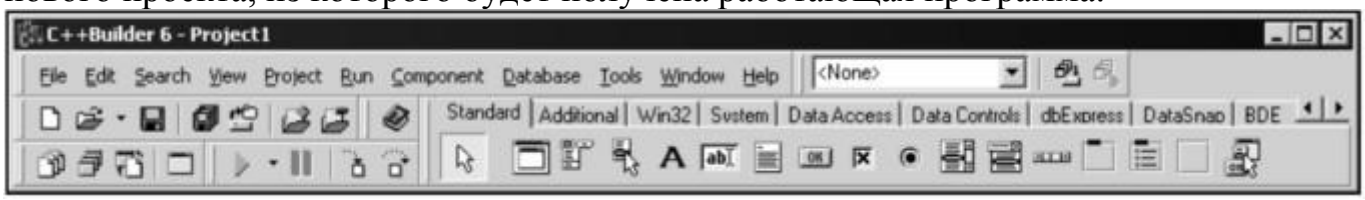


Рис. 6.2. Главное окно интерфейса

Проектом называется вся группа программных файлов, которые необходимы для создания конечной исполняемой программы. Так, например, в состав проекта могут включаться файлы с текстами программ, файл ресурсов с рисунками курсоров и иконок (значков), звуковые файлы и т. п. Первоначально проект хранится в памяти компьютера, и для того чтобы сохранить его на диске, необходимо будет выполнить стандартные операции сохранения, создав при этом отдельную папку. Кроме того, интерфейс сам предложит сохранить проект, если вы решите выйти из программы или попытаетесь создать новый проект.

На строке заголовка проекта находятся кнопки свертывания, восстановления и закрытия окна. Под заголовком размещается строка главного меню, которая предоставляет доступ ко всем функциям и командам среды разработки. Под главным меню располагаются быстрые кнопки, объединенные в группы по назначению. Они позволяют получить быстрый доступ к наиболее часто используемым командам.

Справа от быстрых кнопок расположена палитра визуальных компонентов VCL (VisualComponentLibrary, библиотека визуальных компонентов). Это те самые объекты или программные компоненты, предназначенные для быстрого создания визуальных программ для Windows. Компоненты позволяют быстро создавать в программе различные программные кнопки, рисунки, надписи, таймеры, календари и т. п. Палитра визуальных компонентов состоит из нескольких закладок, на которых располагаются компоненты, распределенные по группам. Именно с помощью этих компонентов можно быстро создавать новые программы.

В центре экрана располагается окно дизайнера форм (рис. 3.3). Это окно будущей программы. Оно содержит строку заголовка, в котором отображается название формы Forml (созданное по умолчанию) и кнопки управления окном. На поле этого окна будут помещаться компоненты VCL в виде программных кнопок, надписей и других элементов будущей программы.

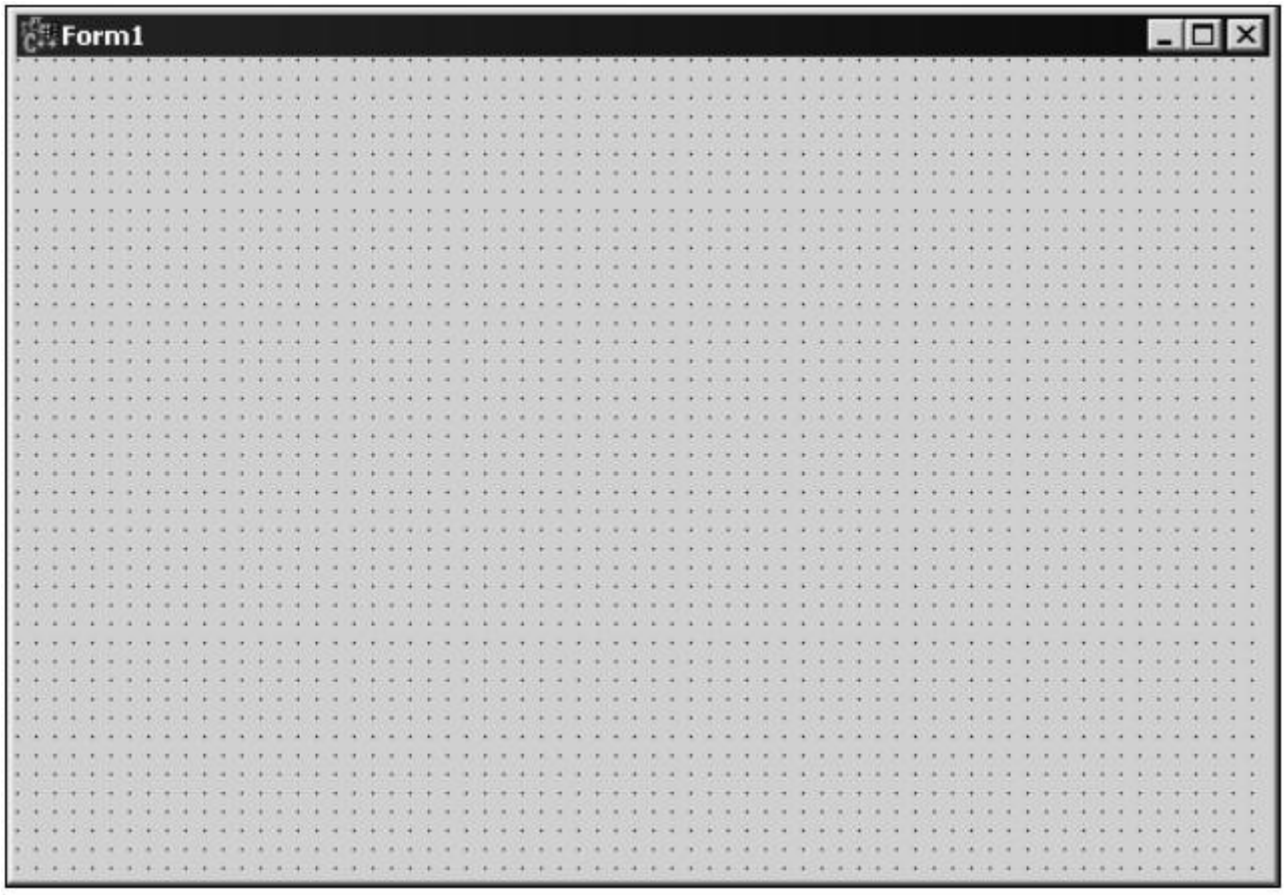


Рис. 6.3. Окно дизайнера форм

Под окном дизайнера форм располагается окно редактора кода (рис. 6.4) с заголовком Unitl.cpp (также созданным по умолчанию), в котором производится набор и редактирование кода (текста) программы. Следующее окно интерфейса располагается в левой нижней части экрана. Это окно инспектора объектов ObjectInspector (рис. 6.5). В этом окне производится настройка основных свойств визуальных компонентов. Расположение окна инспектора объектов в программе не фиксированное и при желании его можно переместить в ту часть рабочей области программы, которая для вас наиболее удобна. Для этого необходимо нажать левую кнопку мыши на строке заголовка окна и, удерживая ее, переместить окно.

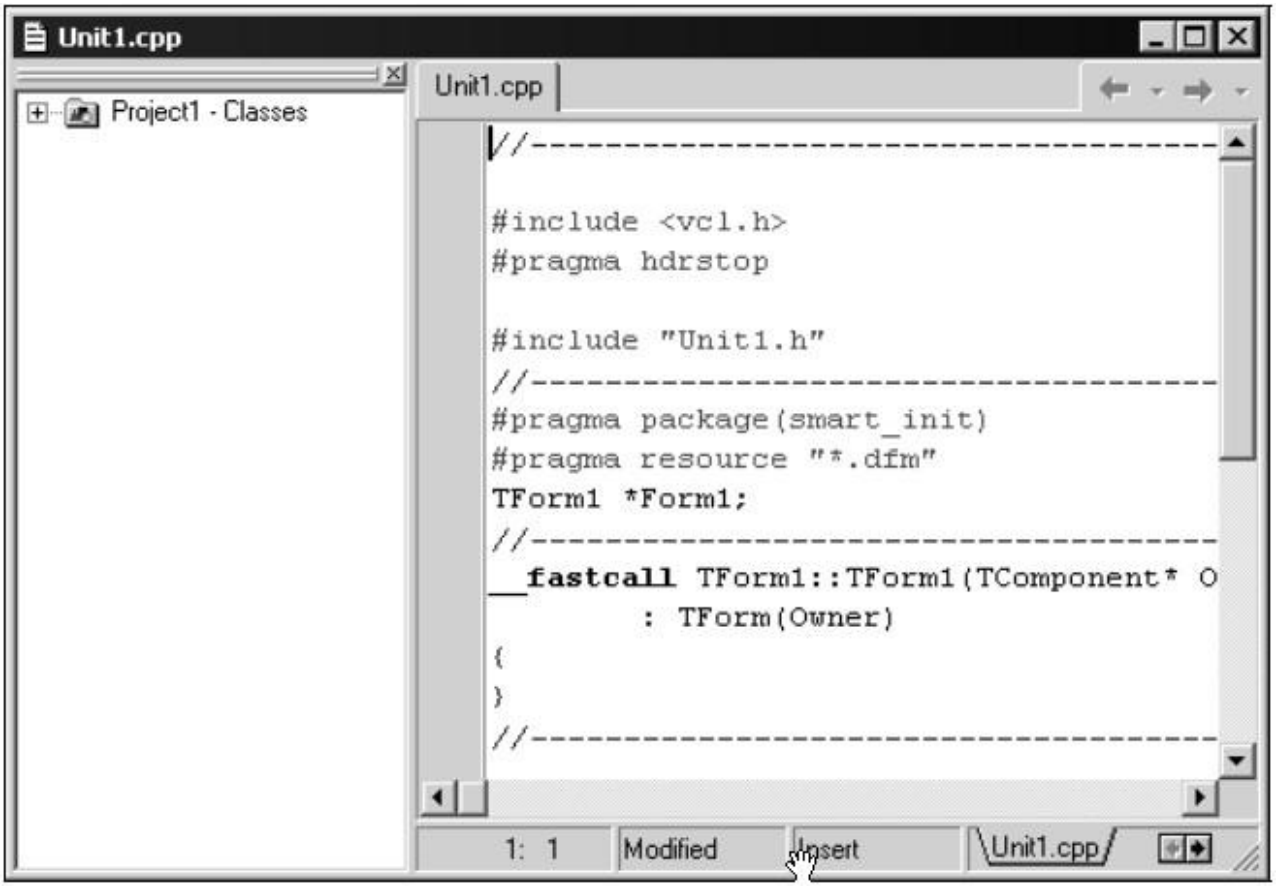


Рис. 6.4. Окно редактора кода

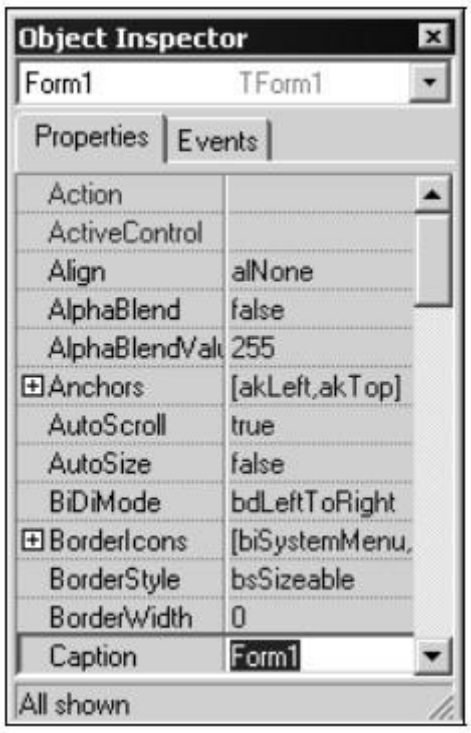


Рис. 6.5. Окно инспектора объектов

Над окном инспектора объектов расположено окно просмотра объектов ObjectTreeView (рис. 6.6). Оно отображает в виде дерева всю структуру проекта, состоящего из форм, кодов (текстов) программ и других ресурсов (файлов).

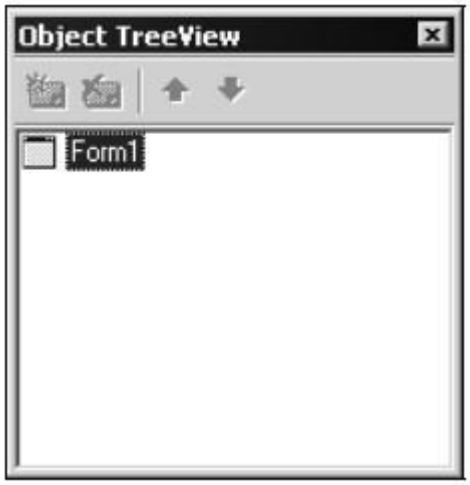


Рис. 6.6. Окно просмотра объектов

Некоторое время надо самостоятельно знакомиться с интерфейсом среды разработки на компьютере, для того чтобы в памяти закрепился его образ. Ведь только желание и живой интерес украшают процесс изучения. Данный интерфейс является самым главным инструментом при создании программ.Приведём описание некоторых команд горизонтального меню (6.1) Первая группа меню команд под названием File (Файл) изображена на рис. 6.7.

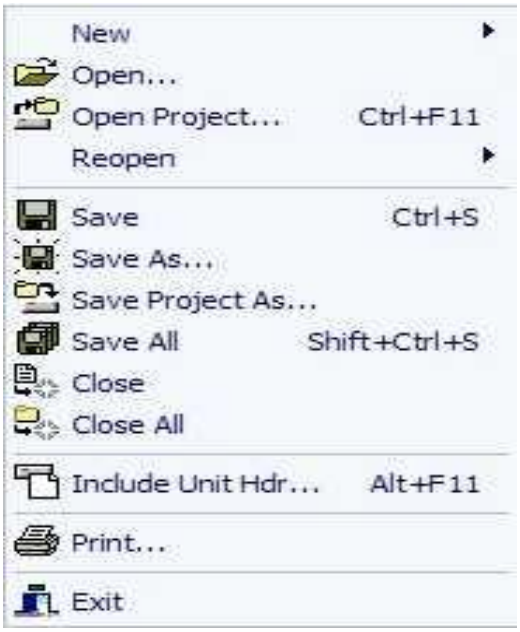
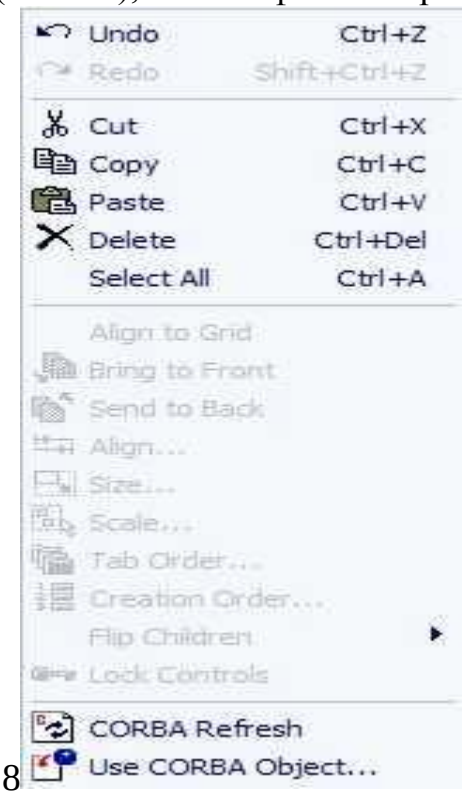


рис. 6.7.

Команды этого меню осуществляют работу с файлами и предоставляют доступ к операциям создания (New) новых форм (окон) и приложений, открытия (Open), сохранения (Save) и закрытия (Close) файлов и проектов, печати текстов программ (Print) и добавления заголовочных модулей (Include), то есть файлов с расширением h.

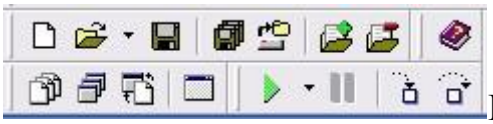


8 рис. 6.8. Вторая по порядку группа команд меню называется Edit (Редактор), она изображена

В этом меню собраны команды редактирования, такие как отмена (Undelete) и повторение (Redo) операций, вырезание (Cut), копирование (Copy), вставка (Paste) и удаление (Delete) фрагментов текста. А также команды выделения

всего текста (SelectAll), выравнивания компонентов (Align) и настройки редактора кода (текста программы

Следующая группа команд Search (Поиск)
Команды данного меню позволяют осуществить поиск текста в файле, продолжить поиск после первого вхождения, произвести автоматическую замену, а также быстро перейти к строке кода, задав ее номер.
Группа команд View
Из этого пункта меню вызываются основные диалоговые окна управления проектом и компонентами, такие как менеджер проектов (ProjectManager), список компонентов (ComponentList) и список окон (WindowList). Также из этого пункта меню открываются все окна отладки программ (DebugWindows), работу с которыми мы рассмотрим позже.
Группа команд Project
В этой группе меню собраны команды управления проектом. С их помощью можно добавлять и удалять модули (файлы с текстами программ), добавлять библиотеку компонентов VCL, откомпилировать проект и т. д.
Группа команд Run
С помощью команд этого меню выполняется запуск и останов программ, запуск программ в непрерывном и пошаговом режимах, добавление переменных для просмотра, установка точек останова и другие действия по отладке программы.
Группа команд Component(Компонент)
Из этого меню вызываются команды добавления в систему новых компонентов и конфигурации их палитры
Быстрые кнопки
Быстрые кнопки расположены рядом с главным меню (рис. 6.9).

 Рис. 6.9. Быстрые кнопки

С их помощью осуществляется быстрый доступ к основным и часто используемым командам главного меню. Все эти кнопки имеют всплывающее меню, появляющееся при наведении на них курсора мыши, и горячие клавиши, показываемые в скобках. Все быстрые кнопки так же, как и главное меню, разделены на группы.
Кнопки Standard (Стандартные)
Первая из этих кнопок позволяет создавать новые объекты программы. Такими объектами могут быть программы, новые формы, файлы, библиотеки и т. п. Для создания нового объекта необходимо нажать кнопку New. При этом откроется диалоговое окно NewItems (рис. 6.2), в котором можно выбрать необходимый для создания программы объект. Это окно имеет множество вкладок, в каждой из которых находятся объекты определенного типа. Для начала работы с Borland C++ Builder 6 достаточно будет

пользоваться лишь несколькими объектами из вкладки New. Другие кнопки этой группы позволяют сохранять (Save) и открывать (Open) как отдельные файлы, так и целые проекты. Замыкают группу две кнопки управления проектами, с помощью которых можно добавлять файлы к проекту (AddFileToProject) и удалять их из него (RemoveFileFromProject).

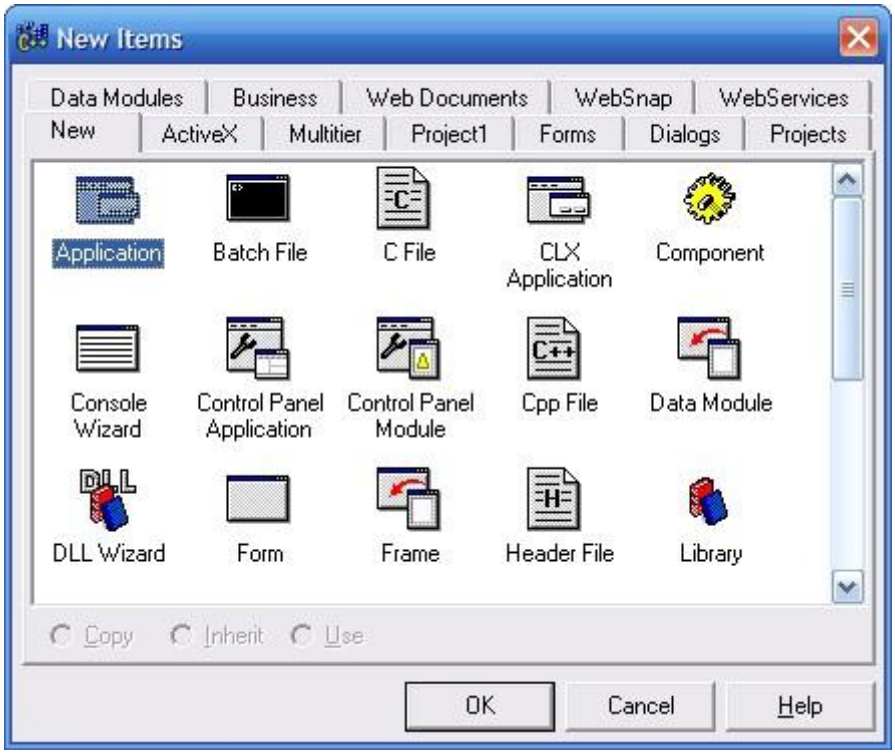


Рис. 6.10. Диалоговое окно NewItems

Обратите внимание на то, что при подведении курсора к любой из кнопок появляется контекстная справка, в которой приведено название кнопки и клавиши для ее активации с клавиатуры. Например, для кнопки сохранения Savetакая справка будет содержать слова Save (Ctrl+S). Это значит, что кнопка сохранения файла может быть активирована с клавиатуры одновременным нажатием клавиш <Ctrl> и <S>. Подобные комбинации клавиш существуют и для других органов управления средой разработки. В дальнейшем мы будем прибегать к помощи этих средств, поскольку при разработке программы встречаются ситуации, когда все окна среды разработки закрыты формой программы, и нажать быструю

Палитра VCL-компонентов

Разработка программ на Borland C++ Builder 6 построена на основе выбора необходимых визуальных компонентов VCL и расположения их на поле форм (окон будущей программы). Компоненты, таким образом, служат кирпичиками, из которых строятся программы. Кстати, Builder в переводе на русский язык означает "строитель". Все компоненты VCL располагаются на палитре, расположенной ниже и правее главного меню. Палитра компонентов состоит из вкладок. Вкладки позволяют разделить большое

число компонентов на группы, близкие по назначению. Щелкая левой кнопкой мыши по вкладкам, можно выбрать необходимую группу компонентов, которая будет отображаться на экране. Например, при щелчке по вкладке Standard на экране появятся компоненты, изображенные на рис. 6.11.

Рис. 6.11. Окно с компонентами

Среди них легко найти компонент Button(Кнопка) с надписью ОК и компонент Label(Этикетка) с надписью A. Иногда часть компонентов не умещается на экране. Для их просмотра существуют кнопки прокрутки в виде треугольных стрелок слева и справа от самих компонентов- Щелкая по ним кнопкой мыши, можно сдвигать компоненты группы влево или вправо. Все компоненты видны на экране в виде иконок. При наведении на них курсора мыши эти иконки приподнимаются, как кнопки, а под курсором появляется строка с надписью, отображающей название данного компонента.
Компоненты Standard
На этой вкладке располагаются компоненты (рис. 18.1), с помощью которых происходит подключение к программе стандартных интерфейсных элементов, имеющихся во всех версиях операционных систем Windows. Рассмотрим эти компоненты по порядку слева направо.
□ Frame (Кадр) — предназначен для создания контейнера (окна) для размещения других компонентов. Данный компонент очень похож на форму Form. Для размещения этого компонента на форме необходимо первый раз создать его с помощью команд File | New | Frame. Именно такое сообщение появляется при попытке размещения этого компонента стандартным образом (рис. 6.13).

Рис. 6.12. Палитра компонентов Standard

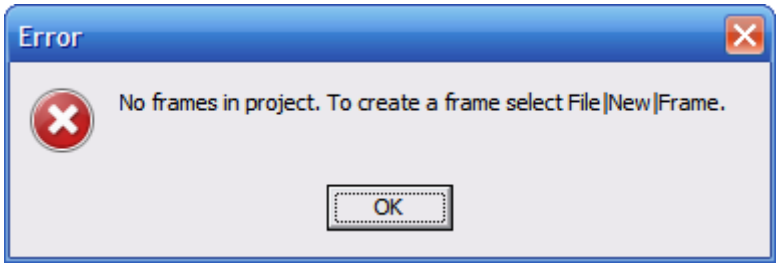


Рис. 6.13. Сообщение об ошибке размещения компонента Frame

Дело в том, что компонент Frame первый раз должен быть сконструирован через главное меню, поскольку является сложным компонентом. После его

создания, или создания нескольких таких компонентов, можно пользоваться стандартным способом размещения их на форме путем выбора имеющихся фреймов в раскрывающемся списке. Для практики создайте компонент Frame2 с помощью команды File | New | Frame, а затем перейдите на форму и щелкните дважды по компоненту Frames на вкладке Standard панели визуальных компонентов. Перед вами должно открыться окно (рис. 18.3), из которого можно выбрать компонент Frame2.

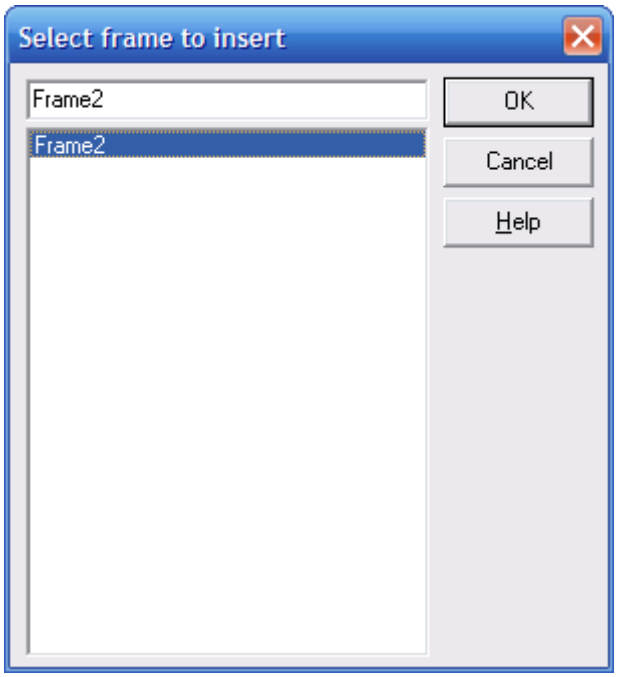


Рис. 6.14. Окно выбора компонента Frame

□ MainMenu (Главное меню) — предназначен для создания главного меню программы. С этим и другими компонентами мы познакомимся поближе в процессе создания новых программ.
□ PopupMenu (Всплывающее меню) — предназначен для создания всплывающего меню некоторых компонентов. Обычно с помощью этого ком-понента создается контекстное меню.
□ Label (Этикетка) — создает на форме текстовую метку или надпись.
□ Edit (Редактирование) — создает на форме поле для редактирования тек-стовой строки.
□ Memo (Поле) — отображает на форме поле для редактирования текстовых строк. Обычно служит для создания редакторов и полей для вывода блоков данных.
□ Button (Кнопка) — является самым распространенным компонентом. Служит для создания в приложении различных прямоугольных кнопок с текстовой надписью.
□ CheckBox (Ячейка состояния) — позволяет создавать на форме приложения ячейку с двумя состояниями (без галочки и с галочкой) и строкой названия. Щелчок левой кнопкой мыши по этому компоненту во время работы программы вызывает изменение состояния компонента на

противоположное. В программе всегда можно узнать состояние этого компонента и тем самым выполнять то или иное действие.
□ RadioButton (Радиокнопка) — создает круглое поле с двумя состояниями (с точкой и без точки) и текстовой строкой, поясняющей ее назначение в программе. Обычно несколько таких компонентов, расположенных на форме, позволяют переключить только один элемент из группы. Для наглядности сказанного и закрепления материала на практике создайте новое приложение и расположите на форме несколько компонентов RadioButton. После чего запустите приложение на выполнение и пощелкайте левой кнопкой мыши поочередно по каждому из них. Вы увидите, что можно изменить состояние только для одного из этих компонентов, так как остальные компоненты переключают при этом свое состояние автоматически.
□ ListBox (Окно списка) — создает прямоугольное поле для отображения текстовых строк с возможностью их выбора, добавления или удаления при работе программы.
□ ComboBox (Комбинированный список) — позволяет создавать на форме элемент, являющийся комбинацией строки ввода и выпадающего списка для выбора. Фактически объединяет в себе компоненты ListBox и Edit.
□ ScrollBar (Линейка прокрутки) — создает элемент, похожий на линейку с бегунком и кнопками для прокрутки окна, к которому относится этот элемент. Кроме того, с его помощью можно изменять в пределах некоторого заданного интервала значение величины какого-либо параметра.
□ GroupBox (Окно группы) — служит для создания области, визуально объединяющей на форме несколько интерфейсных элементов.
□ RadioGroup (Группа радиокнопок) — позволяет создавать на форме контейнер в виде прямоугольной рамки для объединения группы взаимоисключающих радиокнопок.
□ Panel (Панель) — создает пустую область, на которой можно разместить другие компоненты. Как правило, используется для создания панели инструментов в программе.
□ Action List (Список действий) — осуществляет управление взаимодействием между интерфейсными элементами и логикой программы.

На рис. 18.4 приведено окно формы, на которой расположены все перечисленные компоненты в порядке их описания слева направо и сверху вниз, начиная с MainMenu.

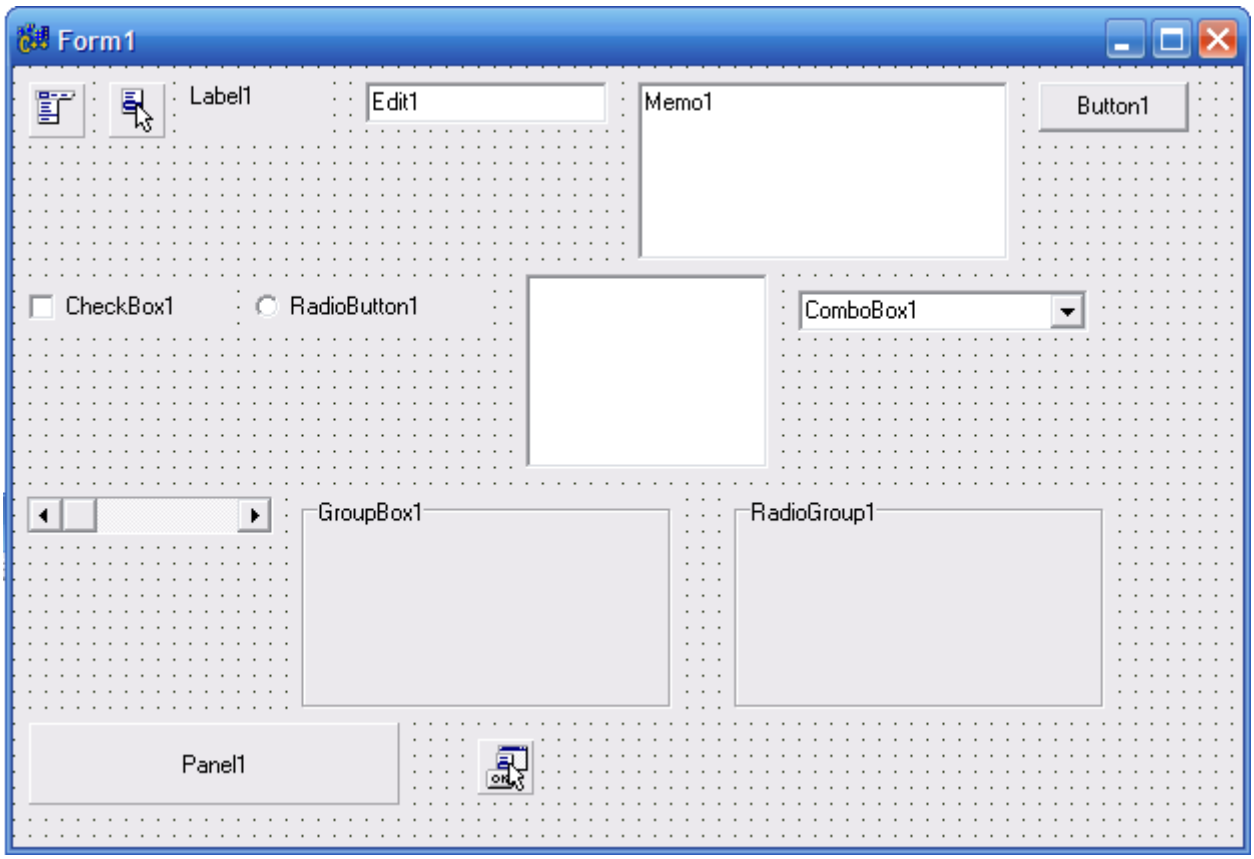


Рис. 6.15. Окно формы с компонентами

Компоненты Addition

Компоненты, расположенные на вкладке Addition (рис. 18.5), предназначены для включения в программу дополнительных интерфейсных элементов, с помощью которых можно создать более привлекательный и дружественный интерфейс программы. Рассмотрим компоненты, расположенные на этой вкладке.

□ BitBtn (Графическая кнопка) — служит для создания на форме приложения кнопки с изображением и надписью.
□ SpeedButton (Быстрая кнопка) — позволяет создать на форме кнопку с изображением без надписи. Знакома нам по панели быстрых кнопок.
□ MaskEdit (Форматированный ввод) — предназначен для создания прямоугольного поля ввода данных в специально заданном формате. Позволяет проверить корректность вводимых данных с помощью маски.

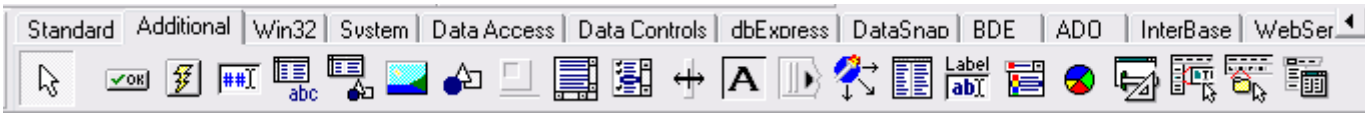


Рис. 6.16. Компоненты вкладки Addition

□ StringGrid (Строковая таблица) — служит для создания таблицы (сетки), состоящей из текстовых строк.
□ DrawGrid (Графическая таблица) — создает на форме двумерную таблицу для отображения графических данных.

□ Image (Образ) — предназначен для создания на форме невидимого контейнера, в который можно поместить один графический файл с битовым образом, пиктограммой или метафайл.
□ Shape (Фигура) — позволяет рисовать на форме простые геометрические фигуры, такие как окружность, квадрат, эллипс или прямоугольник при изменении свойства Shape. Допускает изменение цвета фигур и их штриховки при изменении свойств Color (Цвет) и Style (Стиль) группы Brush (Кисть).
□ Bevel (Скос) — создает объемные рамки для различных групп объектов. Служит только для обрамления и не позволяет выполнять программную функцию.
□ ScrollBox (Контейнер прокрутки) — позволяет создавать контейнер для объектов. Автоматически снабжается линейками прокрутки, если помещенный в него объект превышает размеры самого контейнера на экране.
□ Splitter (Разделитель) — разделяет рабочую область программы на две части и позволяет менять их размеры во время работы программы.
□ StaticText (Статический текст) — создает на форме текстовую строку, с некоторыми возможностями ее оформления.
□ Chart (Диаграмма) — предназначен для создания и вывода на печать многоцветных графиков и диаграмм.

Компоненты Win32

Компоненты, расположенные на вкладке Win32 (рис. 18.6), обеспечивают подключение к программе интерфейсных элементов, используемых в 32-разрядных версиях операционной системы Windows. Использование данных компонентов позволяет придать программе современный и профессиональный вид. Рассмотрим основные компоненты данной вкладки.
□ TabControl (Управление табуляцией) — служит для создания перекрывающих друг друга вкладок и для создания интерфейсов в стиле палитры компонентов.
□ PageControl (Управление страницами) — создает на форме контейнер для размещения дочерних страниц программы. Позволяет во время работы программы изменять ее интерфейс, перелистывая дочерние страницы на главной форме.

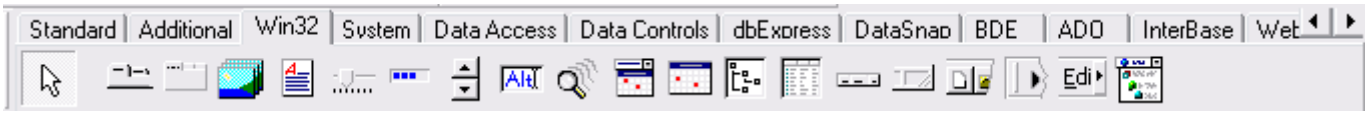


Рис. 6.17. Компоненты вкладки Win32

□ Image List (Список образов) — создает на форме невидимый контейнер для набора графических изображений одинакового размера. Позволяет выбирать изображение из списка.
□ RichEdit (Обогащенный редактор) — позволяет создавать редактор с готовым набором функций, свойственных большинству редакторов.

□ TrackBar (Дорожка с полосками) — создает на форме шкалу с метками и регулятором текущего положения. Применяется в программе как регулятор громкости звука, регулятор размера изображения и т. п.
□ Progress Bar (Прогресс-индикатор) — создает на форме прямоугольный индикатор для отображения процесса выполнения длинных процедур в программе (копирования, поиска и т. п.).
□ UpDown (Вверх-вниз) — служит для создания интерфейсных элементов с возможностью увеличения или уменьшения какого-либо значения с помощью кнопок Вверх и Вниз данного компонента.
□ HotKey (Горячая клавиша) — обеспечивает возможность создания пользователем горячих клавиш, определяющих быстрый доступ к разделам меню.
□ Animate (Аниматор) — создает на форме невидимый контейнер для воспроизведения видеозаписей в формате AVI.
□ DataTimePicker (Сборщик даты и времени) — создает в программе интерфейс для ввода даты и времени.
□ MonthCalendar (Месячный календарь) — размещает на форме календарь с отображением всех дней месяца и возможностью перелистывания месяцев и корректировки даты текущего дня.
□ TreeView (Вид дерева) — позволяет создавать в программе иерархическое древовидное отображение данных.
□ ListView (Вид списка) — создает список элементов с отображением в различных стилях (крупные значки, мелкие значки, таблица и пр.).
□ HeaderControl (Управление заголовком) — служит для управления панелями, расположенными под данным компонентом.
□ StatusBar (Панель состояния) — создает контейнер в нижней части формы для отображения статусной информации. Например, состояние кнопок Caps Lock, Num Lock и Scroll Lock.
□ ToolBar (Панель инструментов) — позволяет создать на форме контейнер для размещения быстрых кнопок.
□ CoolBar (Холодная панель) — позволяет делать перестраиваемые панели, состоящие из полос.
□ PageScroller (Страница прокрутки) — создает на форме контейнер для прокрутки элементов, не вмещающихся на экран целиком.
□ ComboBoxEx (Расширенный выпадающий список) — позволяет создать на форме выпадающий список элементов.

Контрольные вопросы:
1. Визуальное программирование интерфейса, объясните.
2. Функции управляющих кнопок Button и BitBtn
3. Назначение и функции Группы радиокнопок
4. Индикатор CheckBox, для чего используется?

5. Свойства компонента StringGrid, объясните.

Тест

1. Компонент _____ предназначен для создания на форме невидимого контейнера, в который можно поместить один графический файл с битовым образом, пиктограммой или метафайл.
2. Компонент _____ служит для создания таблицы (сетки), состоящей из текстовых строк.

3. Какой компонент создает на форме невидимый контейнер для воспроизведения видеозаписей в формате AVI ?
Animate (Аниматор);
RichEdit (Обогащенный редактор);
Image (Образ);
Shape (Фигура);
4. Компоненты предназначены для включения в программу дополнительных интерфейсных элементов, с помощью которых можно создать более привлекательный и дружественный интерфейс программы, это:
Компоненты Standard;
Компоненты Addition;
Компоненты Dialogs;
Компоненты Samples;
5. Компонент _____ создает круглое поле с двумя состояниями (с точкой и без точки) и текстовой строкой, поясняющей ее назначение в программе.

ЛЕКЦИЯ №11

Тема: Разработка программ базы данных в средах (Borland C++ Builder 6, Visual C++) на C++: возможности создания базы данных, системы управления базами данных**. (2ч.).**

Цель занятия. Введение в Borland C++ и Visual C++, научиться создавать программы базы данных в средах в Borland C++ и Visual C++.

План лекции:

1. Введение.
2. Организация доступа к базам данных
3. Механизм BDE
4. Структура взаимодействия приложений с базами данных.
5. Использование визуальных компонент
6. Взаимосвязь компонент управления и доступа к содержимому баз данных.
7. Запрос

Все приложения СУБД, создаваемые в среде C++Builder, являются клиентами в архитектуре программного взаимодействия клиент/сервер. Клиент выдает запросы к серверу базы данных на получение или передачу информации. Сервер обрабатывает запросы от множества клиентов одновременно, координируя доступ к данным и их обновление.

Все приложения СУБД, создаваемые в среде C++Builder, основаны на компонентах пользовательского интерфейса с некоторой базой данных, которые предоставляют удивительно легкие в использовании средства разработки специальных приложений. Львиная доля времени процесса разработки уходит на визуальную установку свойств выбранных компонент. Удачно спроектированное приложение всегда обеспечивает простоту просмотра и редактирования данных пользователем, независимо от сложности структуры используемой модели данных. Данная глава с очевидностью покажет, что формы приложений СУБД для типично сложной системы в архитектуре взаимодействия клиент/сервер действительно могут быть созданы в интегрированной среде C++Builder весьма быстро и с малыми усилиями.

Воздействия на компоненты многогранны: их реакция на события обеспечивается стандартными методами, а установка значений свойств может производиться во время работы приложения. Таким образом, простое вначале приложение в процессе разработки постепенно усложняется, чтобы в конце концов выглядеть совершенно профессиональным программным изделием.

Организация доступа к базам данных

C++Builder организует доступ приложения к данным таким образом, чтобы полностью отстранить разработчика от специфики обслуживания конкретной базы данных.

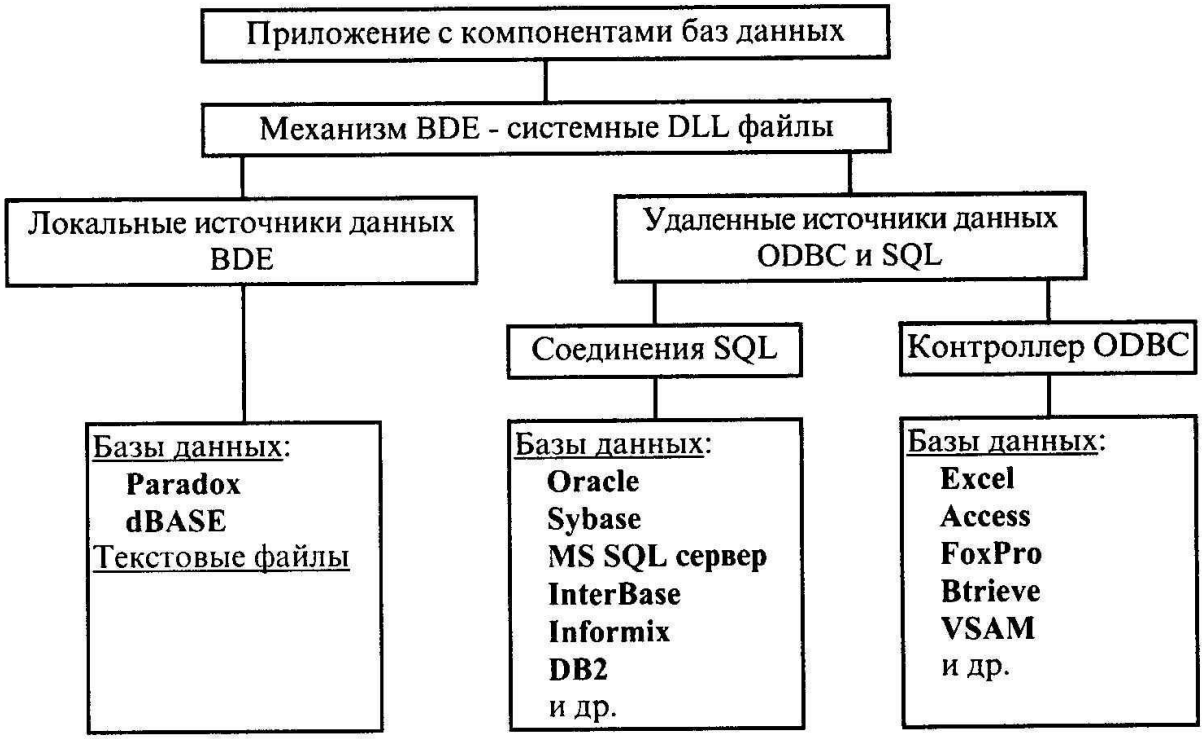
Механизм BDE

Ключевой механизм BDE (Borland Database Engine), обеспечивающий работу визуальных компонент баз данных, действует как интерфейс между вашим приложением и самой базой данных. BDE реализован в виде набора системных DLL файлов. Взаимодействие компонентных объектов с BDE никак не специфицирует конкретную базу данных и не зависит от реализации обмена информацией на нижнем уровне иерархии. Именно BDE

обращается в свою очередь к драйверам, специфическим для базы данных указанного типа, возвращая вашему приложению запрошенные фактические данные. BDE играет роль, аналогичную контроллеру драйверов ODBC (Open Database Connectivity) производства фирмы Microsoft, изолируя приложения от нижнего уровня взаимодействия с базой данных и увеличивая общую производительность связи за счет использования кэш-памяти. Используя BDE, вы получаете доступ ко всем локальным стандартным базам данных вашего компьютера, к источникам данных ODBC и к SQL серверам баз данных в архитектуре сетевой связи клиент/сервер.

Унифицированная технология BDE применяется во всех продуктах производства корпорации Borland: C++Builder, Borland C++, Delphi, IntraBuilder и JBuilder. Чтобы получить доступ к содержимому базы данных, приложению необходимо знать только идентификатор ее псевдонима (alias).

Приложение с компонентами баз данных



Структура взаимодействия приложений с базами данных.

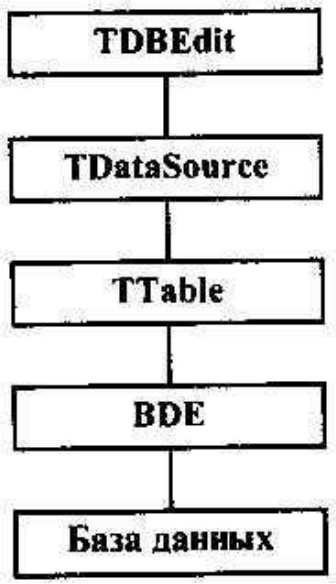
При добавлении компонент баз данных к форме вашего приложения соединение с BDE происходит автоматически - никакого программирования не требуется. Визуальный процесс соединения полностью находится под вашим контролем. Во время выполнения программы BDE делает необходимые запросы и получает данные, заказанные свойствами каждой используемой компоненты.

Использование визуальных компонент

Одним из важнейших достоинств интегрированной среды C++Builder является наличие удобных средств быстрой визуальной разработки приложений СУБД - специализированных компонент баз данных. В отличие от разделяемых элементов управления VBX, C++Builder компилирует компоненты в единую исполняемую программу, что существенно повышает ее надежность и быстродействие. Только очень опытные программисты способны создать программу подобного уровня качества и гибкости, используя исключительно прямые обращения к соответствующим функциям Windows API. При таком подходе даже простое приложение требует написания непомерного по объему кода.

C++Builder предоставляет разработчикам интерфейсные элементы баз данных из Библиотеки Визуальных Компонент на следующих двух вкладках Палитры компонент:

• Компоненты управления данными Data Control (такие как область редактирования TDBEdit или сетка TDBGrid) обеспечивают отображение и редактирования записей на форме приложения.
• Компоненты доступа к данным Data Access (такие как таблица TTable или запрос TQuery) адресуют фактические данные, хранящиеся в файле базы данных, а компонента источника TDataSource служит как интерфейс межкомпонентной связи.
Для. работы с базами данных необходимо проанализировать и правильно установить значения ключевых свойств компонент доступа и управления. В дальнейшем будем выделять ключевые свойства, методы и события подчеркиванием.
C++Builder поддерживает "трехступенчатую" модель разработки приложения баз данных. В этой модели компонента управления связана с компонентой источника, а та, в свою очередь, получает фактические данные таблицы или запроса посредством механизма BDE. Рис. 5.5 показывает пример взаимосвязи компонент.



Отображает область редактируемого ввода текста.

Служит как интерфейс между компонентами управления и доступа.

Содержит все записи данной таблицы базы данных.

Изолирует приложение от специфики обслуживания базы данных конкретного типа.

Взаимосвязь компонент управления и доступа к содержимому баз данных.
Среднее звено, компонента TDataSource, допускает менять фактическую таблицу на стадии проектирования формы без какого бы то ни было перепрограммирования самого приложения - все отображаемые элементы связаны с источником, а не с питающими его таблицей или запросом. Кроме того, компоненты источников берут на себя задачу синхронизации обмена данными между парами таблиц по принципу master-detail.

Запросы

Компоненты таблиц являются полноправными, гибкими и легкими в использовании компонентами доступа, достаточными для многих приложении СУБД. TTable возвращает все строки и столбцы единственной таблицы, если доступ не ограничивается установкой интервалов и фильтров. Компоненты запросов предоставляют разработчикам альтернативные возможности. TQuery обеспечивает доступ к нескольким таблицам одновременно и способна адресовать некоторое подмножество записей. Вид*возвращаемого набора данных (result set)* зависит от формы запроса, который может быть либо статическим, когда все параметры запроса задаются на стадии проектирования, или динамическим, когда параметры определяются во время выполнения программы.

Указанные действия записываются и реализуются на стандартизованном языке структурированных запросов SQL (Structured Query Language), принятом большинством удаленных серверов реляционных баз данных, таких как Sybase, Oracle, InterBase и SQL Server. На SQL можно сформулировать весьма изощренные запросы к базам данных.

C++Builder передает запросы серверу, который интерпретирует их и возвращает результаты вашему приложению.

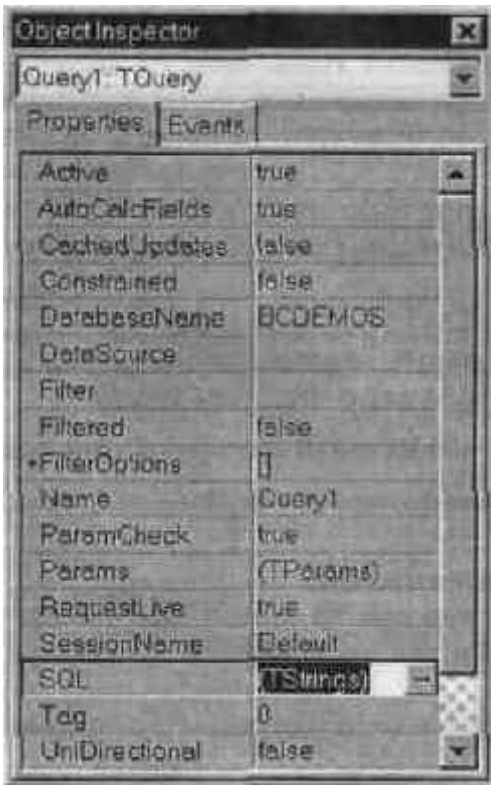


Рис. 5.10. Свойства запроса.

Active разрешает или запрещает режим просмотра "живых данных", возвращаемых запросом на этапе проектирования. Значение false устанавливается по умолчанию.

DatabaseName содержит псевдоним базы данных или полный путь к ее каталогу, необходимые для разрешения запроса.

RequestLive разрешает или запрещает BDE сделать попытку возвратить "живой" результирующий набор. Значение false (устанавливается по умолчанию) указывает, что результаты запроса нельзя модифицировать. Значение true гарантирует возврат результирующего набора при условии, что синтаксис команды SELECT согласуется с требованиями запрашиваемых данных.

SQL используется для ввода команды SQL посредством строчного редактора списка, который открывается двойным щелчком мышью в графе значений этого свойства. Локальные и удаленные серверы баз данных обеспечивают выполнение четырех команд SQL: SELECT - для выбора существующих данных из таблиц; INSERT - для добавления

ЛЕКЦИЯ №12

Тема: Графические возможности Borland C++, Visual C++, инициализация, разметка экрана. Графический режим, построение графиков функций **(2ч.)**.

Цель занятия. Введение в Borland C++ и Visual C++, работа в графическом режиме, построение графиков функции.

План лекции:

1. Инициализация , разметка экрана.

2. Графический режим, построение графиков функций.

В среде C++Builder существует три рода объектов, которые имеют отношение к графике:

- Канва - предоставляет битовую карту поверхности окна приложения, компоненты, принтера и т.п., которая может быть использована для вывода графики. Канва не самостоятельный объект, она всегда является свойством какого-то другого графического объекта.
- Графика - представляет растровое изображение некоторого файла или ресурса (битового образа, пиктограммы или метафайла).

  C++Builder определяет производные от базового класса TGraphic объектные классы:

  - TBitmap,
  - Ticon,
  - TMetafile.
- Рисунок (TPicture) представляет собой контейнер для графики, который может содержать любые классы графических объектов. Таким образом, контейнерный класс TPicture может содержать битовый образ, пиктограмму, метафайл или некоторый другой графический тип, определенный пользователем, а приложение может стандартно обращаться ко всем объектам контейнера посредством объекта TPicture.

Отметим, что графические объекты Windows взаимосвязаны. Так - объект TPicture всегда содержит некоторую графику, которой в свою очередь, может потребоваться для отображения канва, а единственный стандартный графический класс канвы - это TBitmap.

Как отмечалось выше, Borland C++ Builder инкапсулирует функции Windows GDI на разных уровнях. Наиболее завершенным является интерфейс, предоставляемый свойством Canvas (канва), объектного класса канвы, его графических компонент. Использование канвы снимает с программиста заботу при выводе изображений об инициализации контекста устройства и его освобождении. Наличия вложенных свойств (характеристик пера, кисти, шрифтов, растровых изображений) также не требует слежения за состояниями ресурсов - основная задача - это определение характеристик для этих графических объектов и грамотное их использование. Речь об этом в следующем параграфе.

В начало

Объектный класс канвы

Инкапсулированные и перегруженные функции GDI и WinApi объектного класса канвы многие авторы относят к трем различным уровням. В этой условной классификации функции высокого уровня обеспечивают возможность рисования линий, фигур и текста. Определение свойств и методов манипулирования графическими примитивами канвы отнесены к среднему уровню. Нижний уровень обеспечивается доступ к самим функциям Windows GDI. Классификация не бесспорна, но она позволяет ориентироваться в достаточно большом колличестве свойств и методов канвы и, поэтому, приведем эту классификацию.

Уровень	Метод (Функция)	Свойства	Действие
Высокий	**MoveTo**	**PenPos**	**Определяет текущую позицию пера**
	LineTo	**PenPos**	**Рисует прямую до заданной точки**
	Rectangle		**Рисует прямоугольник**
	Ellipse		**Рисует эллипс**
	Arc		**Рисует дугу**
	Polyline		**Рисует ломаную линию**
	PolyBezier		**Рисует кривую Блейзера**
	Chord		**Рисует сектор**
	DrawFocusRect		**Рисует прямоугольник**
	FrameRect		**Выводит рамку вокруг прямоугольника**
	Pie		**Выводит сектор круга**
	TextOut		**Выводит текстовую строку**
	TextHeight		**Задает высоту текстовой строки**
	TextWidth		**Задает ширину для вывода текстовой строки**
	TextRect		**Вывод текста внутри прямоугольника**
	FillRect		**Заливка указанного прямоугольника цветом и текстурой текущей кисти**
	FloodFill		**Заливка области канвы (произвольной формы) заданным цветом**
Средний		**Pen**	**Используется для установки цвета, стиля, ширины и режима пера**
		Brush	**Используется для установки цвета и текстуры при заливке графических фигур и фона канвы.**
		Font	**Используется для установки шрифта заданного цвета, размера и стиля**
		Pixels	**Используется для чтение и записи**

			цвета заданного пикселя канвы
	CopyRect	**CopyMode**	**Копирует прямоугольную область канвы в режиме CopyMode**
	BrushCopy		**Копирует прямоугольную область канвы с заменой цвета**
	Draw		**Рисует битовый образ, пиктограмму, метафайл в заданном месте канвы**
	StretchDraw		**Рисует битовый образ, пиктограмму или метафайл так, чтобы целиком заполнить заданный прямоугольник**
Низкий		**Handle**	**Используется как параметр при вызове функций Windows GDI**

В начало

Примеры использования функций канвы для рисования приметивов

Основы рисования примитивов рассмотрены выше при рассмотрении функций GDI, здесь, на примерах, не повторяя уже описанные ранее приемы вывода примитивов, показаны лишь некоторые особенности их отображения с использованием функций канвы.

Прстейший пример относится к рисованию линий, кроме того показано как можно задавать параметры пера:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 //Задаем цвет пера
 Canvas->Pen->Color=(TColor)RGB(255,0,0);
 //Задаем ширину  пера
 Canvas->Pen->Width=5;
 //Можно переместить перо в исходную точку так
 Canvas->MoveTo(100,200);
 //Или тпереместить перо так
 TPoint tPoint;
 tPoint.x=100;
 tPoint.y=200;
 Canvas->PenPos=tPoint;
 //И рисуем линию от исходной точки 100,200 до конкечной 0,50
 Canvas->LineTo(0,50);
 //Освобождать и восстанавливать ничего не надо
 //однако канва запоминает установленные параметры
}
```

Пример рисования дуги и секторов:

```
void __fastcall
```

```
TForm1::Button1Click(TObject *Sender)
{
 //Стиль кисти
 Canvas->Brush->Style=bsHorizontal;
 //Цвет кисти
 Canvas->Pen->Color = clBlue;
 //Рисуем дугу
 Canvas->Arc(0,0,500,500,250,0,50,0);
 //Рисуем сектор изменяя стиль взаимодействия цвета пера и холста
 Canvas->Pen->Mode=pmWhite;
 Canvas->Chord(0,0,250,250,250,125,0,0);
 //Освобождать и восстанавливать ничего не надо
 //Но помним, что канва запомнила установленные параметры
}
```

Пример рисования ломаных линий (перо, при желании, можно задать как и ранее):

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 TPoint tPoints[6];
 Canvas->Pen->Color = clRed;
  Canvas->Pen->Width=3;
 tPoints[0].x = 40;
 tPoints[0].y = 10;
 tPoints[1].x = 20;
 tPoints[1].y = 60;
 tPoints[2].x = 70;
 tPoints[2].y = 30;
 tPoints[3].x = 10;
 tPoints[3].y = 30;
 tPoints[4].x = 60;
 tPoints[4].y = 60;
 tPoints[5].x = 40;
 tPoints[5].y = 10;
 Canvas->Polyline(tPoints,5);
}
```

Пример рисования кривых Блейзера:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 TPoint tPoints[7];
 tPoints[0]=TPoint(0,0);
 tPoints[1]=TPoint(800,30);
 tPoints[2]=TPoint(0,40);
 tPoints[3]=TPoint(550,400);
 tPoints[4]=TPoint(350,200);
 tPoints[5]=TPoint(550,400);
 tPoints[6]=TPoint(0,500);
 Canvas->PolyBezier(tPoints,6);
```

```
}
```

Пример отображения прямоугольников и эллипсов различными способами и использования кистей:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 //Задаем цвет пера
 Canvas->Pen->Color=(TColor)RGB(0,255,0);
 //Задаем ширину  пера
 Canvas->Pen->Width=1;
 //Стиль пера - пунктир
 Canvas->Pen->Style=psDot;
 //Стиль вывода замкнутой фигуры, зависяший от цвета пера
 //и канвы ( заменяет своими возможностями) функцию SetBkMode()
 //Эдесь прозрачный фон
 Canvas->Pen->Mode=pmCopy;
 //Рисуем прямоугольник по точкам
 Canvas->Rectangle(0,0,100,100);
 //Здесь, аналог GDI непрозрачного фона
 Canvas->Pen->Mode=pmWhite;
 //Рисуем элипс вписанный в прямоугольник
 Canvas->Ellipse(250,250,350,550);
 //Создаем перо функцией CreatePen() 1 - толщина пера
 HPEN hPen=CreatePen(PS_DASHDOTDOT,1,RGB(255,0,0));
 //Устанавливаем это перо как текущее
 Canvas->Pen->Handle=hPen;
 //Изменяем стиль вывода
 Canvas->Pen->Mode=pmCopy;
 //Стиль кисти - вертикальная штриховка
 Canvas->Brush->Style=bsVertical;
 //Можно переопределить прозрачность и так
 SetBkMode(Canvas->Handle,OPAQUE);
 //Рисуем прямоугольник с закругленными краями
 Canvas->RoundRect(100,100,200,200,50,50);
 //Координаты можно задать и так
 TRect tRect;        //Координаты точек
 tRect.Left=100;     //Левыя
 tRect.Right=500;    //Правая
 tRect.Top=250;      //Верхняя
 tRect.Bottom=450;   //Нижняя
 //Используем кисть для закрашивания объекта
 Canvas->Brush->Color=(TColor)RGB(0,0,255);
 Canvas->Rectangle(tRect);
 Canvas->Brush->Color=(TColor)RGB(0,255,255);
 //Стиль кисти
 Canvas->Brush->Style=bsCross;
 //Или по  координатам
 Canvas->RoundRect(100,300,250,450,50,50);
}
```

Использование кисти для заливки фигур:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 Canvas->Brush->Color=(TColor)RGB(0,255,255);
 TRect tRect(0,0,100,100);
 Canvas->FillRect(tRect);
 //Рисуем прямоугольник по точкам
 Canvas->Rectangle(0,0,100,100);
 //Освобождать и восстанавливать ничего не надо
}
```

И интересный эффект заполнения канвы цветом кисти:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 //Параметр fsBorder -  заполнить всю область
 //цветом кисти, до края канвы
 Canvas->Brush->Color=(TColor)RGB(255,0,255);
 //Исходная точка в центре канвы, в данном случае
 //окна приложения
 Canvas->FloodFill(Width/2,Height/2, NULL, fsBorder);
 //Меняем цвет и повторяем
 Canvas->Brush->Color=(TColor)RGB(255,0,0);
 Canvas->FloodFill(Width/2,Height/2,NULL, fsBorder);
}
```

Рисование рамки вокруг прямоугольника:

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 Canvas->Brush->Color=(TColor)RGB(125,0,255);
 TRect tRect;          //Координаты точек
 tRect.Left=100;       //Левыя
 tRect.Right=500;      //Правая
 tRect.Top=250;        //Верхняя
 tRect.Bottom=450;     //Нижняя
 Canvas->FrameRect(tRect);
}
```

Из примеров видно, что (с учетом некоторой модификации параметров функций) рисование графических примитивов аналогично интерфейсу GDI. Однако, так как не требуется следить за состоянием контекста и его графических объектов, написание кода значительно упрощается.

В начало

Вывод текста на канву

Вывод текста на канву предельно прост. Требуется только задать характеристики шрифта (свойство Font канвы), текст в формате AnsiString и использовать метод TextOutA() канвы. Следующий пример показывает как это делается и, также, демонстрирует то, что канва сохраняет заданные свойства - нажатие кнопки Button2, в обработчике нажатия которой изменен лишь цвет, не изменяет остальных свойств шрифта, заданных в обработчике нажатия кнопки Button1.

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 AnsiString vasS="Пример текста";
 //Цвет текста
 Canvas->Font->Color=clRed;
 //Размер шрифта в точках
 Canvas->Font->Size=20;
 //Стиль шрифта
 TFontStyles tFontStyle;
 //Зачеркнутый, наклонный, жирный, подчепкнутый
 tFontStyle << fsStrikeOut << fsItalic << fsBold << fsUnderline;
 Canvas->Font->Style =tFontStyle;
 //Имя шрифта
 Canvas->Font->Name="Times";
 //Вывод текста
 Canvas->TextOutA(10,10,vasS);
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{
 AnsiString vasS="Пример текста";
 Canvas->Font->Color=clBlue;
 Canvas->TextOutA(50,50,vasS);
}
```

Рис 1. Пример работы со шрифтами.

Полезные функции для работы с текстом:

- TextWidth(AnsiString vasS); - возвращает ширину текста в пикселях, необходимую для отображения строки vasS заданным шрифтом канвы;
- TextHeight(AnsiString vasS); - возвращает высоту текста в пикселях, необходимую для отображения строки vasS заданным шрифтом канвы.

Эти функции полезны когда надо быть уверенным, что при езменении размера шрифта текст не выдет за грани компонента.

В следующем примере показано изменение ширины компонента TListBox, при выводе на его канву текста у которого заранее изменен шпифт. В обработчик нажатия кнопки Button1 поместить функцию TextWidth() и за ней функцию TextOutA() нельзя, так как перирисовка компонента происходит после завершения обработчика и часть текста будет утеряна. В примере, кроме того, показано задание цвета в виде 16ти ричного числа и высоты шрифта в пикселях.

```
void __fastcall
TForm1::Button1Click(TObject *Sender)
{
 //Цвет текста
 ListBox1->Canvas->Font->Color=(TColor)0x00FF7D7D;
 //Высота текста в пикселях
 ListBox1->Canvas->Font->Height=25;
 AnsiString vasS="Пример текста";
 ListBox1->Width=ListBox1->Canvas->TextWidth(vasS)+20;
}
void __fastcall
TForm1::Button3Click(TObject *Sender)
{
 AnsiString vasS="Пример текста";
 ListBox1->Canvas->TextOutA(2,10,vasS);
}
```

Каждый шрифт поддерживает один или больше наборов символов, которые и определяют их написание. Следующий пример хотя и выводит абракадабру первые две строки, но показывает как, используя свойство Charset, сменить набор символов шрифта установленный по умолчанию. Кроме того можно отметить разницу набора символов RUSSIAN_CHARSET и DEFAULT_CHARSET - как правило они разные. Пятая функция показывает использование еще одного свойства - Pitch. Символы в шрифтах с переменным шагом отличаются по ширине (значения fpDefault, fpVariable), но если установить значение свойства равным fpFixed, ширина символов будет одинаковой.

```
AnsiString vasS="Пример текста";
ListBox1->Canvas->Font->Name="ArialBlack";
ListBox1->Canvas->Font->Charset = TURKISH_CHARSET;
ListBox1->Canvas->TextOutA(2,10,vasS);
ListBox1->Canvas->Font->Charset = SYMBOL_CHARSET;
ListBox1->Canvas->TextOutA(2,30,vasS);
ListBox1->Canvas->Font->Charset = RUSSIAN_CHARSET;
ListBox1->Canvas->TextOutA(2,50,vasS);
ListBox1->Canvas->Font->Charset =  DEFAULT_CHARSET;
ListBox1->Canvas->TextOutA(2,70,vasS);
ListBox1->Canvas->Font->Pitch=fpFixed;
ListBox1->Canvas->TextOutA(2,90,vasS);
```

Следущее свойство шрифта PixelsPerInch относится к отображению текста и графики при печати, о чем речь будет вестись отдельно, но кратко рассмотрено

здесь, чтобы позже при возврате к свойствам шрифтов вновь не повторяться. Принтер, о чем уже говорилось, также имеет канву. Вывод на печать в Borland C++ Builder - это вывод на канву принтера. При смене принтера, размер шрифтов может отмасштабироваться не правильно. Чтобы сделать правильное масштабирование, необходимо установить свойство PixelsPerInch шрифта. Свойство PixelsPerInch определяет количество пикселей в 1 дюйме. Для его установки можно воспользоваться следующим кодом:

```
TPrinter *ptPrint = Printer();
ptPrint->Canvas->Font->PixelsPerInch=
  GetDeviceCaps(ptPrint->Canvas->Handle,LOGPIXELSY);
```

Отметим также, что свойство PixelsPerInch имеют не только шрифты, но и, например Экран (Screen) и может быть исрользовано при установке фиксированных размеров компонент (1 сантиметр равен 1 * (Screen->PixelsPerInch/2.54, 1 дюйм = 25.4 мм).

И песледнее - шрифт также имеет свойство Handle, которое может быть использовано для быстрого восстановления характеристик шрифта канвы по умолчанию (за исключением цвета).

***Введение в объектно-ориентированное программирование*.**
Инкапсуляция, полиморфизм, наследование.

Цель занятия. *Ознакомить с* инкапсуляцией, полиморфизмом, наследованием.

План лекции:

1. Инкапсуляция.
2. Полиморфизм.
3. Наследование.

Абстра́кция — в объектно-ориентированном программировании это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей её реализации в терминах более примитивных функций, таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня. Такой подход является основой объектно-ориентированного программирования. Это позволяет работать с объектами, не вдаваясь в особенности их реализации. В каждом конкретном случае применяется тот или иной подход: инкапсуляция, полиморфизм или наследование. Например, при необходимости обратиться к скрытым данным объекта, следует воспользоваться инкапсуляцией, создав, так называемую, функцию доступа или свойство. Абстракция данных — популярная и в общем неверно определяемая техника программирования. Фундаментальная идея состоит в разделении несущественных деталей реализации подпрограммы и характеристик существенных для корректного ее использования. Такое разделение может быть выражено через специальный «интерфейс», сосредотачивающий описание всех возможных применений программы[1]. С точки зрения теории множеств, процесс представляет собой организацию для группы подмножеств своего множества. См. также Закон обратного отношения между содержанием и объемом понятия.

Инкапсуля́ция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс. Реализуется с помощью ключевого слова: public. Пользователь не может использовать закрытые данные и методы. Реализуется с помощью ключевых слов: private, protected, internal. Инкапсуляция — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, полиморфизмом и наследованием). Сокрытие реализации целесообразно применять в следующих случаях: предельная локализация изменений при необходимости таких изменений, прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности) и прогнозируемость последствий изменений.

Насле́дование — один из четырёх[источник?] важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса. Простое наследование Класс, от которого произошло наследование, называется базовым или

родительским (англ. baseclass). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derivedclass). В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Множественное наследование

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python и Эйфель. Множественное наследование поддерживается в языке UML. Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова конкретного метода конкретного родителя. Попытка решения проблемы наличия одинаковых имен методов в предках была предпринята в языке Эйфель, в котором при описании нового класса необходимо явно указывать импортируемые члены каждого из наследуемых классов и их именование в дочернем классе. Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования[1]. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций». Полиморфизм — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием). Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение: внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов и их количеством); внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

Формы полиморфизма Edit

Полиморфизм включения

Этот полиморфизм называют чистым полиморфизмом. Применяя такую форму полиморфизма, родственные объекты можно использовать обобщенно. С помощью замещения и полиморфизма включения можно написать один метод для работы со всеми типами объектов TPerson. Используя полиморфизм включения и замещения можно работать с любым объектом, который проходит тест «is-A». Полиморфизм включения упрощает работу по добавлению к программе новых подтипов, так как не нужно добавлять конкретный метод для каждого нового типа, можно использовать уже существующий, только изменив в нем поведение системы. С помощью полиморфизма можно повторно использовать базовый класс; использовать любого потомка или методы, которые использует базовый класс.

Параметрический полиморфизм

Используя Параметрический полиморфизм можно создавать универсальные базовые типы. В случае параметрического полиморфизма, функция реализуется для всех типов одинаково и таким образом функция реализована для произвольного типа. В параметрическом полиморфизме рассматриваются параметрические методы и типы.

Параметрические методы. Если полиморфизм включения влияет на наше восприятие объекта, то параметрические полиморфизм влияет на используемые методы, так как можно создавать методы родственных классов, откладывая объявление типов до времени выполнения. Для избежания написания отдельного метода каждого типа применяется параметрический полиморфизм, при этом тип параметров будет являться таким же параметром, как и операнды.

Параметрические типы. Вместо того, чтобы писать класс для каждого конкретного типа следует создать типы, которые будут реализованы во время выполнения программы то есть мы создаем параметрический тип. Полиморфизм переопределения. Абстрактные методы часто относятся к отложенным методам. Класс, в котором определен этот метод может вызвать метод и полиморфизм обеспечивает вызов подходящей версии отложенного метода в дочерних классах. Специальный полиморфизм допускает специальную реализацию для данных каждого типа. Полиморфизм-перегрузка - это частный случай полиморфизма. С помощью перегрузки одно и то же имя может обозначать различные методы, причем методы могут различаться количеством и типом параметров, то есть не зависят от своих аргументов. Метод может не ограничиваться специфическими типами параметров многих различных типов.

Введение в объектно-ориентированное программирование.

Понятия класса. Классы и объекты.

Цель занятия. *Ознакомится с классами и объектами.*

Ключевые слова: класс , объект.

План лекции:

1. Классы.

2. Объекты.

Определение класса состоит из двух частей: заголовка, включающего ключевое слово class, за которым следует имя класса, и тела, заключенного в фигурные скобки. После такого определения должны стоять точка с запятой или список объявлений:
classScreen { /* ... */ };
classScreen { /* ... */ }myScreen, yourScreen;
Внутри тела объявляются данные-члены и функции-члены и указываются уровни доступа к ним. Таким образом, тело класса определяет список его членов.
Каждое определение вводит новый тип данных. Даже если два класса имеют одинаковые списки членов, они все равно считаются разными типами:

```
classFirst {
intmemi;
doublememd;
};

classSecond {
intmemi;
doublememd;
};

classFirst obj1;
Second obj2 = obj1;   // ошибка: obj1 и obj2 имеют разные типы
```

Тело класса определяет отдельную область видимости. Объявление членов внутри тела помещает их имена в область видимости класса. Наличие в двух разных классах членов с одинаковыми именами – не ошибка, эти имена относятся к разным объектам. (Подробнее об областях видимости классов мы поговорим в разделе 13.9.)
После того как тип класса определен, на него можно ссылаться двумя способами:
написать ключевое слово class, а после него – имя класса. В предыдущем примере объект obj1 класса First объявлен именно таким образом;
указать только имя класса. Так объявлен объект obj2 класса Second из приведенного примера.
Оба способа сослаться на тип класса эквивалентны. Первый заимствован из языка C и остается корректным методом задания типа класса; второй способ введен в C++ для упрощения объявлений.

Объекты классов

Определение класса, например Screen, не приводит к выделению памяти. Память выделяется только тогда, когда определяется объект типа класса. Так, если имеется следующая реализация Screen:

```
class Screen {
public:
  // функции-члены
```

```
private:
string         _screen;
string:size_type _cursor;
short          _height;
short          _width;
};
```

тоопределение

```
ScreenmyScreen;
```

выделяет область памяти, достаточную для хранения четырех членов Screen. Имя myScreen относится к этой области. У каждого объекта класса есть собственная копия данных-членов. Изменение членов myScreen не отражается на значениях членов любого другого объекта типа Screen.

Область видимости объекта класса зависит от его положения в тексте программы. Он определяется в иной области, нежели сам тип класса:

```
classScreen {
// списокчленов
};

int main()
{
  Screen mainScreen;
}
```

Тип Screen объявлен в глобальной области видимости, тогда как объект mainScreen – в локальной области функции main().

Объект класса также имеет время жизни. В зависимости от того, где (в области видимости пространства имен или в локальной области) и как (статическим или нестатическим) он объявлен, он может существовать в течение всего времени выполнения программы или только во время вызова некоторой функции. Область видимости объекта класса и его время жизни ведут себя очень похоже. (Понятия области видимости и времени жизни введены в главе 8.)

Объекты одного и того же класса можно инициализировать и присваивать друг другу. По умолчанию копирование объекта класса эквивалентно копированию всех его членов. Например:

```
Screen bufScreen = myScreen;
// bufScreen._height = myScreen._height;
// bufScreen._width = myScreen._width;
// bufScreen._cursor = myScreen._cursor;
// bufScreen._screen = myScreen._screen;
```

Указатели и ссылки на объекты класса также можно объявлять. Указатель на тип класса разрешается инициализировать адресом объекта того же класса или присвоить ему такой адрес. Аналогично ссылка инициализируется l-значением объекта того же класса. (В объектно-ориентированном программировании указатель или ссылка на объект базового класса могут относиться и к объекту производного от него класса.)

```
int main()
{
  Screen myScreen, bufScreen[10];
  Screen *ptr = new Screen;
```

```
myScreen = *ptr;
deleteptr;
ptr = bufScreen;
  Screen &ref = *ptr;
Screen&ref2 = bufScreen[6];
}
```

По умолчанию объект класса передается по значению, если он выступает в роли аргумента функции или ее возвращаемого значения. Можно объявить формальный параметр функции или возвращаемое ею значение как указатель или ссылку на тип класса. (В разделе 7.3 были представлены параметры, являющиеся указателями или ссылками на типы классов, и объяснялось, когда их следует использовать.В разделе 7.4 с этой точки зрения рассматривались типы возвращаемых значений.)

Для доступа к данным или функциям-членам объекта класса следует пользоваться соответствующими операторами. Оператор "точка" (.) применяется, когда операндом является сам объект или ссылка на него; а "стрелка"(->) – когда операндом служит указатель на объект:

```
#include "Screen.h"

boolisEqual( Screen& s1, Screen *s2 )
{ // возвращает false, если объекты не равны, и true - если равны

if (s1.height() != s2->height() ||
s2.width() != s2->width() )
return false;

for ( int ix = 0; ix < s1.height(); ++ix )
for ( intjy = 0; jy< s2->width(); ++jy )
        if ( s1.get( ix, jy ) != s2->get( ix, jy ) )
returnfalse;

returntrue;    // попали сюда? значит, объекты равны
}
```

isEqual() – это не являющаяся членом функция, которая сравнивает два объекта Screen. У нее нет права доступа к закрытым членам Screen, поэтому напрямую обращаться к ним она не может. Сравнение проводится с помощью открытых функций-членов данного класса.

Для получения высоты и ширины экрана isEqual() должна пользоваться функциями-членами height() и width() для чтения закрытых членов класса. Ихреализациятривиальна:

```
class Screen {
public:
int height() { return _height; }
int width()  { return _width; }
  // ...
private:
short _heigh, _width;
  // ...
};
```

Применение оператора доступа к указателю на объект класса эквивалентно последовательному выполнению двух операций: применению оператора разыменования (*) к указателю, чтобы получить адресуемый объект, и последующему применению оператора "точка" для доступа к нужному члену класса. Например, выражение

```
s2->height()
```

можно переписать так:

```
(*s2).height()
```

Результат будет одним и тем же.

ЛЕКЦИЯ № 15

Тема: Введение в объектно-ориентированное программирование. Перегрузка функций, конструкторы копирования и аргументы по умолчанию. Перегрузка операторов. **(2ч.)**

Цель занятия. *Научиться пользоваться с перегрузка функций, конструкторы копирования и аргументы по умолчанию. перегрузка операторов.*

Ключевые слова. Конструктор , аргумент , оператор.

План лекции:

1. Перегрузка функций
2. Конструкторы копирования и аргументы по умолчанию.
3. Перегрузка операторов.

ПЕРЕГРУЗКА ФУНКЦИЙ

При определении функций в своих программах вы должны указать тип возвращаемого функцией значения, а также количество параметров и тип каждого из них. В прошлом (если вы программировали на языке С), когда у вас была функция с именем *add_values,* которая работала с двумя целыми значениями, а вы хотели бы использовать подобную функцию для сложения трех целых значений, вам следовало создать функцию с другим именем. Например, вы могли бы использовать *add_two_values* и*add_three_values.* Аналогично если вы хотели использовать подобную функцию для сложения значений типа *float,* то вам была бы необходима еще одна функция с еще одним именем. Чтобы избежать дублирования функции, C++ позволяет вам определять несколько функций с одним и тем же именем. В процессе компиляции C++ принимает во внимание количество аргументов, используемых каждой функцией, и затем вызывает именно требуемую функцию. Предоставление компилятору выбора среди нескольких функций называется *перегрузкой.* В этом уроке вы научитесь использовать перегруженные функции. К концу данного урока вы освоите следующие основные концепции:

• Перегрузка функций позволяет вам использовать одно и то же имя для нескольких функций с разными типами параметров.

• Для перегрузки функций просто определите две функции с одним и тем же именем и типом возвращаемого значения, которые отличаются количеством параметров или их типом.

Перегрузка функций является особенностью языка C++, которой нет в языке С. Как вы увидите, перегрузка функций достаточно удобна и может улучшить удобочитаемость ваших программ.

ПЕРВОЕ ЗНАКОМСТВО С ПЕРЕГРУЗКОЙ ФУНКЦИЙ

Перегрузка функций позволяет вашим программам определять несколько функций с одним и тем же именем и типом возвращаемого значения. Например, следующая программа перегружает функцию с именем*add_values.* Первое определение функции складывает два значения типа*int.* Второе определение функции складывает три значения. В процессе компиляции C++ корректно определяет функцию, которую необходимо использовать:

```
#include <iostream.h>
```

```
intadd_values(inta,int b)

{
  return(a + b);
)

intadd_values (int a, int b, int c)

(
  return(a + b + c);
)

void main(void)

{
  cout<< "200 + 801 = " <<add_values(200, 801) <<endl;
  cout<< "100 + 201 + 700 = " <<add_values(100, 201, 700) <<endl;
}
```

Как видите, программа определяет две функции с именами *add_values*Первая функция складывает два значения типа *int,* в то время как вторая складывает три значения. Вы не обязаны что-либо предпринимать специально для того, чтобы предупредить компилятор о перегрузке, просто используйте ее. Компилятор разгадает, какую функцию следует использовать, основываясь на предлагаемых программой параметрах.

Подобным образом следующая программа MSG_OVR.CPP перегружает функцию *show_message.* Первая функция с именем *show_message* выводит стандартное сообщение, параметры ей не передаются. Вторая выводит передаваемое ей сообщение, а третья выводит два сообщения:

```
#include <iostream.h>
voidshow_message(void)
{
  cout<< "Стандартное сообщение: "<<"Учимся программировать на C++" <<endl;
}
voidshow_message(char *message)
{
  cout<< message <<endl;
}
voidshow_message(char *first, char *second)
{
  cout<< first <<endl;
  cout<< second <<endl;
}
void main(void){
  show_message();
  show_message("Учимсяпрограммироватьнаязыке C++!");
  show_message("В C++ нетпредрассудков!","Перегрузка - этокруто!") ;
}
```

Перегрузка операторов

C++ поддерживает перегрузку операторов (operatoroverloading). За небольшими исключениями большинство операторов C++ могут быть перегружены, в результате чего они получат специальное значение по отношению к определенным классам. Например, класс, определяющий связанный список, может использовать оператор + для того, чтобы добавлять объект к списку. Другой класс

может использовать оператор + совершенно иным способом. Когда оператор перегружен, ни одно из его исходных значений не теряет смысла. Просто для определенного класса объектов определен новый оператор. Поэтому перегрузка оператора + для того, чтобы обрабатывать связанный список, не изменяет его действия по отношению к целым числам.

Операторные функции обычно будут или членами, или друзьями того класса, для которого они используются. Несмотря на большое сходство, имеется определенное различие между способами, которыми перегружаются операторные функции-члены и операторные функции-друзья. В этом разделе мы рассмотрим перегрузку только функций-членов. Позже в этой главе будет показано, каким образом перегружаются операторные функции-друзья.

Для того, чтобы перегрузить оператор, необходимо определить, что именно означает оператор по отношению к тому классу, к которому он применяется. Для этого определяется функция-оператор, задающая действие оператора. Общая форма записи функции-оператора для случая, когда она является членом класса, имеет вид:

тип имя_класса::operator#(список_аргументов)
{
// действия, определенные применительно к классу
}

Здесь перегруженный оператор подставляется вместо символа #, а тип задает тип значений, возвращаемых оператором. Для того, чтобы упростить использование перегруженного оператора в сложных выражениях, в качестве возвращаемого значения часто выбирают тот же самый тип, что и класс, для которого перегружается оператор. Характер списка аргументов определяется несколькими факторами, как будет видно ниже.

Чтобы увидеть, как работает перегрузка операторов, начнем с простого примера. В нем создается класс three_d, содержащий координаты объекта в трехмерном пространстве. Следующая программа перегружает операторы + и = для класса three_d:

```
#include<iostream.h>
classthree_d {
int x, y, z; // трехмерные координаты
public:
three_doperators+(three_d t);
three_doperator=(three_d t);
voidshow ();
voidassign (intmx, intmy, intmz);
};
// перегрузка +
three_dthree_d::operator+(three_d t)
{
three_dtemp;
temp.x = x+t.x;
temp.y = y+t.y;
temp.z = z+t.z;
returntemp;
}
// перегрузка =
three_dthree_d::operator=(three_d t)
{
x = t.x;
y = t.y;
```

```
z = t.z;
return *this;
}
// вывод координат X, Y, Z
voidthree_d::show ()
{
cout<< x << ", ";
cout<< y << ", ";
cout<< z << "\n";
}
// присвоение координат
voidthree_d::assign (intmx, intmy, intmz)
{
x = mx;
y = my;
z = mz;
}
intmain()
{
three_d a, b, c;
a.assign (1, 2, 3);
b.assign (10, 10, 10);
a.show();
b.show();
c = a+b; // сложение a и b
c.show();
c = a+b+c; // сложение a, b и c
c.show();
c = b = a; // демонстрация множественного присваивания
c.show();
b.show ();
return 0;
}
```

Эта программа выводит на экран следующие данные:

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

Если рассмотреть эту программу внимательно, может вызвать удивление, что обе функции-оператора имеют только по одному параметру, несмотря на то, что они перегружают бинарный оператор. Это связано с тем, что при перегрузке бинарного оператора с использованием функции-члена ей передается явным образом только один аргумент. Вторым аргументом служит указатель this, который передается ей неявно. Так, в строке

```
temp.x = x + t.x;
```

x соответствует this->x, где x ассоциировано с объектом, который вызывает функцию-оператор. Во

всех случаях именно объект слева от знака операции вызывает функцию-оператор. Объект, стоящий справа от знака операции, передается функции.

При перегрузке унарной операции функция-оператор не имеет параметров, а при перегрузке бинарной операции функция-оператор имеет один параметр. (Нельзя перегрузить триадный оператор ?:.) Во всех случаях объект, активизирующий функцию-оператор, передается неявным образом с помощью указателя this.

Чтобы понять, как работает перегрузка операторов, тщательно проанализируем, как работает предыдущая программа, начиная с перегруженного оператора +. Когда два объекта типа three_d подвергаются воздействию оператора +, значения их соответствующих координат складываются, как это показано в функции operator+(), ассоциированной с данным классом. Обратим, однако, внимание, что функция не модифицирует значений операндов. Вместо этого она возвращает объект типа three_d, содержащий результат выполнения операции. Чтобы понять, почему оператор + не изменяет содержимого объектов, можно представить себе стандартный арифметический оператор +, примененный следующим образом: 10 + 12. Результатом этой операции является 22, однако ни 10 ни 12 от этого не изменились. Хотя не существует правила о том, что перегруженный оператор не может изменять значений своих операндов, обычно имеет смысл следовать ему. Если вернуться к данному примеру, то нежелательно, чтобы оператор + изменял содержание операндов.

Другим ключевым моментом перегрузки оператора сложения служит то, что он возвращает объект типа three_d. Хотя функция может иметь в качестве значения любой допустимый тип языка C++, тот факт, что она возвращает объект типа three_d, позволяет использовать оператор + в более сложных выражениях, таких, как a+b+c. Здесь a+b создает результат типа three_d. Это значение затем прибавляется к c. Если бы значением суммыа+b было значение другого типа, то мы не могли бы затем прибавить его к c.

В противоположность оператору +, оператор присваивания модифицирует свои аргументы. (В этом, кроме всего прочего, и заключается смысл присваивания.) Поскольку функция operator=() вызывается объектом, стоящим слева от знака равенства, то именно этот объект модифицируется при выполнении операции присваивания. Однако даже оператор присваивания обязан возвращать значение, поскольку как в C++, так и в C оператор присваивания порождает величину, стоящую с правой стороны равенства. Так, для того, чтобы выражение следующего вида

```
a = b = c = d;
```

было допустимым, необходимо, чтобы оператор operator=() возвращал объект, на который указывает указатель this и который будет объектом, стоящим с левой стороны оператора присваивания. Если сделать таким образом, то можно выполнить множественное присваивание.

Можно перегрузить унарные операторы, такие как ++ или --. Как уже говорилось ранее, при перегрузке унарного оператора с использованием функции-члена, эта функция-член не имеет аргументов. Вместо этого операция выполняется над объектом, осуществляющим вызов функции-оператора путем неявной передачи указателя this. В качестве примера ниже рассмотрена расширенная версия предыдущей программы, в которой определяется оператор-инкремент для объекта типа three_d:

```
#include<iostream.h>
classthree_d {
int x, y, z; // трехмерные координаты
public:
three_doperator+(three_d op2); // op1 подразумевается
three_doperator=(three_d op2); // op1 подразумевается
three_doperator++ (); // op1 также подразумевается
```

```
voidshow();
voidassign (intmx, intmy, intmz);
};
// перегрузка +
three_dthree_d::operator+(three_d op2)
{
three_dtemp;
temp.x = x+op2.x; // целочисленное сложение
temp.y = y+op2.y; // и в данном случае + сохраняет
temp.z = z+op2.z; // первоначальное значение
returntemp;
}
// перегрузка =
three_dthree_d::operator=(three_d op2)
{
x = op2.x; // целочисленное присваивание
y = op2.y; // и в данном случае = сохраняет
z = op2.z; // первоначальное значение
return *this;
}
// перегрузка унарного оператора
three_dthree_d::operator++()
{
x++;
y++;
z++;
return *this;
}
// вывести координаты X, Y, Z
voidthree_d::show()
{
cout<< x << ", ";
cout<< y << ", ";
cout<< z << "\n";
}
// присвоение координат
voidthree_d::assign (intmx, intmy, intmz)
{
x = mx;
y = my;
z = mz;
}
intmain()
{
three_d a, b, c;
a.assign (1, 2, 3);
b.assign (10, 10, 10);
a.show();
b.show();
c = a+b; // сложение a и b
c.show();
```

```
c = a+b+c; // сложение a, b и c
c.show();
c = b = a; // демонстрация множественного присваивания
c.show();
b.show ();
++c; // увеличение c
c.show();
return 0;
}
```

В ранних версиях C++ было невозможно определить, предшествует или следует за операндом перегруженный оператор ++ или --. Например, для объекта O следующие две инструкции были идентичными:

```
O++;
++O;
```

Однако более поздние версии C++ позволяют различать префиксную и постфиксную форму операторов инкремента и декремента. Для этого программа должна определить две версии функции operator++(). Одна их них должна быть такой же, как показано в предыдущей программе. Другая объявляется следующим образом:

```
locoperator++(int x);
```

Если ++ предшествует операнду, то вызывается функция operator++(). Если же ++ следует за операндом, то тогда вызывается функция operator++(int x), где x принимает значение 0.

Действие перегруженного оператора по отношению к тому классу, для которого он определен, не обязательно должно соответствовать каким-либо образом действию этого оператора для встроенных типов C++. Например, операторы << и >> применительно к cout и cin имеют мало общего с их действием на переменные целого типа. Однако, исходя из стремления сделать код более легко читаемым и хорошо структурированным, желательно, чтобы перегруженные операторы соответствовали, там где это возможно, смыслу исходных операторов. Например, оператор + по отношению к классу three_d концептуально сходен с оператором + для переменных целого типа. Мало пользы, например, можно ожидать от такого оператора +, действие которого на соответствующий класс будет напоминать действие оператора ||. Хотя можно придать перегруженному оператору любой смысл по своему выбору, но для ясности его применения желательно, чтобы его новое значение соотносилось с исходным значением.

Имеются некоторые ограничения на перегрузку операторов. Во-первых, нельзя изменить приоритет оператора. Во-вторых, нельзя изменить число операндов оператора. Наконец, за исключением оператора присваивания, перегруженные операторы наследуются любым производным классом. Каждый класс обязан определить явным образом свой собственный перегруженный оператор =, если он требуется для каких-либо целей. Разумеется, производные классы могут перегрузить любой оператор, включая и тот, который был перегружен базовым классом. Следующие операторы не могут быть перегружены:
. :: * ?

ЛЕКЦИЯ № 16

Тема: Введение в объектно-ориентированное программирование. Конструкторы и деструкторы.

(2ч.)

Цель занятия. *Научиться управлять конструкторы и деструкторы*
Ключевые слова. Конструктор , деструкторы.
План лекции:
1. Конструктор.
2. Деструкторы.

Перед использованием объекта может потребоваться инициализировать некоторые его данные. Для примера рассмотрим класс queue, определенный выше в этой главе. Перед тем как использовать queue, необходимо присвоить переменным rloc и sloc значения 0, используя функцию init(). Поскольку требование инициализации является чрезвычайно распространенным, то C++ позволяет производить инициализацию объектов во время их создания. Такая автоматическая инициализация выполняется с помощью функции, которая называется конструктором.

Функция-конструктор, являющаяся членом класса и имеющая имя, совпадающее с именем класса, представляет собой специальный тип функции. В качестве примера ниже показано, как выглядит класс queue, преобразованный таким образом, чтобы использовать для инициализации конструктор:

```
/ / создание класса очередь
classqueue {
int q[100];
intsloc, rloc;
public:
queue (); // конструктор
voidqput(int i);
intqget ();
};
```

Обратим внимание, что конструктор queue() не имеет типа возвращаемого значения. В C++ функции-конструкторы не могут возвращать значений.

Код, реализующий функцию queue(), выглядит следующим образом:

```
// конструктор
queue ::queue ()
{
sloc = rloc = 0;
cout<< "Queueinitialized. \n";
}
```

Обратим внимание, что выводимое сообщение «queueinitialized» служит для иллюстрации работы конструктора. В обычной практике большинство конструкторов не занимаются выводом или вводом данных. Они служат для инициализации.

Конструктор объекта вызывается автоматически при создании объекта. Это означает, что он вызывается тогда, когда происходит объявление объекта. В этом заключается важнейшее различие

между объявлениями в языке С и в С++. В С объявления переменных, попросту говоря, являются пассивными и в большинстве случаев выполняются во время компиляции. Иными словами, в языке С объявления переменных не являются исполнимыми инструкциями. В отличие от этого в С++ объявления переменных служат активными инструкциями, исполняемыми по существу во время работы программы. Одна из причин этого заключается в том, что объявление объекта может требовать вызова конструктора, то есть вызывать выполнение функции. Хотя это различие может показаться едва уловимым и носить больше академический характер, как будет видно далее, оно может оказывать существенное влияние на инициализацию переменных.

При инициализации глобальных или статических объектов конструктор вызывается только один раз. Для локальных объектов конструктор вызывается всякий раз, когда встречается объявление объекта.

Дополнением конструктора является деструктор. Во многих случаях перед уничтожением объекта необходимо выполнить определенные действия. Локальные объекты создаются при входе в соответствующий блок программы и разрушаются при выходе из него. Глобальные объекты уничтожаются при завершении работы программы. Имеется много причин тому, чтобы существовал деструктор. Например, может потребоваться освободить память, которая была ранее зарезервирована. В С++ за дезактивацию отвечает деструктор. Он имеет то же самое имя, что и конструктор, только к нему добавлен значок ~. Ниже представлен вариант класса queue, использующий конструктор и деструктор. (Следует иметь в виду, что класс queue не нуждается в деструкторе, поэтому ниже он приведен только для иллюстрации.)

```
// создание класса очередь
classqueue {
int q[100];
intsloc, rloc;
public:
queue(); // конструктор
~queue(); // деструктор
voidgput(int i);
intqget();
};
// конструктор
queue::queue()
{
sloc = rloc = 0;
cout<< "Queueinitialized.\n";
}
// деструктор
queue ::~queue ()
{
cout<< "Queuedestroyed.\n";
}
```

Для того чтобы продемонстрировать работу конструктора и деструктора, ниже представлена новая версия программы:

```
#include<iostream.h>
// создание класса очередь
classqueue {
int q[100];
intsloc, rloc;
public :
```

```
queue (); // конструктор
~queue(); // деструктор
voidgput(int i);
intqget();
};
/ / конструктор
queue::queue ()
{
sloc = rloc = 0;
cout<< "Queueinitialized.\n";
}
// деструктор
queue::~queue ()
{
cout<< "Queuedestroyed.\n";
}
voidqueue::qput(int i)
{
if (sloc == 99) {
cout<< "Queueisfull.\n";
return;
}
sloc++;
q[sloc] = i;
}
intqueue::qget()
{
if (rloc == sloc) {
cout<< "Queueunderflow.\n";
return 0;
}
rloc++;
return q[rloc];
}
intmain()
{
queue a, b; // создание двух объектов типа queue
a.qput(10);
b.qput(19) ;
a.qput(20);
b.qput(1);
cout<<a.qget ()<< " ";
cout<<a.qget ()<< " ";
cout<<b.qget()<<" ";
cout<<b.qget()<< "\n";
return 0;
}
```

Эта программа выводит на экран следующий текст:

```
Queueinitialized.
```

Queueinitialized.
10 20 19 1
Queuedestroyed.
Queuedestroyed.

ЛЕКЦИЯ № 16

Тема: Введение в объектно-ориентированное программирование. Управление доступом к членам базового класса.(2ч.)

Цель занятия. Научиться управлению доступом к членам базового класса

План лекции:

1. Управление доступом к членам базового класса

В предыдущих примерах базовый класс являлся общим базовым классом для производного класса:

classDerived: publicBase{…};

Это означает, что уровень доступа к членам класса Base для функций-членов класса Derived и просто пользователей класса Derived остался неизменным: личные члены класса Base не доступны в классе Derived, общие и защищенные члены класса Base остались соответственно общими и защищенными в Derived. Если не указать, что базовый класс является общим, то по умолчанию он будет личным:

class Derived: Base{…}; // Base - личныйбазовыйкласс.

Если базовый класс является личным базовым классом, то его личные члены по-прежнему недоступны ни в производном классе, ни для пользователя производного класса, а защищенные и общие члены базового класса становятся личными членами производного класса.

Базовый класс не может быть защищенным базовым классом.

Если базовый класс является личным, то для некоторых его членов в производном классе можно восстановить уровень доступа базового класса. Для этого их полное имя приводится в соответствующей части определения класса:

```
class Base {
private: intprivm;
protected: intprotm;
public: intpubm;
};
class Derived: Base{// Личныйбазовыйкласс.
public:
Base::pubm; // Теперьpubm - общийчленкласса Derived;
Base::protm; // ошибка - изменениеуровнядоступа.
protected:
Base::protm; //Теперьprotm - защищенныйчленкласса Derived;
Base::pubm; // ошибка - изменениеуровнядоступа.
```

Структуры могут использоваться подобно классам, но с одной особенностью. Если производным классом является структура, то ее

базовый класс всегда является общим базовым классом, т.е. объявление вида:

struct B: A {…};

Эквивалентно

class B: public A{…};

Если же производный класс строится на основе структуры, все происходит точно также, как и при использовании в качестве базового обычного класса. Таким образом, если и базовым, и производным классами являются структуры, то запись вида:

struct B: A{…};

эквивалентна

class B: public A {public: …};

ЛЕКЦИЯ № 18

Тема: Введение в объектно-ориентированное программирование. Контейнеры STL. Линейный контейнер -vector. Контейнер для строк -string. Множество элементов- set. Ассоциативный контейнер -map. **(2ч.)**

Цель занятия. *Понять Контейнеры STL. Линейный контейнер -vector. Контейнер для строк -string. Множество элементов- set. Ассоциативный контейнер -map.*

План лекции:

4. Контейнеры STL.
5. Линейный контейнер -vector.
6. Контейнер для строк –string.
7. Множество элементов- set.
8. Ассоциативный контейнер -map.

Кроме массивов в с++ существуют еще контейнеры, которые позволяют вам немного по другому хранить данные и, вдобавок, применять к ним различные функции (поиск, сортировка и т.д.) . Сегодня вы узнаете об одном из контейнеров - векторе (vector)

Возможно вы уже сталкивались с такой проблемой, что массивы в с++ имеют ограниченный размер, а мы точно не знаем количество элементов, необходимое в массиве. В таких случаях необходимо использовать динамическое программирование. Т.е. выделять память под элементы массива при необходимости добавить какой-либо элемент. В принципе, вс++ это можно реализовать вручную, но зачем? если есть специальный класс - vector. Он позволяет создавать нам массивы переменной длинны в зависимости от ситуации.

Для создания вектора вам необходимо подключить <vector>. Затем создание вектора почти ничем не отличается от создания переменной и/или массива:

vector<type>name; //здесьtype- тип данных в векторе, а name - имя вектора

Для записи в вектор достаточно набрать имя вектора.push_back(что положить)

```
vector<int> test;

test.push_back(10);

test.push_back(20);
```

Обращение к n-ому элементу ничем не отличается от обращения к элементу массива:

```
test[0]++;

cout<<test[1];

test[1]=222;
```

Для удаления последнего элемента вектора используется функция pop_back()

```
test.pop_back();
```

Еще немного полезных функций:

- test.at(i) - равносильно записи test[i], но при этом, если i-ого элемента не существует, программа не вылетит:)
- test.asign(n,m) - записывает в массив n элементов со значением m
- test.asign(start,end) - записывает в вектор значения от start до end.**Внимание!!!** start и end - интераторы (указатели) на элементы другого вектор.
- test.front() - возвращает ссылку на первый элемент
- test.back() - возвращает ссылку на последний элемент
- test.begin() - возвращает интератор первого элемента вектора
- test.end() - последнего + 1
- test.clear() - очищает вектор
- test.erase(i) или test.erase(start,end) - удаляет элемент с итератором i или элементы с интераторамимежду старт и енд
- test.size() - возвращает количество элементов в векторе
- test.swap(test2) - меняет местами содержимое вектора test и вектора test2
- test.insert(a,b) - вставляет в test переменную b перед элементом с интератором a и возвращает интератор вставленного элемента
- test.insert(a,n,b) - вставляет n копий b
- test.insert(a,start,end) - вставляет элементы между между итераторами start и end перед a

После прочтения этого куска текста у вас наверняка возник вопрос, а что собственно итератор и как ими пользоваться.

Итераторы

Итераторы являются собой, можно сказать, указателями на переменную. Они знают, где находится необходимая нам переменная и могут "добыть" её из памяти. Итераторы в основном используются дляоперация с элементами контейнеров: сортировка, поиск, копирование и т.д. Для создание итератора необходимо написать имя контейнера <тип данных> :: iterator и имя итератора.

Например,

vector<float>::iterator begin;

string::iteratorend,cur;

Да, да. Строки это тоже контейнеры;)

Теперь что мы можем делать с итераторами.

Мы можем получить элемент, на который они ссылаются:

cout<<*cur<<endl;

.

Здесь мы выводим элемент, на который указывает cur. Как вы наверное уже поняли, оператор * позволяет нам обращаться не к итераторы, а к элементу

Мы можем перейти к итераторы на следующий элемент или даже дальше:

cur++; // перейти к следующему элементу

cur+=10; // <=>cur=cur+10 перейти на 10 элементов вперед

Вот, например, вывод всего вектора на экран:

vector<string>test;

//как-то его заполнили

```
vector<string>::iterator cur;

for (cur=test.begin();cur<test.end();cur++)

cout<<*cur<<endl;
```

А теперь немного практики.

Во-первых, по изучайте все функции вектора и просмотрите как они работают. Также постарайтесь немного поработать с итераторами.

Во-вторых, попробуйте сделать такую программу:

вам вводят числа, вы их должны сохранить в вектор. Затем применить такую операцию для всех элементов начиная с 2:

```
a[i]+=a[i-1]*2+a[i]%a[i+1];
```

И вывести результат работы в текстовый файл

В следующий раз я расскажу об различных алгоритмах, которые можно применять к контейнерам

Множество элементов: `set`

Мы подошли к двум наиболее интересным с точки зрения изучения STL контейнерам: `set` и `map`. С которым из них стоит познакомиться в первую очередь -- вопрос, не имеющий однозначного ответа. Мнение автора заключается в том, что при академическом подходе к изучению STL, в первую очередь следует познакомиться с `set`, как с более простым контейнером из рассматриваемой пары. Всё, что можно сделать с `set`, можно сделать и с `map`, обратное же утверждение не всегда истинно. С алгоритмической точки зрения `map` является логическим продолжением `set`, в то время как многие программисты-практики зачастую смутно понимают назначение контейнера `set`, и всегда используют `map` , что менее элегантно и часто более сложно для понимания сторонними людьми.

Контейнер `set`, как уже было упомянуто, содержит множество элементов. Строго говоря, `set` обеспечивает следующую функциональность:

-- добавить элемент в рассматриваемое множество, при этом исключая возможность появления дублей;

-- удалить элемент из множества;

-- узнать количество (различных) элементов в контейнере;

-- проверить, присутствует ли в контейнере некоторый элемент.

Об алгоритмической эффективности контейнера `set` мы поговорим позже, вначале познакомимся с его интерфейсом.

```
set<int> s;

for(int i = 1; i <= 100; i++) {
s.insert(i); // добавим сто первых натуральных чисел
}

s.insert(42); // ничего не произойдёт ---
  // элемент 42 уже присутствует в множестве
```

```
for(int i = 2; i <= 100; i += 2) {
s.remove(i); // удалим чётные числа
}

// set::size() имееттип unsigned int
int N = int(s.size()); // N будетравно 50
```

У `set` нет метода `push_back()`. Это неудивительно: ведь такого понятия, как порядок элементов или индекс элемента, в `set` не существует, поэтому слово «back» здесь никак не применимо.

А раз уж у `set` нет понятия «индекс элемента», единственный способ просмотреть данные, содержащиеся в `set`, заключается в использовании итераторов:

```
set<int> S;
...
// вычисление суммы элементов множества S
int r = 0;
for(set<int>::const_iterator it = S.begin();
it != S.end(); it++) {
  r += (*it);
}
```

Если вы пользуетесь GNU C++, то TraversingMacros будет весьма кстати. Показательныйпример:

```
set< pair<string, pair<int, vector<int>>> SS;
...
int total = 0;
tr(SS, it) {
total += it->second.first;
}
```

Обратите внимание на синтаксис `it->second.first`. Ввиду того, что `it` является итератором, перед использованием его необходимо разыменовать. «Верным» синтаксисом было бы `(*it).second.first`. Однако, в C++ есть негласное правило, что если при описании некоторого объекта есть возможность обеспечить тождественное равенство конструкций `(*it).` и `it->`, то это следует сделать, дабы не вводить пользователей в заблуждение. Разработчики STL, конечно, позаботились об этом в случае с итераторами.

Основным преимуществом `set` перед `vector` является, несомненно, быстродействие. В основном это быстродействие проявляется при выполнении операции поиска. (При добавлении операция поиска также неявно присутствует, потому как дубли в `set` не допускаются). Однако, с операцией поиска в `set`/`map` есть существенный нюанс.

Нюанс заключается в том, что вместо глобального алгоритма `std::find(...)` следует использовать метод `setset::find(...)`.

Это не означает, что `std::find(...)` не будет работать с `set`. Дело в том, что `std::find(...)` ничего не знает о типе контейнера, с которым он работает. Принцип работы `std::find(...)` крайне прост: он просматривает все элементы до тех пор, пока либо не будет найден искомый элемент, либо не будет достигнут конец интервала. Основное преимущество `set` перед `vector` заключается в использовании нелинейной структуры данных, что существенно снижает алгоритмическую сложность операции поиска; использование же `std::find(...)` ануллирует все старания разработчиков STL.

Метод `set::find(...)` имеет всего один аргумент. Возращаемое им значение либо указывает на найденный элемент, либо равно итератору `end()` для данного экземпляра контейнера.

```
set<int> s;
...
if(s.find(42) != s.end()) {
        // 42 присутствует
}
else {
        // 42 не присутствует
}
```

Кроме `find(...)` существует также операция `count(...)`, которую следует вызывать как метод `set::count(x)`, а не как алгоритм `std::count(begin, end, x)`. Ясно, что `set::count(x)` может вернуть только 0 или 1. Некоторые программисты считают, что вышеприведённый код лучше выглядит, если использовать `count(x)` вместо `find(x)`:

```
if(s.count(42) != 0) {
  ...
}
```

Илидаже

```
if(s.count(42)) {
...
}
```

Мнение автора заключается в том, что подобный код вводит читателя в заблуждение: сам смысл операции `count()` несовместим со случаями, когда элемент либо присутствует, либо нет. Если же вам предтавляется слишком длинным каждый раз писать `"[некоторая форма find]" != container.end()`, сделайте следующие макросы:

```
#define present_member(container, element) \
      (find(all(container),element) != container.end())
#define present_global(container, element) \
      (container.find(element) != container.end())
```

Здесь `all(c)` означает `c.begin(),c.end()`

Более того, в соответствии с положением стандарта, которое называется «конкретизация шаблонов», можно написать следующий код:

```
template<typename T, typename T2>bool present(const T& c,
const T2&obj) {
return find(c.begin(), c.end(),
          (T::element_type)(obj)) != c.end();
}

template<typename T, typename T2>bool
present(const set<T>& c, const T2&obj) {
returnc.find((T::element_type)(obj)) != c.end();
}
```

Приработесконтейнеромтипа `set present(container, element)` вызоветметод `set::find(element)`, вдругихслучаях -- `std::find(container.begin(), container.end(), element)`.

Для удаления элемента из `set` необходимо вызвать метод `erase(...)`, передав ему один элемент -- элемент, который следует удалить, либо итератор, указывающий на удаляемый элемент.

```
set<int> s;
...
s.insert(54);
...
```

```
s.erase(29);
s.erase(s.find(57));
```
Как и полагается `erase(...)`, `set::erase(...)` имеет интервальную форму.
```
set<int> s;
...
set<int>::iterator it1, it2;
it1 = s.find(10);
it2 = s.find(100);
// Будет работать, если как 10, так и 100 присутствуют в множестве
if(...) {
s.erase(it1, it2); // при таком вызове будут удалены
   // все элементы от 10 до 100 не включительно
}
else {
   // сдвинем it2 на один элемент вперёд
   // set::iterator является normaliterator
   // операция += не определена для итераторов set'a,
   //но ++ и -- допускаются
   it2++;
s.erase(it1, it2); // а при таком --- от 10 до 100 включительно
   // приведённый код будет работать, даже если 100 был
   // последним элементом, входящим в set
}
```
Также, как и полагается контейнерам STL, у `set` есть интервальный конструктор:
```
int data[5] = { 5, 1, 4, 2, 3 };
set<int> S(data, data+5);
```
Кстати, данная функция `set` предоставляет эффективную возможность избавиться от дубликатов в `vector`:
```
vector<int> v;
...
set<int> s(all(v));
vector<int> v2(all(s));
```
Теперь `v2` содержит те же элементы, что и `v`, но без дубликатов. Приятной особенностью также является тот факт, что элементы `v2` упорядочены по возрастанию, но об этом мы поговорим позже.

В `set` можно хранить элементы любого типа, которые можно упорядочить. Об этом мы тоже поговорим позже.

ЛИТЕРАТУРА

1. Б.Ю.Ходиев, Б.А.Бегалов и др. Введение в базы данных и знаний. Тошкент.- 2003
2. Замулин А.В. Система программирования баз данных и знаний. Новосибирск : Наука, 1990.
3. Р. Джордейн. Справочник программиста персональных компьютеров типа IBM PC, XT и АП. М.: Финансы и статистика, 1992.
4. Безручко В.Т. Компьютерный практикум по курсу «Информатика»: Учебное пособие. 3-е изд., перераб. и доп. М.: ИД «ФОРУМ»: ИНФРА-М, 2009. -368 с.
5. Иванова ГС. Объектно-ориентированное программирование: Учебник. МГТУ. 2003. -320 с.
6. Смайли Джон. Учимся программировать на C++ вместе с Джоном Смайли. –СПб: ООО «ДиаСофтЮП», 2003.-560с.
7. Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильяме", 2005. — 1296 с.
8. Культин Н. Б. C/C++ в задачах и примерах. — СПб.: БХВ-Петербург, 2005. -288 с.
9. Подбельский В. В., Фомин С. С. Программирование на языке Си: Учеб. пособие. - 2-е доп. изд. - М.: Финансы и статистика, 2004. - 600 с
10. Долинский М. С. Решение сложных и олимпиадных задач по программированию: Учебное пособие. — СПб.: Питер, 2006. — 366 с.
11. Павловская Т.С. Щупак Ю.С. C/C++. Структурное программирование. Практикум.-СПб.: Питер,2007-240с
12. Павловская Т.С. Щупак Ю.С. C++. Объектно- ориентированное программирование. Практикум.-СПб.: Питер,2005-265с.
13. Романов Б.А. Практикум по программированию на C++: Учебное пособие. СПб.: BXB-Петербург, Новосибирск: Из-во НГТУ, 2006.-432с.
14. Шилдт Г. Искусство программирования на C++, - СПБ.: БХВ – Петербург, 2005, - 496 с.
15. Герб, Саттер. Новые сложные задачи на C++.: Пер. с анг. – Москва.: Издательский дом «Вильямс», 2005. – 272 с.:
16. Мозговой М. В. C++ Мастер-Класс. 85 нетривиальных проектов, решений и задач. – СПБ.: Наука и Техника, 2007. – 272 с.
17. Прата Стивен. Язык программирования C++. Лекции и упражнения. Учебник; Пер. с англ. – СПБ .; ООО «ДиаСофтЮП», 2005. – 1104 с.
18. Х. Дейтел. C# в подлиннике. Наиболее полное руководство Х. Дейтел Название: C# в подлиннике. Наиболее полное руководство. БХВ-Петербург, 2006
19. Мэтью Уилсон - Расширение библиотеки STL для C++. Наборы и итераторы. Издательство: ДМК Пресс, Б-П 608с
20. Макс Шлее - Qt4.5. Профессиональное программирование на C++. Издательство: БХВ-Петербург. 896с

21. Жасмин Бланшет, Марк Саммерфилд - Qt 4: Программирование GUI на C++ Изд. 2-е (+CD) Издательство: Кудиц-Пресс 2008. 736с
22. Герберт Шилдт - Swing: руководство для начинающих. Вильямс, 2007. 704с Шлее М. - Qt4. Профессиональное программирование на C++ , БХВ-Петербург, 2007, 831с
23. Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн - C++ Boost Graph Library. The Boost Graph Library: User Guide and Reference Manual,: Питер, 2006 г 304с,Переводчик: Р. Сузи
24. Д. Райан Стефенс, Кристофер Диггинс, Джонатан Турканис и Джефф Когсуэлл C++. Сборник рецептов C++ Cookbook Д. Райан Стефенс, Кристофер Диггинс, Джонатан Турканис и Джефф Когсуэлл - C++. Сборник рецептов. КУДИЦ-Пресс, 2007, 324с
25. Динман М.И. - C++. Освой на примерах. БХВ-Петербург, 2006, 384с.
26. Холзнер С. - Visual C++ 6. Учебный курс. Питер, 2007, 570с.
27. Герберт Шилдт - Искусство программирования на C++. БХВ, 2005, 474с.
28. Мэтью Уилсон - C++: практический подход к решению проблем программирования. Кудиц-образ, 2006, 736с
29. Каррано Ф.М., Причард Дж.Дж.- Абстракция данных и решение задач на C++. Стены и зеркала. М.: Вильямс, 2006, 848 с.\
30. Ермолаев В., Сорока Т. - C++ Builder: Книга рецептов, КУДИЦ-Образ, 2006, 208с.
31. Х. М. Дейтел, П. Дж. Дейтел - Как программировать на С, Бином-Пресс, 2006, 912 с.

Дополнительная литература

32. Стивен С. Скиена, Мигель А. Ревилла. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям / Пер. с англ. - М: КУДИЦ-ОБРАЗ, 2005. - 416 с.
33. Меньшиков Ф. В. Олимпиадные задачи по программированию. - СПб.: Питер, 2006. - 315 с.
34. Кнут Д. Искусство программирования. Том 1-4., СПб. Вильямс 2007.
35. Холзнер С. Visual C++ 6. Учебный курс. — СПб.: Питер, 2007. - 570 с.
36. Павловская Т.С. Щупак Ю.С. C/C++. Структурное программирование. Практикум.-СПб.: Питер,2007-240с
37. Павловская Т.С. Щупак Ю.С. C++. Объектно- ориентированное программирование. Практикум.-СПб.: Питер,2005-265с
38. Романов Б.А. Практикум по программированию на C++: Учебное пособие. СПб.: BXB-Петербург, Новосибирск: Из-во НГТУ, 2006.-432с.
39. Смайли Джон. Учимся программировать на C++ вместе с Джоном Смайли. –СПб: ООО «ДиаСофтЮП», 2003.-560с.
40. Пильшиков В.Н. Упражнения по языку Паскаль-М.: МГУ, 1986.
41. А. Фридман и др. Архив программ на C/C++ - М. Бином 2001- 638 с.
42. Абрамов С.А., Гнездилова Капустина Е.Н. и др. Задачи по программированию. - М.: Наука, 1988.

43. Брябрин В.М. Программное обеспечение персональных ЭВМ. –М.: Наука.,1989.-272с.
44. Вирт Н. Алгоритмы + структуры данных = программа.-М.:Мир,1985.-405с.
45. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT. -М.: Финансы и статистика, 1992.-544с.
46. Информатика. Базовой курс. Учебник для Вузов., Санк-Петербург, 2001. под редакцией С.В.Симоновича.
47. Нортон П. Программно-аппаратная организация IBM PC.-М.:Мир,1991.-327с.

Интернет ресурсы

1. www.ZiyoNET.uz - Узбекистан Республикаси ахборот-таълим портали.
2. www.intuit.ru
3. Виртуальный университет Евразии – http://virtual-university-eurasia.org/
4. http://www.opennet.ru
5. www.linux.org.ru
6. Энциклопедия поисковых систем http://www.searchengines.ru/
7. Павел Храмцов "Поиск и навигация в Internet". http://www.osp.ru/cw/1996/20/31.htm
8. How Intranet Search Tools and Spiders Work http://linux.manas.kg/books/how_intranets_work/ch32.htm
9. Martijn Koster "Robots in the Web: threat or treat?" http://info.webcrawler.com/mak/projects/robots/threat-or-treat.html