17. КЛАССЫ

17.1. Объявление классов

C++ – это язык, поддерживающий объектно ориентированное программирование, которое основано на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Размещение в одном компоненте **данных** и **методов**, которые с ними работают, в объектно ориентированном программировании называют **инкапсуляцией**.

Тип данных **класс** можно определить с помощью конструкции

ключ_класса, **имя_класса** { **список _членов** }; Здесь

ключ_класса – одно из служебных слов struct, union, class;

имя_класса – произвольный идентификатор;

список_членов – определения и описания членов класса – данных и функций.

Класс – это набор из одной или более переменных и функций, возможно, различных типов, сгруппированных под одним именем.

Пример структуры (struct) – учётная карточка сотрудника, в которой содержится фамилия, имя, отчество, адрес, должность, год поступления на работу и т. д. Некоторые из этих атрибутов сами могут оказаться структурами. Так, Ф.И.О. имеет три компоненты, адрес – также несколько компонент.

Класс может иметь имя, иногда называемое тегом. Тег становится именем нового типа в программе. Каждый член класса распознаётся по своему имени, которое должно быть уникальным в данном классе. Члены класса иногда называют его элементами или полями.

Хотя с каждым именем члена класса сопоставлен определённый тип, самостоятельным объектом такой член не является. Память выделяется только для конкретного объекта вновь определенного типа в целом. Введем новые типы FIO и sotrudnik:

```
struct FIO{ char familia [40],
            imya [30],  otchestvo
            [30]
};
struct sotrudnik{ FIO name;
            char dolgnost [30];
```

```
int year; double
oklad};
```

Здесь заданы два новых типа структурных переменных, и имена этих типов – FIO, sotrudnik. Заметим, что наличие ';' после фигурных скобок при объявлении класса обязательно.

После введения этих новых типов можно объявить структурные переменные типа FIO или sotrudnik обычным образом: FIO name1, name2, name3; sotrudnik s1, s2, s [50];

и компилятор выделит память под переменные name1, name2, name3, s1, s2 и под массив s из пятидесяти структур. Отметим, что число байтов, выделяемое под структурную переменную, не всегда равно сумме длин отдельных членов структуры из-за эффекта выравнивания, производимого компилятором.

Чтобы определить выделенное число байтов, надо воспользоваться операцией sizeof, например, так:

int nf = sizeof(FIO), ns = sizeof(sotrudnik);

Заметим также, что объявить структурные переменные можно одновременно с определением имени структуры:

```
struct DATE{
int day; int
month;
int year;
char mon_name[4]  }
d1, d2, d3;
```

Здесь объявлены три переменных d1, d2, d3, которые имеют тип структуры DATE.

Можно объявить структурную переменную и без введения имени (тега) структуры: struct{ int price;

```
double length [10] }
a, b, c, d;
```

После того, как определены структурные переменные, доступ к их членам осуществляется с помощью **операции извлечения** (или уточнения**)**, обозначаемым символом '.':

a.price c.length d1.day d3.mon_name s[25].oklad s[0].name.familia.

Имена наподобие c.length, d1.day, d3.mon_name, с помощью которых происходит доступ к членам класса, иногда называют уточненными именами. Если определить указатель на структуру, DATE* datep=&d1, то

обратиться к члену структуры d1 можно так: **(*datep).year**, или с помощью операции извлечения из указателя на структуру **->:**

datep->year, что эквивалентно.

Введем теперь простейший класс «комплексное число»:

```
struct compl {  double real, imag; void define ( double re = 0.0,
double im = 0.0 ){  real = re; imag = im;              // задание
комплексного числа.
}
void display( ){
cout << "real = " << real << ", imag = " << imag << "\n";
}
};
```

Здесь real, imag – **данные-члены**, а define(), display() – **функциичлены** класса. С точки зрения объектно ориентированного программирования их называют соответственно **полями** и **методами** класса.

Имя класса **compl** стало именем нового типа в программе.

Теперь можно описать объекты типа compl: compl a, b, c,

*pc = &c;

После этих определений данные-члены структурных переменных доступны в области их видимости:

```
a.define(3, 7);        // Определяется комплексное число 3 + 7i,
                       // т.е. a. real = 3; a.imag = 7;
b.define(2);           // определяется комплексное число 2 + 0i = 2;
c.define( );           // комплексное число = 0;
                       // оба параметра выбираются по умолчанию.
```

Данные-члены можно задавать и использовать непосредственно, не через функции define( ), display( ):

```
a.real = 3; a.imag = 7;  (*pc).real = 1; pc-
>imag = -1; a.real += b.real * 3 + 7; cout
<< "pc -> real: " << pc -> real << "\n";
a.display( );
b.display( );
c.display( );
```

Здесь данные-члены структуры доступны для использования в программе минуя функции-члены. Можно запретить произвольный

доступ к данным. Для этого обычно вместо слова struct в определении класса используют слово class:

```
class complex{  double
real;  double imag;
public: void display( ){
cout <<" real =" << real;
cout << ", imag =" << imag <<"\n";
} void define(double re = 0.0, double im =
0.0){ real = re; imag = im;} };
```

Метка **public**, которая может присутствовать в объявлении класса, в нашем примере делит его тело на две части – «личную», или «собственную» (private), и общую – (public).

Доступ к данным-членам класса, находящимся в собственной части, возможен лишь через функции-члены класса:

complex s1, s2; complex *ps = &s1; s1.define( ); // s1.real = 0; s1.imag = 0; s1.display( ); // Выводится real = 0, imag = 0; ps->display( ); // то же самое. s1.real = 3; // Ошибка! private-член s2.real недоступен! В определении класса может также явно присутствовать метка private.

Метки private и public делят тело класса в общем случае на части, различающиеся по уровню доступа. К членам класса, находящимся в собственной (private) части, доступ возможен только с помощью функций-членов и так называемых **дружественных** или **привилегированных** функций.

К общим же членам класса можно обратиться из любой функции программы.

Основное отличие struct и class состоит в уровне доступа по умолчанию. Если нет явного указания уровня доступа, то все члены структуры считаются общими, а класса – собственными. Явное указание уровней доступа делает слова struct и class равнозначными. Обычно использование слова struct вместо class говорит о том, что в ограничении уровня доступа к данным нет необходимости (здесь предполагается, что все члены структуры общие).

Заметим, что типы, созданные программистом с помощью механизма классов, часто называют **абстрактными типами данных**.

17.2. Конструкторы

В предыдущем примере инициализация объектов типа complex производилась с помощью функции-члена define ( ). При этом

переменная s2 осталась неинициализированной. В C++ предусмотрены специальные функции-члены класса, которые в большинстве случаев вызываются не программистом, а компилятором и которые предназначены для инициализации объектов абстрактных типов. Такие функции называются **конструкторами**. Рассмотрим пример:

```
class cl{  int num; public:
void  set(int i){ num = i; }
void show( ){ cout << "Число: "  << num << "\n"; }
};
```

Чтобы использовать объект такого типа, его надо объявить, инициализировать, а затем уже использовать:

```
void f( ){
cl obj;                 // Объект создан.
obj.set (10);           // Объект инициализирован.
obj.show( );            // Объект можно использовать.
}
```

Теперь используем для инициализации конструктор. Это просто специальная функция – член класса cl, имя которой обязательно совпадает с именем класса: class cl{ // Конструктор.

```
int num; public:
cl( int i ){  num
= i;
}  void show(
){
cout << "Число: "  << num << '\n';} };
```

Заметим, что для конструктора никогда не указывается тип результата!

Функция, использующая этот класс, примет вид: void f( ){ cl obj(10); // Объект создан и инициализирован! obj.show( ); // Здесь объект obj используется!
}

Возможна другая, полная форма записи объявления объекта абстрактного типа, имеющего конструктор:

```
cl obj = cl(10);
```

В этом примере конструктор – так называемая инлайн-функция (inline), так как его определение находится в теле класса. Однако его можно представить и как обычную функцию, для чего в классе

конструктор только объявляется, а определяется он вне тела класса с использованием квалифицированного имени:

```
class cl{  int
num;
public:
cl ( int i );
void show( ){
cout << "Число: " << num << '\n';
} };
cl::cl( int i ){// Полное, или квалифицированное, имя.
num = i; }
```

Часто бывает удобно предусмотреть несколько путей инициализации, используя механизм перегрузки функций.

Приведем пример программы, в которой происходит вывод строки символов на экран.

```
#include<windows.h> // Заголовочный файл для WinApi –
// совокупности различных библиотек ОС Window.
#include<cstring> const
unsigned short
Black = 0,
Blue = 1,
Green = 2,
Cyan = 3,
Red = 4,
Magenta = 5,
Brown = 6,
LightGray = 7,
DarkGray = 8,
LightBlue = 9,
LightGreen = 10,
LightCyan = 11,
LightRed = 12,
LightMagenta = 13,
Yellow = 14,
White = 15; class
mystring{ char
*str;
```

```
unsigned short text, background; int
row, col;
void SetColor (unsigned short text, unsigned short background){ void*
out = GetStdHandle(STD_OUTPUT_HANDLE);
```

/* Функция GetStdHandle извлекает дескриптор для стандартного ввода-вывода данных. Консольный процесс использует так называемые дескрипторы для того, чтобы обратиться к буферу ввода и вывода данных и экранным буферам консоли.*/

```
SetConsoleTextAttribute(out, (background << 4 | text));
/* Функция SetConsoleTextAttribute  устанавливает
атрибуты символов,
записанных в экранный буфер консоли */
}
void gotoxy(int x, int y){
COORD coord;
coord.X = x;
coord.Y = y;
SetConsoleCursorPosition
(GetStdHandle(STD_OUTPUT_HANDLE), coord);
```

/* Функция SetConsoleCursorPosition устанавливает позицию курсора в заданном экранном буфере консоли.

```
Структура COORD содержит новую позицию курсора. */
} public:
mystring();
mystring(char *, unsigned short, unsigned short, int = 0, int = 0); void
write();
};
```

/* Конструктор без аргументов. Определяются все данные объекта – строка, видеоатрибут ее символов и позиция для вывода на экран: */
mystring::mystring(){ str = new char[sizeof "Здравствуйте!"]; strcpy(str, "Здравствуйте!"); text =LightBlue; background=LightRed;

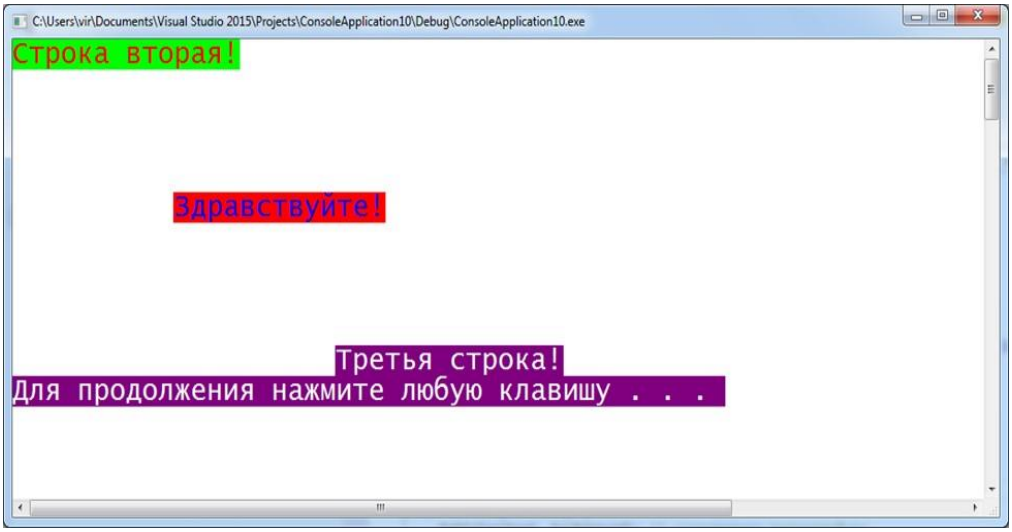
```
row = 5;
col = 10; }
mystring::mystring(char *line, unsigned short tex, unsigned short back,
int  y,  int  x){  str  =  new
char[strlen(line) + 1];
strcpy(str, line); // копируем строку из line в str
text = tex; background = back;
```

```
row = y; col = x; }
void mystring::write(){
SetColor(text, background); // установление видеоатрибута.
gotoxy(col, row); cout << str << "\n";
} void main( ){
setlocale(0, "rus");
system("color F0");        // Установка белого фона и черного текста
mystring string1;          // Эквивалентно string string1=string(); //
Написать string string(); нельзя, так как это – прототип функции!

mystring string2("Строка вторая!", LightRed,  LightGreen); mystring
string3("Третья строка!", White , Magenta, 10, 20);

// Печать строк:

string1.write ( );
string2.write ( );
string3.write ( ); }
```

В случае вызова первого конструктора – без аргументов – инициализация любого объекта будет происходить всегда совершенно одинаково, теми значениями, которые жестко определены в этом конструкторе. В данном случае объект string1 инициализирован конструктором без аргументов и при вызове функции string.write( ) произойдет печать строки **Здравствуйте!** голубого цвета на красном фоне в 5-й строке начиная с 10-й позиции.

Объекты string2 и string3 инициализируются другим конструктором.

Результат работы программы:



Как и для других перегружаемых функций, выбор требуемого конструктора осуществляется по числу и типу аргументов.

Отметим, что в классе может быть только один конструктор с умалчиваемыми значениями параметров.

17.3. Деструкторы

Важную роль наряду с инициализацией объектов абстрактных типов имеет и обратная к ней операция – удаление таких объектов. В частности, конструкторы многих классов выделяют память под объекты динамически, и после того как необходимость в таких объектах отпадает, их рекомендуется удалить.

Это удобно выполнить в **деструкторе** – функции, которая вызывается для объекта абстрактного типа, когда он выходит из области существования. В рассмотренном выше примере место для хранения строк в памяти выделяется динамически, поэтому полезно определить деструктор. Имя деструктора, как и конструктора, не может быть произвольным, оно образуется символом ~ и именем класса (дополнение к конструктору): class string{ . . . public:

~ string( ){ delete str; }

. . .};

Здесь деструктор очень прост. Он может быть сложнее и оформлен в виде аутлайн-функции. Деструктор никогда не может иметь аргументов. Отметим, что нельзя получить адрес ни конструктора, ни деструктора. Вызов конструктора происходит автоматически во время определения объекта абстрактного типа. Вызов деструктора происходит автоматически при выходе объекта из области своего существования. Деструктор может быть вызван и явно с обязательным указанием его полного имени. Отметим также, что для класса, в котором явно не определен ни один конструктор, компилятор самостоятельно генерирует так называемый конструктор по умолчанию, не имеющий аргументов, с уровнем доступа public. То же относится и к деструктору.

Отметим, что данные класса не обязательно должны быть определены или описаны до их первого использования в принадлежащих классу функциях. То же самое справедливо и для принадлежащих классу функций, то есть обратиться из одной функции класса к другой можно до ее определения внутри тела класса. Все компоненты класса видны во всем классе.

17.4. Статические члены класса

Данное-член класса можно объявить со служебным словом static. Память под такие данные резервируется при запуске программы, то есть еще до того, как программист явно создаст первый объект данного абстрактного типа. При этом все объекты, сколько бы их ни было, используют эту заранее созданную одну-единственную копию своего статического члена. Статический член класса должен быть инициализирован после определения класса и до первого описания объекта этого класса с помощью так называемого полного, или квалифицированного, имени статического члена, которое имеет вид **имя_класса::имя_статического_члена.**

Статический член класса (уровня доступа public) можно использовать в программе не только с помощью уточненного имени, но и через квалифицированное имя.

Например, создадим класс object, в статическом члене которого хранится число существующих в каждый момент времени объектов типа object. class object{ char *str; public:

```
static int num_obj; object( char
*s){ // Конструктор. str = new
char[strlen(s) + 1];
strcpy( str, s );
cout <<"Создается " << str <<"\n";  num_obj ++ ;
}
~ object( ){ cout <<"Уничтожается " <<str << "\n"; delete
[]str;
num_obj --;}
};
int object::num_obj = 0;   // Инициализация. Об этом говорит  //
                                ключевое слово  int!
object s1 ("первый  глобальный объект. ", s2
("второй  глобальный объект. ");
void f( char *str ){ object
s( str );
cout << "Всего объектов – " << object::num_object<< ".\n"; cout
<< "Проработала функция  f()"  << ".\n";}
void main( ){
```

```
cout << "Пока объектов – " << object::num_obj << ".\n"; object
m("объект в main( ). ");
cout << "А сейчас объектов – " << m.num_obj << ".\n";
f("локальный объект. "); f("другой локальный объект.
");
cout <<"Перед окончанием main() объектов – "
<< s1.num_obj << ".\n"; }
```

Результаты работы программы:

```
Создается первый глобальный объект.
Создается второй глобальный объект.
Пока объектов – 2.
Создается объект в main( ).
А сейчас объектов – 3.
Создается локальный объект.
Всего объектов – 4.
 Проработала функция f().
Уничтожается локальный объект.
Создается другой локальный объект.
Всего объектов – 4.
Проработала функция f().
Уничтожается другой локальный объект.
Перед окончанием main( ) объектов – 3.
Уничтожается объект в main( ).
Уничтожается второй глобальный объект.
Уничтожается первый глобальный объект.
```

Обратим внимание, что конструкторы для глобальных объектов вызываются **до** функции main(), а деструкторы **после** main( ).

Отметим, что классы, определенные внутри функции, не могут иметь статических членов.

17.5. Указатель *this*

Рассмотрим пример:

```
class str{ char * string; public:
void set(char *text){string = text;}
void write(){
cout << "Строка: " << string << "\n";}
```

}; void main(){ str str1,
str2;
str1.set((char*)"Привет!");
str2.set((char*)"Hello!");
str1.write(); str2.write(); } В результате выполнения этой программы на экране появится следующее:

```
Строка: Привет! Строка:
Hello!
```

Зададимся вопросом: как функция-член write узнает, для какого именно объекта она вызвана? Функция-член определяет, для какого объекта она вызвана, так как ей в качестве неявного первого аргумента передается адрес этого объекта. В данном случае – указатель типа str*.

Внутри функции-члена класса можно явно использовать этот указатель. Он всегда имеет имя **this** (ключевое слово).

Перед началом выполнения кода функции указатель this инициализируется значением адреса объекта, для которого вызвана данная функция-член. Таким образом, приведенное выше определение функции str**::**write() представляет собой сокращенную форму следующей записи: void write(){

cout << "Строка : "<<this->string << "\n";}

Отметим, что явное присваивание указателю this некоторого значения запрещено.

17.6. Статические функции-члены

Перед объявлением функции-члена класса можно поставить служебное слово static. Особенностью таких статических функций-членов является следующее: как и к статическому данному-члену класса, к ней можно обратиться еще до того, как в программе создан первый объект такого класса. Статические функции-члены (компонентные функции) позволяют получить доступ к частным статическим данным-членам класса, не имея еще ни одного объекта данного типа в программе.

Для статической компонентной функции не определен указатель this. Когда это необходимо, адрес объекта, для которого вызывается статическая функция-член, должен быть передан ей явно в виде аргумента. Пример: class prim{ int numb;

static stat; public:

prim(int i){ numb
= i;}

/* Далее – статическая функция. Указатель this не определен и выбор объекта осуществляется по явно переданному указателю.

Член stat не требует указателя на объект, так как он общий для всех объектов класса prim. m*/ static void func(int i, prim *p = 0){ if(p) p -> numb = i; else stat = i;

}

static void show( ){

/* Статическая функция обращается только к статическому члену класса, никаких указателей не требуется: */ cout << "stat = "<< stat << "\n";

}

}; // Конец класса prim. int prim::stat = 8;

// Инициализация статического члена класса. void main( ){

/* До создания объектов типа prim возможен единственный способ обращения к статической функции-члену: */ prim::show();

// Можно изменить значение статического члена класса: prim::func(10);

/* После создания объекта типа prim можно обратиться к статической функции обычным для абстрактных типов способом: */ prim obj(23); // obj.numb становится равным 23. obj.show();

// Можно изменить значение

// созданного объекта:

prim::func(20, &obj); // obj.numb равно 20. obj.func(27, &obj); // obj.numb равно 27.

}

17.7. Указатели на члены класса

Для членов класса (кроме битовых полей) определена операция получения адреса. Указатели на данные-члены класса никаких особенностей не имеют. Особенностью указателя на функцию-член класса является явное присутствие в его объявлении имени класса, за которым следует **::**. class cl{. . . public: int f(char*, int); void g();

. . .

};

Как и для указателя на обычную функцию, при объявлении указателя на функцию-член класса необходимо объявить типы

результата и аргументов функции, на которую заводится указатель. Как обычно, указатель можно инициализировать при объявлении:

```
int (cl ::*fptr) (char *, int) = cl::f;
```

Пример:

```
struct s{int  mem; s(int
a){mem = a;}
void func(int a){cout << a + mem << '\n';}
};          void
main(){
void (s::*fp)(int) = s::func;
s obj(5); s *p
= &obj;
// Два варианта вызова функции-члена по указателю – (obj.*fp)(6);
// используя объект  obj  типа s
(p ->* fp)(9);          // и указатель  p  на него. }
```

Здесь **.*** (как и **->***) являются символами одной-единственной операции, а не находящимися рядом символами двух ранее знакомых нам операций "." ("->") и *. Правым операндом операций **.*** и **->*** обязательно должен быть **указатель на член класса**, а не любой указатель.

17.8. Инициализация данных-членов класса

Инициализация членов абстрактных типов

Пусть класс содержит в себе члены абстрактных типов. Особенностью их инициализации является то, что она выполняется с помощью соответствующего конструктора. Рассмотрим класс:

```
class coord { double x,
y, z; public: coord (){ x
= y = z =0; }
coord(double xv, double yv, double zv=0){ x = xv; y = yv; z = zv; }
coord(coord & c){ x = c.x; y = c.y; z = c.z; }
}; class triang{ coord vert1, vert2, vert3; // Координаты вершин
треугольника.
public: triang();
triang( coord &v1, coord &v2, coord &v3 );
};
```

При инициализации некоторого объекта класса triang потребуется три раза вызвать конструкторы для его вершин – объектов типа coord. Для этого в определении конструктора класса triang после двоеточия нужно поместить список обращений к конструкторам класса coord: Traing::triang(coord &v1, coord &v2, coord &v3): vert1 (v1), vert2 (v2), vert3 (v3){. . .}

Вызов конструкторов класса coord происходит до выполнения тела самого конструктора класса triang. Порядок их вызова определяется порядком появления объявлений членов типа coord при создании класса triang.

Класс coord содержит конструктор без аргументов. Вместо записи при обращении к такому конструктору

triang::triang(): vert1(), vert2(), vert3(){. . .} допускается

написать просто так:

triang::triang(){. . .}

Инициализация констант

Если среди данных-членов класса имеются члены, описанные с модификатором const, то при инициализации используется та же форма записи конструктора, что и в случае с данными абстрактных типов:

```
class cl{ int v; const int c;
public: cl (int a, int b): c
(b){v = a;} };
```

Константу можно инициализировать только в конструкторе, попытка сделать это любым другим способом (например, с помощью другой компонентной функции) приведет к сообщению об ошибке. Инициализация констант в теле конструктора тоже недопустима.

Заметим, что способ записи конструктора, обязательный для констант и данных абстрактных типов, можно использовать и для обычных членов класса: class ro{ int var; const int c; public: ro(int v, int u): c(u), var (v){}

};

17.9. Конструктор копирования и операция присваивания

При работе с объектами абстрактных типов может возникнуть ситуация, когда один объект должен являться копией другого. При этом возможны два варианта:

1. *Вновь создаваемый* объект должен стать копией уже имеющегося.

2. Нужно скопировать один объект в другой, причем *оба были созданы ранее*.

В первом случае используется **конструктор копирования**, во втором – **операция присваивания**.

Конструктор копирования – это конструктор, первым аргументом которого является ссылка на объект того типа, в котором этот конструктор объявлен.

```
class cl{. . . cl(cl&);        //        Конструктор
копирования.
. . .
};
cl ca;          // Здесь используется конструктор без аргументов. cl
cb = ca;        // Используется конструктор копирования.
```

Инициализация копированием происходит и при передаче функциям их аргументов и при возврате результата. Если аргумент или возвращаемое значение имеет абстрактный тип, то неявно вызывается конструктор копирования, как это было в примере с классами coord и triang.

Конструктор копирования генерируется компилятором самостоятельно, если он не был написан программистом. В этом случае создается точная копия инициализирующего объекта, что требуется далеко не всегда. Пример 1:

```
class cl{int num; float val; public:
cl(int i, float x){num = i; val = x;}
};


void  main(){  cl
obj1(10, 20.3);
// Для создания объектов obj2 и obj3 // используется
конструктор копирования по умолчанию:
cl obj2(obj1); cl
obj3 = obj2; }
```

Пример 2:

```
class prim{int n; float v;
public: prim(int i, float
x){ n = i;  v = x; }
```

```
prim (const prim &obj, int i = 0){
if(i) n = i; else n = obj.n;   v =
obj.v;
} }; void main(){
prim obj1(10, 23.5);
/* Для создания
объектов obj2 и
obj3 используется
явно описанный
конструктор
копирования:
*/
prim obj2 = obj1;
prim obj3(obj1, 12);
}
```

Теперь внесем минимальные изменения, и компилятор будет вынужден в дополнение к имеющемуся конструктору копирования добавить свой:

```
class prim{ int n;
float v; public:
prim(int i, float x){
n = i;  v
= x; }
prim(const prim &obj, int i){
n = i;  v =
obj.v;
}
};

void  main(){  prim
obj1(10, 23.5);
/* Сейчас будет использован конструктор копирования
по умолчанию */ prim obj2 = obj1;
// а сейчас будет использован явно определенный конструктор,  //
причем копируется лишь часть объекта:
prim obj3(obj, 12); }
```

Отметим, что модификатор const используется для предотвращения изменения копируемого объекта.

Объект одного класса можно инициализировать значением объекта другого класса. При этом конструктор не является конструктором копирования, так как в качестве аргумента в нем фигурирует ссылка на объект другого класса:

```
struct s1{int i; float x; s1(int
j, float y){
i = j;
x = y;
} };


struct s2{ int i;  float x; s2(const s1& a){              // Это не
конструктор копирования! i = a.i;   x = a.x;
}  };


void main(){ s1
obj1 (1, 3.7); s2
obj2 (obj1); }
```

В отличие от конструктора копирования, операция присваивания используется тогда, когда объекты, являющиеся операндами этой операции, уже существуют. Операция присваивания, наряду с операцией получения адреса, предопределена для объектов абстрактных типов по умолчанию, и ее можно использовать без каких-либо дополнительных действий со стороны программиста.

```
class cl{. . . }; void f(){ cl obj1;  cl obj2 = obj1;              //
Используется конструктор копирования. cl obj3; obj3 = obj1;
// Присваивание!
}
```

Не всегда требуется при выполнении присваивания просто создавать копию. Если требуется нечто иное, то нужно переопределить операцию присваивания для класса.

17.10. Дружественные функции

Могут встретиться ситуации, когда желательно иметь доступ к личным данным класса, минуя функции-члены. Наиболее распространена ситуация, когда функция-член одного класса должна иметь доступ к личным членам другого.

Рассмотрим снова пример с классами coord и triang.

```
class coord{
double x, y, z;
public: coord();
coord(double, double, double = 0);
coord(coord & c); };


class triang{ coord
vert1, vert2, vert3;
public:  triang();
triang(coord &v1, coord &v2, coord &v3);
};
```

Пусть нам необходимо добавить в класс triang функцию-член, вычисляющую координаты центра треугольника.

Язык предоставляет для некоторых функций, как обычных, так и членов некоторого класса X, возможность получения доступа к личным членам класса Y. Такая функция называется **привилегированной в классе** Y. Говорят также, что класс **X** является **дружественным** классу **Y**.

Для объявления привилегированной функции используется служебное слово **friend**. Чтобы функция стала привилегированной в классе Y, она должна быть объявлена в этом классе как дружественная функция.

Напишем три функции-члена класса triang, вычисляющие координаты центра треугольника по каждой из осей: double triang::midx(){

return (vert1.x + vert2.x + vert3.x)/3;

} и аналогично triang::midy (), triang::midz

().

Для того чтобы компилятор не выдал сообщение об ошибке, необходимо добавить в объявление класса coord, в любой его части, следующие объявления:

```
class coord{
…
friend triang::midx();
friend triang::midy();
friend triang::midz();
}
```

Достаточно распространенным является случай, когда все функции-члены одного класса являются привилегированными в другом; предусмотрена даже упрощённая форма записи:

```
class coord{… friend
triang;
…
};
или        class
coord{…
friend class triang;
…};
```

В этом случае говорят, что класс triang является дружественным классу coord.

Заметим, что для дружественных функций не определён указатель this, они не имеют неявных аргументов, для них не определены уровни доступа. Одна и та же функция может быть объявлена привилегированной сразу в нескольких классах.

Разницу в способе использования функций-членов и дружественных функций покажем в следующем примере:

```
class cl{ int
numb;
friend void f_func (cl*, int);
                                        // f_func() – не личный член класса,
                                        // хотя объявляется в private-части.
public: void m_func(int);
};
void f_func(cl* cpt, int i){
cptr->numb = i;                         // Нужен явный указатель на объект,
// так как указатель this не определён!
}  void  cl::m_func(int
i){
numb = i;                               // То же, что this -> numb = i;
}          void
main(){
cl obj; f_func(&obj,
10);
obj.m_func(10);
```

```
                    // Сравните способы вызова и аргументы функций!
…}
```

Следующий пример демонстрирует возможность доступа к статическим личным членам класса ещё до создания хотя бы одного объекта этого класса. class cl{ static int num; public: void set(int i){ num = i;}

```
void m_show(){ cout
<< num << "\n";
}       friend       void
f_show(){
cout << cl::num << "\n";}
}; int cl::num =
8; void main(){
cout << "Объектов типа cl  нет.\n"; cout << "Статический член
класса = "; // Пока можно использовать только
дружественную функцию:
f_show(); cl
obj;
obj.set(200);
cout << "Создан объект типа cl.\n"; cout <<
"Статический член класса = "; // Теперь можно
использовать и функцию-член. obj.m_show();
}
```

17.11. Конструктор и операция *new*

Если абстрактный тип имеет конструктор без аргументов, то обращение к операции new полностью совпадает с тем, что используется для выделения памяти под обычные типы данных без инициализирующего выражения.

```
class integer{int i;};


void main(){ integer *ptr =
new integer;
 . . .
}
```

Если же конструктор класса integer имеет аргументы, то список аргументов помещается там же, где при работе со стандартными типами

данных находится инициализирующее выражение. class integer{ int i; public: integer();

integer(int j): i(j){}

}; void main(){ int *ip = new

int(10); integer *iptr = new

integer(30); }

Если в операции new происходит обращение к конструктору без аргументов, то допустимы следующие формы записи: integer *ip1 = new integer(); integer *ip2 = new integer;

Если конструктор без аргументов для класса X не определён, то при попытке выполнить оператор

X *xp = new X;

компилятор выдаст сообщение об ошибке. В этом случае требуется явно определить конструктор без аргументов.

17.12. Вызов деструктора

Вызов деструктора для объекта абстрактного типа производится автоматически при его выходе из области существования. Для локальных переменных деструктор вызывается при выходе из блока, в котором эта переменная была объявлена. Для глобальных переменных вызов деструктора является частью процедуры завершения программы, выполняемой после функции main(). Выход указателя на объект абстрактного типа из области существования этого указателя не приводит к вызову деструктора для объекта, на который он указывает. Надо различать указатели на объект, созданные при помощи операции new, и другие объекты.

Рассмотрим пример с указателем на автоматический объект абстрактного типа: class cl{ int num;

public: cl

(int i){

num = i;

}

~cl(){} };

void main(){

// Создание объекта obj типа cl: cl

obj (1);

```
// Создание указателя ptr на объект класса cl и его инициализация //
адресом, создаваемой здесь же безымянной переменной типа cl: cl
*ptr = &cl (2);
{// Указатель tmp в блоке относится к тому же объекту, что и ptr. cl
*tmp = ptr;
}
}
```

В этом случае как конструктор, так и деструктор будут вызываться дважды. Сначала вызывается конструктор для объекта obj, затем конструктор для безымянного объекта, на который указывает ptr. При выходе из внутреннего блока указатель tmp теряется, однако сам объект сохраняется. При завершении main() в первую очередь вызывается деструктор для безымянного объекта, а затем – деструктор для obj.

Если указатель относится к объекту абстрактного типа, созданному динамически, то деструктор для него вызывается в операции delete:

```
class cl{
int num;
public:
cl (int i){
num = i;
} ~cl(){
cout << "Деструктор класса cl. \n";
};

void  main(){  cl
*ptr = new cl(1);
…
delete ptr;
}
```

Перед удалением из памяти объекта, на который указывает ptr, для него будет вызван деструктор. В результате на экране появится строка

деструктор класса cl.

Вызов деструктора можно осуществить явно по его полному имени:

```
class cl{ int num; public: cl (int i){ num = i; }
~cl(){}
}; void main(){
cl obj(1); cl *ptr
= &cl(2);
```

obj.cl::~cl(); ptr
-> cl::~cl(); }

Также можно вызвать деструктор и для динамического объекта.

cl *ptr = new cl(1); ptr
-> cl::~cl();

Отметим, что явный вызов деструктора не отменяет его автоматический вызов. class X{ int *ip; public: X(int y){ ip = new int(y);
}
~X(){
cout << "Деструктор класса X;\n";
delete ip;
} };

void main(){ X *xp = new X(5); xp -> X::~X(); //
Явный вызов деструктора. delete xp; //
Вызов деструктора из delete.
}

В результате получим два сообщения, если не произойдёт зацикливания при повторной операции delete, применённой к одному и тому же указателю ip:

```
Деструктор класса X;
Деструктор класса X;
```

Пользоваться явным вызовом деструктора надо очень осторожно.