СОДЕРЖАНИЕ

ГЛАВА 1

Новые возможности Си++, не связанные с ООП.

Язык Си++ отличается от обычного Си в первую очередь поддержкой объектно-ориентированного программирования (ООП). Однако в нем есть ряд очень полезных нововведений, не связанных непосредственно с ООП, с которых хорошо начинать изучение языка. Как мы увидим, даже привычные, не объектно-ориентированные программы удобнее писать на Си++, чем на обычном Си.

Комментарии

В Си++ можно использовать два вида комментариев: обычные, оформленные по правилам Си, и однострочные, начинающиеся с символов // и продолжающиеся до конца строки.

```
// Это однострочный комментарий
/* Это комментарий,
продолжающийся на другой строке*/
/* Коммен- /* Такие вложенные комментарии обычно недопустимы*/- тарий*/
/* Коммен- // А вот такое вложение возможно! -тарий*/
// Коммен- /* И так тоже можно!*/тарий
```

В редких случаях при комментировании бывают недоразумения, например:

```
x=y  //*это деление*/ z;
```

Достаточно добавить пробел, чтобы избежать двусмысленности:

```
x=y //*это деление*/ z;
```

Размещение описаний переменных внутри блока.

Язык Си требует, чтобы все описания локальных переменных внутри блока помещались перед первым исполняемым оператором. В Си++ можно (и часто это оказывается удобно) описывать переменные в любой точке блока перед их использованием.

Листинг 1. Локальные переменные внутри блока.

```
// расширение 'cpp' принято для файлов с текстами программ ип Си++
#include <stdio.h>
int main()
{
// В Си++ "модно" таким образом описывать и присваивать ничальное значение счетчику
// цикла for
for (int counterl=0; counterl<3; counter1++)
// Переменная counter1 "видна", начиная с этой строки и до конца main
// (а не только внутри блока for!)
// Ей присваивается значение 0 перед входом в цикл.
        {
        // Автоматической переменной i ПРИСВАИВАЕТСЯ значение 0 при
        // КАЖДОМ проходе тела цикла,
        int i=0;
        // а статическая переменная j Я нулем!
        static int j=0;
        for (int counter2=0; counter2<10; counter2++ )
              printf("i=%d j=%d\n", i++, j++ ) ;
        }
// counter2 "существует" вот до этоой скобки.
char quit_message [ ] = " Bye !" ;
```

```
printf ( "%s", quit_message) ;
return 0 ;
}
```

Строка из программы, приведенной в листинг1:

```
int i=0;
```

полностью эквивалентна сочетанию описания и оператора присваивания:

```
int i ;
i=0;
```

Она описывает локальную в охватывающем блоке автоматическую переменную и осуществляет присваивание этой переменной значения 0 при каждом проходе тела цикла. Выражение в правой части присваивания, разумеется, может и не быть константным. В то же время запись

```
static int j=0;
```

не эквивалентна

```
static int j; j =0 ;
```

Такая запись описывает локальную в охватывающем блоке статическую переменную и инициализирует ее нулем — иными словами, j будет иметь значение 0 при первом входе в охватывающий блок. В этом случае инициализирующее выражение обязательно должно быть константным. Однако возможны неожиданности: попробуйте, например, в программе из листинга 1 заменить

```
static int j=0;
```

на

```
static int j=j+i;       // не константное выражение в правой части!
```

Результат зависит от используемого компилятора. Borland C++ 3.0, не выдавая никаких предупреждений, инициализирует j нулем, а присваивания при каждом проходе цикла делать не будет. Так что будьте внимательны. А вот пример фрагмента программы с синтаксическими ошибками:

```
int k;
k=int i=0; // Ошибка!
print f ( " %d" , int j=i;); // Ошибка!
for (int count=0; count<max_count; count++)
{
// операторы..
}
// ОШИБКА! Повторное описание count
for (int count=0; count<max_count; count++)
{
// операторы...
}
```

Прототипы функций. Аргументы по умолчанию.

В «классическом» Си наличие прототипов функций необязательно. Такая «снисходительность» часто порождает массу трудно обнаруживаемых ошибок, поскольку компилятор не может проверить, соответствуют ли при вызове функций типы передаваемых аргументов и тип возвращаемого значения определению (definition) данной функции. Си++ более строг: он требует, чтобы в модуле, где происходит обращение к функции, причем обязательно до соответствующего обращения, присутствовало либо определение этой функции, либо ее объявление (declaration) с указанием типов передаваемых аргументов и возвращаемого значения, или, по терминологии Си и Си++, прототип. В последнем случае определение функции может находиться в другом модуле. Обычно прототипы функций помещают в заголовочный файл, который включается в компилируемый модуль директивой **#include**. Именно такая ситуация представлена в листинге 2: файлы **mod1.cpp** и **mod2.cpp** входят в один и тот же проект, а заголовочный файл **header.h** содержит прототипы функций, определяемых в **mod2.cpp**.

Листинг2. Использование прототипов функций.

```
// header, h (иногда заголовочным файлам для программ на Си++ дают расширение 'hpp')
void func1(int, double*);
// void означает, что функция не возвращает никакого значения;
// при желании можно написать и так: void func1 (int j, double* x_ptr);
// имена аргументов в прототипах необязательны, но их наличие облегчает чтение листингов.
Double func2() ;
// Внимание! В отличие от Си, где пустые скобки в описании функции означают произвольное
// количество аргументов любого типа, в Си++ они означают отсутствие передаваемых
// аргументов (void).
Void func3(int, ...);
// Первый аргумент - типа int, тип и количество остальных могут варьироваться.
// mod1.cpp
#include "header.h"
. . .
int main()
{
int i;
double x[20] , y;
func1(i, x) ;
y=func2() ;
func3(i, x, func2);
}
// mod2.cpp
// Определения функций, вызываемых в modl.cpp. Обратите внимание, что заголовки
// функций в определениях также оформляются  "по-новому".
// ПРИМЕЧАНИЕ: некоторые компиляторы Си++ допускают заголовки типа int f(i, j) int i,j;
// {тело функции), но большинство - нет.
Void func1(int i, double *x)
{
    // тело функции...
}
 double func2 ()
{
. . .
}
void func3(int i, ...)
{
. . .
}
```

В прототипах функций можно также задавать значения аргументов, передаваемые по умолчанию. Предположим, вы написали функцию DrawCircle, которая рисует на экране окружность данного радиуса с центром в данной точке, и задали ее прототип:

```
void DrawCircle(int x=100, int y=100, int radius=100);
```

или просто:

```
void DrawCircle(int=100, int=100, int=100);
```

хотя так, по-моему, труднее читается.

Тогда вызовы этой функции будут проинтерпретированы, в зависимости от количества передаваемых аргументов, следующим образом:

```
DrawCircle ();      //рисуется окружность с центром в точке (100, 100) и с радиусом 100 DrawCir-
cle(200) ;                  //центр в точке (200, 100), радиус 100
DrawCircle (200, 300);          // центр в точке (200, 300), радиус 100
DrawCircle (200, 300, 400);  // центр в точке (200, 300), радиус 400
DrawCircle (, ,400);              // Ошибка! Разрешается опускать аргументы только справа!
```

Значения по умолчанию можно задавать для всех аргументов, но начинать надо обязательно «справа». Такой прототип, например, был бы ошибочным:

```
void DrawCircle(int x, int y=200,
int radius);                                    // Ошибка!
```

А вот пример возможной двусмысленности при объявлении аргументов по умолчанию:

```
void    func (double*=0) ; // ОШИБКА! *= является оператором присваивания!
void    func (double* =0); // Вот так правильно.
```

Доступ к глобальным переменным, скрытым локальными переменными с тем же именем.

Оператор разрешения области видимости **:: (scope resolution operator** — двойное двоеточие) позволяет воспользоваться глобальной переменной в случае, если она скрыта локальной переменной с тем же именем:

```
int i=0;          // глобальная переменная i
. . .
int f()
{
int i=0;     //локальная переменная i внутри функции f
i++;              //увеличивается локальная i
i++;             //увеличивается глобальная i
. . .
}
```

Функциональная запись преобразования типа.

Вместо привычной операторной, в Си++ допустима функциональная запись явного преобразования типа *(см. листинг 3)*. В действительности функциональная запись несет в себе глубокий смысл при преобразовании типов данных, определенных пользователем, поскольку представляет собой вызов функции, осуществляющей такое преобразование заданным образом (к этому вопросу мы вернемся во второй части нашей книги). В ряде случаев использование функциональной записи вместо операторной существенно упрощает выражения, уменьшая число скобок.

Листинг 3. Функциональная запись преобразования типа.

```
Int a, b;
Typedef char* PChar;
Typedef void* PVoid;
Typedef int* Pint;
PChar c_ptr, d_ptr;
PVoid v_ptr;
// . . .
a=(int*)v_ptr;                  // операторная запись
b=PInt (v_ptr);                // функциональная запись
c_ptr= (PChar) v_ptr;    // операторная
d_ptr=PChar (v_ptr);     // функциональная
v_ptr=PVoid(PChar(v_ptr)+1);  // выглядит явно лучше,чем следующая строка!
v_ptr=(void*)((char*)v_ptr+1);
```

Модификаторы const и volatile в Си++.

Модификатор **const,** как и в обычном Си, запрещает изменение значений переменных. Разумеется, константа должва быть инициализирована при описании, ведь в дальнейшем ей ничего нельзя присваивать. Кроме того, в Си++ переменные, описанные как const, становятся недоступными в других модулях проекта, подобно статическим переменным.

Поэтому раздельная компиляция двух модулей из листинга 4 пройдет успешно, но компоновщик сообщит, что символ **pi** в модуле **mod2.cpp** не определен.

В большинстве случаев компилятор Си++ трактует описанную как **const** переменную, не локальную ни в одном блоке (то есть, областью видимости которой является весь файл), точно так же, как и поименованную константу, созданную директивой **#define,** т. е. просто подставляет в соответствующих местах величину, которой данная переменная иницяализировава. Однако **const** обладает тем преимуществом перед # **define,** что обеспечивает контроль типов, поэтому его использование сможет уберечь вас от многих ошибок.

Листинг 4. Модификатор const делает переменную недоступной в других модулях.

```
// Первый модуль.
// mod1.cpp
const float pi=3.14159;
// Второй модуль.
// mod2.cpp
#include <stdio.h>
extern float pi;
int main()
{
printf("%f", pi );
return 0;
}
```

Модификатор **volatil**e, напротив, сообщает компилятору, что значение данной переменной может быть изменено каким-либо фоновым процессом — например, при обработке прерывания. С точки зрения компилятора это означает, что вычисляя значение выражения, в которое входит такая переменная, он должен брать ее значение только из памяти (а не использовать копию, находящуюся в регистре, что допустимо в других случаях).

Применение **const** и **volatil**e распространяется в Си++ также на классы и функции-члены, о чем будет рассказано несколько позднее.

Ссылки.

В большинстве языков программирования параметры передаются либо по ссылке (**by reference**), либо по значению (**by value**). В первом случае подпрограмма работает непосредственно с переменной, переданной ей в качестве параметра, во втором же ей доступна не сама переменная, а только ее значение. Различие здесь очевидно: переменную, переданную по ссылке, подпрограмма может модифицировать, а переданную по значению — нет.

В языке Си параметры передаются только по значению; общепринятый способ обеспечить функции непосредственный доступ к какой-либо переменной из вызвавшей программы состоит в том, что вместо самой переменной в качестве параметра передается ее адрес.

При работе на Си++ нет необходимости прибегать к подобным ухищрениям — в нем, как и в Паскале, реализованы оба способа передачи параметров. В листинге 5 приведены четыре версии функции **swap**, обменивающей значения двух переменных: в **swap1** и **swap2** использована передача по значению непосредственно переменных (что, разумеется, не даст нужного эффекта) и указателей на них, в **swap3** и **swap4** — передача по ссылке.

Листинг 5. Передача параметров по значению и по ссылке.

```
#include <stdio.h>
void swapl(int x, int y)
{
printf("внутри swap1 перед обменом x=%d, y=%d\n", x, y);
int z=y;
y=x;
x=z;
printf("внутри swap1 после обмена x=%d, y=%d\n", x, y);
void swap2(int *x, int. *y)
{
printf("внутри swap2 перед обменом *x=%d, *y=%d\n", *x, *y);
int z=*y;
*y=*x;
*x=z;
printf ("внутри swap2 после обмена *x=%d, *y=%d\n", *x, *y);
void swap3(int& x, int& y)
// & x означает, что x будет передаваться по ссылке, то есть воздействие на формальные
// параметры внутри swap3 приведет к изменению значений аргументов, с которыми swap3
// вызвана, а в остальном swap3 ничем не отличается от swap1!
// ПРИМЕЧАНИЕ: type & читается как "ссылка на тип type".
{
print f ("внутри swap3 перед обменом x=%d, y=%d\n", x, y);
int z=y;
У=x;
x=z ;
printf ("внутри swap3 после обмена x=%d, y=%d\n", x, y);
}
void swap4 (double &x, double &y)
{
printf("внутри swap4 перед обменом x=%f, y=%f\n", x, y);
int z=y;
y=x;
x=z;
print f ("внутри swap4 после обмена x=%f, y=%f\n", x, y);
}
int main()
{
int a=0, b=1;
print f ("первоначально a=%d, b=%d\n", a, b);
swap1(a, b) ;
printf("после swapl a=%d, b=%d\n", a, b);
swap2(&a, &b) ;
printf ("после swap2 a=%d, b=%d\n", a, b);
swap3(a, b) ;
printf ("после swap3 a=%d, b=%d\n", a, b);
swap4(a, b) ;
printf ("после swap4 a=%d, b=%d\n", a, b);
return 0 ;
}
```

Опытный программист, конечно, сразу же догадается, что *использование ссылок* по сути дела представляет собой *неявное использование указателей,* и код, генерируемый компилятором для функций **swap2 и swap3,** скорее всего совпадет. Затруднения, однако, может вызвать функция **swap4.** Если Вы уже успели откомпилировать пример и проверить его работу, то должны были заметить, что вызов **swap4,** несмотря на использование ссылок, не влияет на значения аргументов в вызывающей программе. Кроме того, если Вы пользовались для компиляции компилятором *Borland C++ 3.0,* то он должен был выдать два предупреждения:

Warning ex2.cpp 52: Temporary used for parameter 'x' in call to 'swap4(double &, double &)' in function main()

Warning ex2.cpp 52: Temporary used for parameter 'y' in call to 'swap4(double &, double &)' in function main()

Дело в том, что параметру ссылочного типа должна быть присвоена ссылка на переменную *этого же* типа. Если типы не совпадают, то компилятор создает временную переменную требуемого типа, присваивает ей значение передаваемого параметра (конечно, после преобразования типа, если таковое возможно!), и функции передается адрес этой временной переменной. Поясним вышесказанное более подробно. Предположим, функция **f** требует ссылку на **int,** а переменная **ch** имеет тип **char.**

```
. . .
void f(int&);    // прототип     f
. . .
char ch;
f(ch) ;
. . .
```

Вместо вызова **f** с адресом **ch** в качестве параметра, компилятор будет генерировать код, соответствующий следующему выражению:

```
int Tmp=(int)ch, f(Tmp);
```

Очевидно, что вызов **f** может изменить лишь значение временной переменной **Tmp** (к которой у программиста нет доступа!); **ch** в любом случае останется неизменной.

Следует также заметить, что компилятор создает временную переменную и в случае, если в качестве параметра будет передана константа. Например, в нашем случае, вызов:

```
f(6) ;
```

будет эквивалентен:

```
int Tmp=6, f(Tmp) ;
```

Передавать аргументы по ссылке бывает полезно и для того, чтобы избежать копирования больших блоков данных при передаче в качестве параметров «объемистых» структур, даже если и не требуется, чтобы вызов функции изменял значение параметра в основной программе. В этом случае имеет смысл «застраховаться» от случайной модификации параметра, переданного по ссылке, описав его как **const:**

```
void f( const LargeStructure&; nonmodif_par) {тело функции...}
```

Теперь, если внутри функции **f** компилятор встретит выражение, изменяющее значение **nonmodif_par,** он выдаст сообщение об ошибке. Функции также могут возвращать значения ссылочного типа, что бывает особенно полезно при «конвейерных» вызовах **f1(f2(f3(object))),** когда каждая последующая функция должна обрабатывать тот же самый объект, а не его копию.

Листинг 6 содержит пример использования ссылок при передаче структур. Если ваш компилятор может транслировать программу на язык ассемблера, попробуйте сравнить ассемблерные листинги для первоначального варианта этой программы и варианта, в котором удалены все символы & и, следовательно, структуры передаются по значению. Поддерживает ли ваш компилятор передачу структур по значению? Выдает ли он при этом какие-либо предупреждения? Сравните эффективность передачи структур по ссылке и по значению. Рассматривая ассемблерные листинги, обратите внимание, каким образом компилятор при трансляции модифицирует имена функций, помещаемые затем в объектный файл (позже мы еще вернемся к этому вопросу).

Листинг 6. Передача структур по ссылке.

```
#include <stdio.h>
#include <string.h>
struct Student
{
        char FirstName[20];     // имя
        char SecondName[20]  //  в данном случае отчество
        char Surname[20];       // фамилия
        int Age;                // возраст
        char Dept[20] ;         // факультет
        int Year;               // курс
};
// Распечатать информацию о студенте. Обратите внимание, что функция возвращает ссылку
// на  Student, что позволит в дальнейшем использовать ее в левой (!) части оператора
// присваивания.
Student& PrintStudentInfo(Student& st)
{
 printf("Firstname %s\n"
        "Secondname %s\n"
        "Surname %s\n"
        "Age %d\n"
        "Department %s\n"
        "Year %d\n"
        * * * * * * * * * * * * * * * * * *\n",
        st.FirstName,
        st.SecondName,
        st.Surname,
        st.Age,
        st.Dept,
        st.Year) ;
        return st;
}
// сократить имя и отчество до инициалов
Student& ShortenNames(Student& st)
( st.SecondName[1]=st.FirstName[1]='.' ;
st.SecondName[2]=st.FirstName[2]='\x0' ;
return st;
// записать фамилию заглавными буквами
StudentSe Capitalize(Student& st)
{
strupr(st .Surname) ; // прототип в string.h
return st ;
}
int main()
{
// В Си++ не обязательно использовать ключевое слово struct при описании переменных
//  структурного типа.
Student stl={ "Ivan",
              "Ivanovitch",
```

```
                "Sidorov",
                24,
                "Chemistry",
                3
                };
Student st2={
                "Sidor",
                "Sidorovitch",
                "Ivanov",
                26,
                "Math",
                5
                }
PrintStudentInfo(st1) ;
Student st1_mod=Capitalize(ShortenNames(st1)) ;
// можно было и наоборот:
// ShortenNames(Capitalize(st1)) ;
PrintStudentInfo(st1_mod)=st2 ;      //  С точки зрения "обычного" Си это
// выражение - жуткая ересь, а для Си++ ничего необычного: PrintStudentInfo возвращает
// ссылку на st1_m,  а присвоение значения ссылке эквивалентно присвоению значения тому
// объекту, на который она ссылается (в данном случае st1_mod).
PrintStudentInfo (stl_mod) ;    // st1_mod теперь имеет то же значение, что и st2
Return 0 ;
}
```

Ссылочный тип можно использовать не только для передачи параметров, но и для создания псевдонимов (**aliases**) перемных.

```
int x=1;
int &xr=x; // ссылка xr становится псевдонимом x
xr=2 ;       // все равно, что x=2, так как xr  полностью эквивалентна x
xr++;        // то же, что и x++
```

Однако,

```
int x=l;
char &xr=x;     // Так как типы x и xr не совпадают, компилятор создает временную
// переменную типа char, для которой xr будет псевдонимом, и присваивает ей значение
// (char)x, т. е. '\x1'.
xr=2;              // значение x не изменяется!
```

При программировании на Си++ ссылки применяются весьма широко и поэтому программист должен хорошо потренироваться в их использовании, чтобы в дальнейшем чувствовать себя свободно и избегать всяческих недоразумений.

Подставляемые функции.

В обычном Си для уменьшения расходов на вызовы небольших часто вызываемых функций принято использовать макросы с параметрами. Однако, их применение сильно запутывает программу и служит неиссякаемым источником трудноуловимых ошибок, способных взбесить даже закаленного программиста на Си.

В этом легко убедиться, откомпилировав и выполнив программу, приведенную в *листинге 7*. Из-за того, что **MacrosCube(a++)** «разворачивается» в **(a++)*(a++)*(a++)**, результат действия макроса существенно отличается от результата вызова функции.

Листинг 7. Замена функции макрокомандой подчас может обернуться трудноуловимой ошибкой.

```
#include <stdio.h>
int FunctionCube(int x)
{ return x*x*x; }
#define MacrosCube(x) ((x)*(x)*(x))
int main()
{
int a=2 ;
printf(" FunctionCube(a++)=%d, ",
        FunctionCube(a++));
printf ("a=%d\n", a) ;
a=2;
printf ("MacrosCube(a++)=%d, ", MacrosCube(a++));
printf("a=%d\n", a);
return 0;
}
```

А что же предлагает взамен Си++? Достаточно описать функцию как **inline** — и компилятор, если это возможно (сравните с описанием переменной как **register**!), будет подставлять в соответствующих местах тело функции, вместо того, чтобы осуществлять ее вызов. Конечно, определение подставляемой функции должно находиться перед ее первым вызовом:

```
. . .
inline int InlineFunctionCube(int x)
{
return x*x*x;
}
. . .
b=InlineFunctionCube(a) ;
c=InlineFunctionCube(a++) ;
. . .
```

Вот теперь можно повысить эффективность программы, пользуясь при этом всеми преимуществами контроля типов и не опасаясь побочных эффектов! Позже, при рассмотрении функций-членов классов мы покажем, что не всегда необходимо использовать ключевое слово inline, чтобы сделать функцию подставляемой. Невозможна подстановка функций, содержащих операторы **if, case, for, while, goto**. Если для данной функции, описанной как **inline**, компилятор не может осуществить подстановку, то он трактует такую функцию как статическую (**static**), выдавая, как правило, соответствующее предупреждение.

Изучите опции используемого вами компилятора, связанные с обработкой подставляемых функций. Заметим, что бывает полезно указать компилятору, чтобы он все подставляемые функции оформил как обычные, то есть осуществлял не подстановку, а вызов, если вы компилируете программу для последующей отладки символьным отладчиком.

Операторы динамического распределения памяти.

Так как занятие и освобождение блоков памяти является очень распространенной операцией, в Си++ введены два «интеллектуальных» оператора **new** и **delete**, освобождающих программиста от необходимости явно использовать библиотечные функции **malloc, calloc и free**. Применять эти операторы очень легко (*см. листинг 8*).

Листинг 8. Использование операторов new и delete.

```
#include <stdio.h>
int main()
{
int *i_ptr;
double *d_ptr;
char *string;
int str_len=80;
i_ptr=new int;  // Зарезервировать место под переменную типа int и присвоить i_ptr ее
// адрес.  Блок памяти будет занят до соответствующего оператора delete или, при его
// отсутствии, до  конца программы.
// Значение *i_ptr не определено!
d_ptr=new double (3.1415); // Аналогично предыдущему, только *d_ptr //инициализируется
значением 3.1415
string=new char [str_len] ; // Отвести место в свободной памяти под массив из
// str_len элементов типа char. string теперь содержит адрес нулевого элемента массива. Если
// выделить память невозможно, то оператор new возвращает значение NULL, т. е. (void*)0
if (!(i_ptr && d_ptr && string))
{
printf("Не хватает памяти для всех динамически "
            "размещаемых переменных!");
return 1;
}
string[0]='H'; string[1]='i';
string[2]='!'; string[3]='\x0' ;
printf("i_ptr=%p случайное значение *i_ptr=%d\n", i_ptr, *i_ptr"):
delete i_ptr;        // освободить блок памяти, на который указывает i_ptr. Примечание:
// если указатель, на который действует оператор delete, не содержит адрес блока,
// зарезервированного ранее оператором new, или же не равен NULL, то последствия будут
// непредсказуемыми!!!
printf("d_ptr=%p *d_ptr=%f\n", d_ptr, *d_ptr);
delete d_ptr;
print f("string=%p string contents=%s\n", string, string);
pelete[str_len] string;
// можно и так: delete string; компилятор поймет...
return 0 ;
}
```

Оператор **new** выделяет блок памяти, необходимый для размещения переменной или массива (необходимо указывать тип и, в случае массива, размерность), и при этом может присвоить вновь созданной переменной начальное значение. Оператор **delete** освобождает ранее выделенную память. Размер занятого блока, необходимый для того, чтобы обеспечить правильную работу **delete**, записывается в его начало и обычно занимает дополнительно четыре байта. Следует также помнить, что реальный размер занятого блока не произволен, а кратен определенному числу байт (например, в реализации Borland C++ 3.0 — 16), поэтому с точки зрения расхода памяти невыгодно резервировать много блоков под небольшие объекты.

В случае успешного выполнения **new** возвращает адрес начала занятого блока памяти, увеличенный на количество байт, занимаемых информацией о размере блока (то есть адрес созданной переменной или адрес нулевого элемента созданного массива). Как видно из листинга, значение, возвращаемое оператором **new**, не требует явного приведения к типу указателя, которому присваивается соответствующий адрес. В ситуации когда **new** не может выделить требуемую память,

он возвращает значение **(void*)0**, на что программа, разумеется, должна правильно отреагировать.

Помимо проверки возвращаемого значения и последующих действий, Си++ предоставляет еще один способ обработки ситуации нехватки свободной памяти, а именно, программист может определить специальную функцию обработки, которая будет вызываться при неудачном выполнении оператора **new** и пытаться, как сейчас выражаются, «изыскать» необходимую память, освободив ранее занятые блоки, либо выдавать соответствующее сообщение и завершать выполнение программы путем вызова библиотечной функции **exit** или **abort**. Следует иметь в виду, что нормальный выход из этой функции в случае неудачи невозможен: после возврата оператор **new** еще раз пытается зарезервировать требуемую память, и, если ее по-преж-невду не хватает, программа зациклится.

Чтобы установить определенную пользователем функцию обработки, необходимо присвоить ее адрес указателю **_new_handler**, объявленному в стандартном заголовочном файле new.h как

```
typedef void (*pvf) () ; // указатель на функцию без аргументов,
                         // не возвращающую значения
extern pvf _new_handler;
```

Другой способ состоит в вызове библиотечной функции **set_new_handler**, прототип которой также находится в **new.h**:

```
pvf set_new_handler(pvf);
```

Аргумент функции **set_new_handler** — адрес устанавливаемой функции-обработчика, возвращаемое значение — адрес старого обработчика.

Как видите, все это очень просто. В дальнейшем вы убедитесь, что операторы **new** и **delete** значительно «мощнее», чем кажется на первый взгляд. Особенно это проявляется при их использовании для динамического размещения и удаления объектов, принадлежащих к типу данных, определенному пользователем. Остается добавить, что в программе, использующей **new** и **delete**, не запрещается применять также **malloc**, **free** и т. п.

Перегрузка функций.

Предположим, вам по ходу программы часто необходимо печатать значения **типа int, double** и **char***. Почему бы не создать для этого специальные функции (*см. листинг 9*}?

Листинг 9. На Си каждой из функций для вывода переменных разного типа придется дать особое имя.

```
void print_int(int i)
{printf("%d", i);}
void print_double(double x)
{printf("%f" , x);}
void print_string(char* s)
{printf("%s", s);}
    . . .
int j=5;
print_int(j) ;
print_double(3.141592) ;
print_string("Hi, there'");
    . . .
```

В стандартном Си потребовалось дать этим трем функциям различные имена, а вот в Си++ можно написать «умную» функцию **print**, существующую как бы в трех ипостасях (*см. листинг 10*):

Листинг 10. Перегрузка функций.

```
#include <stdio.h>
void print(int i)
{printf("%d ", i);}
void print(double x)
{print£("%f ", x);}
void print(char* s)
{printf("%s ", s);}
int main()
{
int j=5;
double e=2.7183;
float pi=3.1415926;
print(j);
print(e) ;
print(pi) ;
print("Hi, there!");
return 0 ;
}
```

Видите, как удобно! Компилятор сам выбирает, какую из трех *различных* (!) функций с именем **print** (по терминологии Си++ «перегруженных» — **overloaded**) вызвать в каждом случае. Как Вы, должно быть, уже догадались, критерием выбора служат количество и тип аргументов, с которыми функция вызывается, причем, если не удается найти точного совпадения, то компилятор выбирает ту функцию, при вызове которой «наиболее легко» выполнить для аргументов преобразование типа. Обратите внимание на два существенных обстоятельства.

❖ Перегруженные функции не могут различаться только по типу возвращаемого значения:

```
void f(int., int);
int f(int, int);  // ОШИБКА!
```

❖ Перегрузка функций не должна приводить к конфликту с аргументами, заданными по умолчанию:

```
. . .
void f ( int=0); // прототипы
void f ();
. . .
f();        // КАКУЮ ФУНКЦИЮ ВЫЗВАТЬ?
. . .
```

Исследуйте, в каком виде помещает компилятор имена трех функций **print** из листинга 10 в объектный файл. Разумеется, поскольку компилятор Си++ позволяет давать различным функциям одинаковые имена, то, помещая имена функций в объектный файл — результат компиляции, — он должен их каким-либо образом модифицировать, чтобы сделать уникальными. Очевидно, модифицированные имена должны содержать информацию о количестве и типе аргументов, так как именно по этому признаку перегруженные функции различаются между собой. Такая модификация получила название «искажение имен» (**name mangling**). В некоторых ситуациях, например, при необходимости скомпоновать программу на Си++ с объектными файлами или библиотеками, созданными «обычным» Си-компилятором, искажение имен нежелательно. Чтобы сообщить компилятору Си++, что имена тех или иных функций не должны искажаться, их следует объявить как **extern "C"**:

```
extern "C" int funcl(int); // отдельная функция
extern "C" // несколько функций
{
```

```
void func2(int) ;
int func3 ( ) ;
double func4(double);
};
```

Модификатор **extern "C"** можно использовать не только при описании, но и при определении функций:

```
extern "C" int my_func(int i)
// my_func можно вызвать в модуле, написанном на Си
{
// тело функции...
}
```

Как нетрудно догадаться, функции, описанные как **extern "C"**, не могут быть перегруженными.

Шаблоны функций.

При написании программ на Си++ довольно часто приходится создавать множество почти одинаковых функций для обработки данных разных типов. Используя ключевое слово **template** («шаблон»), можно задать компилятору образец, по которому он сам сгенерирует код, необходимый для конкретных типов.

Давайте рассмотрим маленькую программу, использующую шаблон для функции **swap**, которая обменивает значения двух переменных (*см. листинг 11*). К слову сказать, эта функция, которая действительно очень удобна для иллюстрации различных приемов программирования, — настоящая «рабочая лошадь», кочующая по многочисленным руководствам по программированию на всевозможных языках вместе со своей верной подругой — функцией **max**.

Листинг 11. Шаблоны функций.

```
#include <string.h>
#include <stdio.h>
// Компилятор создаст подходящую функцию когда "узнает" какой тип аргументов T
// подходит в конкретном случае. Не пугайтесь нового ключевого слова "class": T может
// быть и именем простого типа данных.
template <class T> void swap(T &a, T &b)
{
T c;     // создать переменную для временного хранения значения
c=b; b=a; a=c;      //обменять
};                  //здесь символ ";" не обязателен
int main()
{
int i=0, j=l;
double x=0.0, y=1.0;
char *sl="Hi, I am the first string!",
*s2="Hi, I am the second string!";
printf ("Перед обменом: \n"
          "i=%d  j=%d\n"
          "x=%f  y=%£\n"
          "sl=%s s2=%s\n",
          i, j, x, y, si, s2) ;
swap(i,j);
swap(x,y);
swap(s1,s2) ;
printf ("После обмена:\n"
          "i=%d j=%d\n"
```

```
            "x=%f y=%f\n"
            "sl=%s s2=%s\n",
            i, j, x, y, si, s2) ;
    return 0;
    }
```

Аргументы, помещаемые в угловые скобки после ключевого слова **template**, иногда называют параметрами настройки шаблона. Заметим, что параметры настройки шаблонов функций, в отличие от параметров настройки шаблонов классов, речь о которых пойдет во второй части книги, обязательно должны быть именами типов.

Попробуйте написать для функции **swap** макрос с параметрами, выполняющий те же задачи, что и вышеприведенный шаблон. Оцените выгоды применения шаблонов. Изучите опции используемого вами компилятора, связанные с обработкой шаблонов.

В данном параграфе сведения о шаблонах функций даны скорее для ознакомления. По мере изучения объектно-ориентированного программирования мы вернемся к использованию шаблонов для решения менее тривиальных задач, чем описано выше. Следует также помнить, что не все компиляторы Си++ поддерживают шаблоны.

Перегрузка операторов.

Если в Си++ можно самому определять новые типы данных, например, структуры, то почему бы не заставить привычные операторы выполнять те действия над определенными нами типами, которые мы хотим? Пожалуйста! Пусгь **@** есть некоторый оператор языка Си++, кроме следующих:

. .* : : ? :

Тогда достаточно определить функцию с именем **operator@** и Требуемым числом и типами аргументов так, чтобы эта функция выполняла необходимые действия (*см. листинг 12*).

Листинг 12. Перегрузка операторов.

```
#include <stdio.h>
#include <string.h>
const MAX_STR_LEN=80;
struct String  // структурный тип Строка (с заглавной буквы)
{
char s [MAX_STR_LEN] ; // массив символов "содержимое" Строки
int ) str_len;         // текущая длина Строки
};
// переопределим ("перегрузим") оператор сложения для данных типа String
String operator+ (String s1, String s2)
String TmpStr; // для временного хранения...
// Длина Строки-результата сложения равна сумме длин складываемых Строк. Позаботимся
// также о том, чтобы не выйти за границу отведенного массива.
If ((TmpStr.str_len=sl.str_len+s2.str_len)
        >=MAX_STR_LEN)
{
TmpStr.s[0]='\x0' ;
TmpStr.str_len=0 ;
return TmpStr;
}
// а теперь сольем Строки
strcpy(TmpStr.s, sl.s);
strcat(TmpStr.s, s2.s);
```

```
return TmpStr; // и возвратим результат
int main()
{
String strl, str2, str3 ;  // Еще раз заметим, что при объявлении переменных
// типа struct ключевое слово struct можно не использовать. "Начиним" strl и str2
// содержимым.
strcpy(strl.s, "Перегрузка операторов - ");
strl.str_len=strlen(strl.s) ;
strcpy(str2.s, "это очень здорово!");
str2.str_len=strlen(str2.s) ;
printf ("Первая Строка: длина=%d, содержимое=%s\n",
       strl.str_len, strl.s);
printf ("Вторая Строка: длина=%d, содержимое =%s\n",
   str2.str_len, str2.s);
str3=strl+str2;       // Используем перегруженный оператор!
// Компилятор, ориентируясь на типы слагаемых, генерирует код, эквивалентный вызову
// str3=operator+(str1, str2);
printf ("Третья Строка: длина=%d, содержимое=%s\n",
   str3.str_len, str3.s):
return 0 ;
}
```

На этом описание особенностей Си++, не имеющих отношения к ООП, можно закончить. Тем самым, настало время переходить к основам объектно-ориентированного программирования. Однако сначала — заключительное задание к первой части. Возьмите какую-нибудь свою программу на Си и «переведите ее на Си++: напишите прототипы для всех функций, введите поименованные константы при помощи **const** вместо **#define**, замените макросы с параметрами на **inline** – функции, используйте **new** и **delete** для работы с динамически распределяемой памятью вместо **malloc,calloc** и **free** и т.д.

ГЛАВА 2

Основы объектно-ориентированного программирования

Язык программирования, служа для того, чтобы программист мог изложить свои идеи в форме, «понятной» компьютеру, фактически определяет не только форму изложения, но в значительной степени и сами эти идеи. Действительно, любой язык содержит в себе определенную систему терминов и понятий, в рамки которой программист, подчас сам того не замечая, «втискивает» создаваемую им программу еще на стадии проектирования. Освободиться от диктата языка невозможно, - да этого и не требуется. Желанным выходом является система понятий, приближенная к той, которой мы пользуемся при общении на естественном языке, — с ней искажения при переходе от замысла к воплощению станут минимальными, а сам этот переход будет требовать меньших усилий. В результате значительно увеличится производительность программистского труда.

Имено такой подход и предлагает объектно-ориентированное программирование (ООП), с успехом используя природные способности человеческого мышления к классификации и абстрагированию. Объекты и их свойства, наследование и иерархия — с этими понятиями человек ежесекундно сталкивается в своей повседневной жизни, они привычны и интуитивно понятны любому. Например, говоря «окно», мы подразумеваем, что сквозь него можно нечто видеть. Этим свойством обладает и окошко в доме, через которое мы можем взглянуть на улицу, и «окно» на экране монитора, которое рисует прикладная программа. А ООП фактически представляет собой проектирование классов объектов. При этом программист всякий раз становится творцом особого мира, где эти объекты возникают, разрушаются, меняют свое состояние и взаимодействуют друг с другом. И прежде всего ООП требует от нас рассматривать программу не как последовательность действий, а как особый мир, живущий своей жизнью.

Как показывает практика, самое сложное при изучении ООП опытными программистами — это отказ от старых стереотипов и переход от мышления в терминах данных и процедур к мышлению в "терминах объектов, наделенных определенными свойствами и поведением. Предположим, вы создаете графическую программу, которая может рисовать на экране различные геометрические фигуры, а затем трансформировать их различным образом и перемещать по экрану. При традиционном программировании вы будете рассматривать и проектировать свою программу как набор процедур, каждая из которых производит определенные действия: очистку экрана, рисование линии, стирание линии, рисование квадрата и т. п. Данные в такой программе играют как бы подчиненную роль и рассматриваются как «поле деятельности» для функций.

ООП предполагает совершенно иной подход. Разрабатывая программу, вы должны искать ответы на вопросы: какой набор данных характеризует состояние той или иной фигуры? как эта фигура должна вести себя в ответ на тот или иной запрос? что общего у всех фигур? как эти фигуры должны взаимодействовать с пользователем? какие запросы направлять фигурам по ходу программы?

«Три кита» на которых стоит ООП — это *инкапсуляция, наследование* и *полиморфизм.* Ознакомимся с этими понятиями подробно и посмотрим, как они реализуются в Си++.

Инкапсуляция. Классы.
Класс = данные + функции для работы с ними.

Инкапсуляцией **(encapsulation)** называется слияние данных и функций, работающих с этими данными, порождающее абстрактные типы данных» определяемые пользователем **(abstract user-defined data types). В терминологии** Си++ эти абстрактные **типы** данных называются **классами (classes).** Определение класса специфицирует члены-данные **(data members),** необходимые для представления **объектов** этого типа, и операции для работы с этими объектами, т. е. функции-члены **(member functions)** класса. Синтаксис определения класса подобен синтаксису определения структур и объединений в стандартном Си. Данные, принадлежащие объекту **(object)** некоторого класса, обусловливают *состояние* данного объекта, а набор функций-членов обусловлива-

ет *поведение* объектов класса. Если рассматривать вызов функции-члена как *запрос* к объекту совершить некоторое действие, то можно также сказать, что набор функций-членов определяет множество запросов, на которые объекты данного класса будут откликаться соответствующим образом.

```
class AnyClass
{
private:
 int x;                 // члены-данные
 double y,z;
 // ...
 void f1();                      // функции-члены
 int f2(int) ;
public:
 char ch, chl;                 // члены-данные
 int f3(int, int.);     // функции-члены
 int GetX() {return x;}
 // ...
 };
```

Метки **private: и public:** определяют *режим доступа* **(access mode)** к членам класса: первая делает члены класса, объявляемые после нее, *закрытыми,* вторая, наоборот, *открытыми.* Эти метки могут быть использованы в определении класса многократно и в произвольном порядке.

К *закрытым* **(private)** членам класса (как к функциям, так и к данным) имеют доступ только функции-члены данного класса (а также функции-друзья класса, о которых речь пойдет дальше). *Открытые,* или общедоступные **(public),** члены класса предназначены для обеспечения интерфейса объектов класса с программой, в которой они существуют. Проектируя класс, необходимо тщательно продумать, какие его члены сделать открытыми, а какие — закрытыми. Объекты спроектированного класса в идеале должны быть подобны хорошему автомобилю: водителю нужно знать, как крутить баранку, когда нажимать на тормоз, а когда — на газ, а что там под капотом — это дело изготовителя.

При определении класса с помощью ключевого слова **class** его члены будут считаться по умолчанию закрытыми. Для определения класса допустимо также использовать два других ключевых слова **— struct и union. Если** класс определен с использованием ключевого слова **struct,** то объекты класса станут по умолчанию открытыми, но их можно закрыть при помощи метки **private:.** При использовании **union** члены класса могут быть только общедоступными.

В большинстве случаев определение класса не локализовано в блоке, и областью видимости имени класса является весь файл. Если же класс определен внутри блока, то его называют локальным классом **(local class), и** на такой класс накладывается два ограничения, о которых будет сказано далее.

Определения функций-членов класса могут находиться непосредственно внутри определения класса (это имеет смысл для очень коротких функций), и в таком случае они автоматически считаются подставляемыми. Смысл этого правила Си++ состоит в том, что обычно классы содержат много очень коротких функций-членов типа GetX() **{return** X;}, **IsErrorQ {return ErrorState;}, AssignValue(int InitValue) {Value=lnitValue;}** и т. п., которые предназначаются для доступа к закрытым членам классов. Само собой напрашивается для лаконичности поместить «тощее» тело таких функций внутри определения класса и сделать эти функции подставляемыми для увеличения эффективности.

Альтернативой *для нелокальных классов* является определение функции-члена где-либо в другом месте подобно «обычной» функции. С помощью оператора разрешения области видимости **:: (scope access operator, resolution operator),** именуемого также оператором привязки, компилятору сообщается, к какому классу принадлежит данная определяемая функция. например:

```
int MyClass :: f(int i) // определяется функция fчлен класса MyClass
{
// тело функции...
```

```
}
```

Определенную таким образом функцию тоже можно сделать подставляемой, но для этого ее при определении потребуется явно описать как **inline.**

```
inline void MyClass::funci(int i, int j)
{
// тело функции..,
}
```

Такое определение необходимо поместить *перед* первым использованием этой функции.

```
class C
{ public:
//...
void f();
};
int main ()
{
C c_obj ;
c_obj . f ( ) ;  // Компилятор генерирует код для вызова функции...
//...
}
inline void C:: f()  //... и вдруг узнает, что надо было осуществить подстановку.
                                    //    ОШИБКА!
{
// тело функции
}
```

Следует подчеркнуть, что *определение класса не создает, объектов данного класса!* Объекты создаются только при описании переменных, например:

```
MyClass Obj1, Obj2, ObjArray[10] ;
```

Сделаем еще одно существенное замечание: при работе с многомодульными проектами определение класса должно присутствовать во всех модулях, где используются объекты данного класса или определяются его функции-члены. Поэтому целесообразно помещать определение класса в заголовочный файл, включаемый при помощи директивы **#include** в те модули, в которых оно необходимо. Если определение *подставляемой* функции-члена находится вне определения класса, то она обязательно должна быть определена в тех модулях, где используется (до ее вызова). Соответственно, определение такой функции также лучше поместить в заголовочный файл вместе с определением класса.

Доступ к открытым членам объекта некоторого класса осуществляется при помощи хорошо известных программистам, работающим на Си, операторов прямого и косвенного выбора компонентов структур и объединений **(direct component selector. and indirect component selector ->),** как это показано в *листинге 13*.

Листинг13. Обращение к членам объекта.

```
class MyClass
        {
        // . . .
        int i;
        int j;
public:
        int state;
        int Get_i();
        int Get_j();
        //…
//...
} ;                 // конец определения MyClass
// Конечно, такие функции стоило бы сделать подставляемыми.
```

```
Int MyClass::Get_i () {return i;}
int MyClass::Get_j () {return j;}
int main()
{
. . .
int m, n, nl ;
MyClass obj, objl;  // Создаем объекты obj и obj1 класса MyClass...
MyClass *obj_ptr=&obj ; // ...и obj_ptr - указатель на объекты типа MyClass, который
                        // инициализируем адресом объекта obj.
m=obj.i;                // ОШИБКА! i - закрытый член класса!
m=obj_ptr -> i;     // Тоже ОШИБКА!
n=obj.state;            // Так можно, но все же рекомендуется не оставлять открытых
// данных-членов, а определять соответствующие открытые функции, доступа.
m=obj_ptr-estate;       // аналогично...
. . .
m=obj .Get_i( );            // OK!
n=obj_ptr -> Get_j();  // OK!
nl=objl.Get_j();            //OK!
. . .
}
```

Здесь сразу же уместно ответить на следующий принципиальный вопрос: а каким же образом функции-члены при вызове «узнают», для какого именно объекта они вызваны и с членами-данными какого объекта класса им следует работать? В частности, в приведенном в *листинге 13* фрагменте программа значение члена **j** какого объекта должна возвращать функцш **Get_j?** Соответствующий механизм очень прост: хотя на пер вый взгляд кажется, что функция **Get_j** не имеег параметров на самом деле она при вызове получает один «скрытый» аргумент – адрес объекта, для которого она вызвана. То есть, при вызове **obj1.Get_j()** в стек загружается значение **&obj1** и выра жение:

return j ;

в теле функции на самом деле интерпретируется компилятором как:

return (&obj1)->j;

Внутри функции-члена адрес объекта, для которого она вызвана, доступен для программиста при помощи ключевого слова **this** (к нему мы еще вернемся). Отметим, что в вышеприведенном примере компилятор <знает», к какому классу принадлежат объекты, для которых вызывается та или иная функция-член, и формирует все вызовы функций-членов *уже на стадии компиляции,* что получило название «раннее связывание» **(early binding).**

В *листинге 14* содержится определение типа данных **WiseString** («УмнаяСтрока»). Объекты типа **WiseString** сумеют» присваивать себе значение и по запросу выводят информацию о себе на монитор.

Листинг 14. «Умные строки».

```
#include <stdio.h>
#include <string.h>
// Определим новый тип данных под названием WiseString, используя ключевое слово class.
class WiseString
        {
        // Данные-члены класса, необходимые для представления объектов типа WiseString:
        const char *s;   // указатель на текстовую строку

        int len;            // длина строки
        // До метки public располагаются закрытые члены класса (как функции, так и
        // данные), к которым имеют доступ только функции-члены данного класса, а
        // после - общедоступные, обеспечивающие интерфейс объектов данного
```

```
        // класса с программой.
public:
        // Функции-члены класса, которые специфицируют операции для работы с
        // объектами типа WiseString
        void Assign(const char *String)
        // Assign становится подставляемой функцией, так как она
        // определена непосредственно при описании класса
        {
        s=String;
        len=strlen(s);
        }
        void TellAboutYourself() const;  // Определение этой
        // функции будет дано далее. Модификатор const в заголовке
        // функции-члена гарантирует, что данная функция не изменит
        // значения членов-данных объекта, для которого она вызвана.
        // Конец определения WiseString
        };
int main()
{
WiseString str1, str2;   // создаем две "умных" строки
str1.Assign ("Привет, программисты!");    // А ну-ка, строки.
str2.Assign ("Да здравствует ООП!");       // наполнитесь содержимым!
str1.TellAboutYourself();                  // А теперь расскажите нам о себе!
str2.TellAboutYourseif();
return 0;
}
// Определение функции TellAboutYourself – члена класса WiseString.
void WiseString::TellAboutYourself() const
{
// Благодаря модификатору const в заголовке этой функции компилятор не позволит данной
// функции изменить значения членов-данных объекта, для которого она вызвана, например,
// оператор len++ был бы ОШИБКОЙ !
const char *fmt=" Я - умная строка! Могу рассказать Вам о себе. \n"
                        " Вот сообщение, которое я храню:\n%s\n"
                        " Длина сообщения равна %d символ(ов).\n";
printf(fmt, s, len);
}
```

Статические члены класса

Каждый создаваемый объект некоторого класса имеет свой базовый адрес, а для обращения к членам-данным какого-либо объекта компилятор использует их смещение относительно базового адреса этого объекта. Таким образом, все члены-данные с которыми мы до сих пор имели дело (назовем их нестатическими — **non-static),** принадлежащие некоторому объекту данного класса, никак не связаны с одноименными членами-данными любого другого объекта данного класса и располагаются в памяти по различным адресам.

А как же сделать, чтобы некоторые члены класса были общими для всех одновременно существующих объектов данного класса? Нет ничего проще! Достаточно описать такой член как **static.** Давайте модифицируем пример из *листинга 14* и создадим класс «ОченьУмныхСтрок», которые умеют еще и сами себя посчитать.

Листинг 15. «Очень умные строки».

```
#include <stdio.h>
#include <string.h>
class VeryWiseString
        {
        const char *s;   // указатель на текстовую строку
        int len;              // длина строки
        int my_number;   // личный номер
        static int counter;   // Каждая строка знает, сколько всего строк...
public:
        // Assign наполняет строку содержимым, присваивает ей очередной
        // номер и увеличивает счетчик строк на единицу
        void Assign(const char * String)
        {
        my_number=++counter ;
        s=String;
        len=strlen(s);
        }
        void TellAboutYourself() const;
        // Статические функции-члены НЕ получают при вызове скрытого аргумента-адреса
        // объекта и поэтому могут работать ТОЛЬКО со СТАТИЧЕСКИМИ членами-
        // данными, если адрес объекта не передать явно.
        static int HowManyStrings() {return counter;}
        } ;                                          // конец определения VeryWiseString
/*************************************************************************
Статические члены-данные и функции-члены являются глобальными и, в отличие от обычных статических переменных и функций, доступны во ВСЕХ модулях многомодульного проекта, содержащих описание данного класса.
Статические члены-данные, как и обыкновенные глобальные переменные, обязательно необходимо ОПРЕДЕЛИТЬ в одном из модулей проекта.
Обратите внимание, что при определении слово static НЕ используется.
*************************************************************************/
int VeryWiseString::counter=0;  // Определение и инициализация. Компилятор
// отводит место для VeryWiseString::counter в сегменте данных.
int main()
{
// Статические члены-данные и статические функции-члены доступны и до создания хотя
// бы одного объекта. Обратиться к ним можно при помощи оператора ::
printf ("Всего создано строк - %d\n",
        VeryWiseString::HowManyStrings());
VeryWiseString str1, str2, str3;  // создаем 3 объекта...
str1.Assign(" Привет, программисты!");
str2.Assign(" Да здравствует ООП!");
str3.Assign(" Мы умеем считать себя!");
printf (" Всего создано строк - %d\n",
        VeryWiseString::HowManyStrings()) ;
// Можно было бы вызывать статическую функцию-член и так:
// str1.HowManyStrings(); или
// str2.HowManyStrings(): или


// str3.HowManyStrings():
// все четыре вызова, естественно, совершенно эквивалентны.
```

```
str1 .TellAboutYourself();
str2.TellAboutYourselfO ;
str3.TellAboutYourself() ;
return 0 ;
}
void VeryWiseString::TellAboutYourseif() const
{
const char *fmt=
                "Я - умная строка! Могу рассказать Вам о себе. \n"
                "Мой личный номер %d\n"
                "Вот сообщение, которое я храню:\n%s\n"
                "Длина сообщения равна %d символ(ов).\n";
printf(fmt, my_number, s, len);
}
```

Конечно, статические члены не могут принадлежать локальным классам, поскольку имя такого класса невидимо за пределами блока и определение статических членов-данных локального класса становится невозможным.

Желающие более подробно узнать о свойствах и применении статических членов классов могут обратиться к моей статье «Использование статических и нестатических членов классов при программировании на Си++» («Мир ПК», №1/92, сс. 117-123).

Друзья класса.

В некоторых ситуациях желательно, чтобы функция, не являющаяся членом класса, тем не менее имела доступ к закрытым членам этого класса. Си++ позволит это сделать, если функцию объявить другом класса (может быть, лучше сказать подругой?) при помощи ключевого слова **friend**. Один класс также можно «подружить» с другим. Фрагмент программы, иллюстрирующий использование этой возможности, приводится в *листинге 16*.

Листинг 16. Дружественные функции и классы.

```
. . .
class MyClass1;  // Предварительное "tentative"объявление класса.
class MyClass
        {
        // . . .
        int j;          // закрытый член класса
        // . . .
        friend void IncJ (MyClass &);    // Примечание: метки public: и private: не
                // оказывают воздействия на доступность дружественной функции,
                // функции- члены  дружественного класса MyClass1 получат доступ к
                // закрытым членам класса MyClass.
        friend MyClass1;      // Благодаря предварительному объявлению компилятор
                // "знает"  что MyClass1 - это имя класса, который будет определен
                // позднее.
        // . . .
        };
class MyClass1
        {
        // . . .
        int j;
public:
        void MakeJEqual(MyClass &);
        // . . .
```

```
        };
// Определение функции IncJ.
// Заметьте, что все, к чему обязывает дружба класса MyClass и функции IncJ - это то, что
// класс разрешает функции пользоваться своими закрытыми членами.
void IncJ(MyClass &obj)
{
obj.j ++;
}
// MakeJEqual является членом MyClass1 и, кроме того, имеет доступ к закрытым членам
// объектов класса MyClass.
void MyClass1::MakeJEqual (MyClass &obj )
{
// Присвоить члену j объекта класса MyClass1 значение члена j объекта
// класса MyClass.
j =obj.j ;
}
// ...
int main()
{
// ...
MyClass obj;
MyClass1 obj1;
//   ...
// Функция IncJ НЕ является членом класса MyClass и поэтому НЕ может быть вызвана как obj.IncJ().
// Скрытый аргумент - адрес объекта - ей не передается. В данном случае, чтобы изменить значение
// члена j объекта obj, функция в явном виде получает ссылку на объект obj.
IncJ(obj);
// Присвоим obj1.j значение obj.j
obj1.MakeJEqual(obj);
// . . .
}
```

Перегрузка операторов для классов.

Перегрузка операторов для введенных пользователем типов уже обсуждалась в первой главе. Однако с тех пор мы узнали много нового и поэтому имеет смысл еще раз вернуться к этому вопросу.

Как Вы, уважаемый читатель, наверное, уже догадываетесь, если некоторый оператор @ языка Си++ перегружается для работы с объектами каких-либо классов, то в ряде случаев разумно и даже необходимо сделать соответствующую функцию **operator@** либо членом класса, либо дружественной функцией. Давайте рассмотрим пример программы, где (в учебных целях) реализованы оба варианта. В этой программе, приведенной в *листинге 17*, создается класс **Vector,** для которого определяются четыре операции: сложение, вычитание, уменьшение и увеличение, причем, последние два унарных оператора перегружаются как префиксные.

Здесь необходимо сказать, что в ранних стандартах Си++ нельзя было определить две различных функции **operator@** для унарного оператора **@**, допускающего как префиксное, так и постфиксное использование. Новый стандарт Си++ AT&T 2.1 предоставляет такую возможность. Функция-член некоторого класса **Class :: operator@()** будет соответствовать префиксному оператору **@**, тогда как **Class :: operator@(int) —** постфиксному. При вызове последней версии компилятор будет передавать функции некоторую целую константу, значение которой не играет роли.

Листинг 17. Перегрузка операторов для классов.

```
#include <stdio.h>
class Vector
        {
        int X, Y; // Компоненты (координаты) вектора.
        friend Vectors& operator--(Vector&);  // Префиксный оператор
        // уменьшения вычитает из вектора единичный вектор и возвращает при помощи
        // ссылки САМ вектор, а не его значение.
        friend Vector operator-(const VectorS&, const Vector&);
        // Найти разность двух векторов. Обратите внимание, что возвращается НЕ ссылка!
public:
        // Присвоить значения компонентам вектора.
        void Assign(int x, int y) {X=x; Y=y;}
        Vector operator++(); // Вектор увеличения прибавляет к вектору
        // единичный вектор и возвращает САМ вектор.
        Vector operator++(int); // Постфиксный оператор увеличения возвращает
        // ЗНАЧЕНИЕ вектора ДО увеличения.
        Vector operator+(const Vector&) const; // Найти сумму двух
        // векторов. Обратите внимание, что возвращается НЕ ссылка!
        void print() const  // Напечатать значения, компонент вектора.
        {
        printf ("Координаты вектора X=%d, Y=%d\n", X, Y) ;
        }
};
// Увеличение - унарная операция. Функция-член operator++ не имеет явно передаваемых
// аргументов, так как при вызове получает адрес объекта, для которого она вызвана.
Vectors& Vector::operator++()
{
X++;
Y++;
// this - адрес вектора, для которого вызывается функция operator++, а возвратить надо сам
// вектор ПОСЛЕ увеличения, то есть (*this). В данном случае оператор увеличения
// перегружен как ПРЕФИКСНЫЙ!
return *this;
}
Vector Vector::operator++(int)
{
// аргумент функции игнорируется
Vector vec_tmp=*this ;
X++;
Y++;
return vec_tmp;     // Постфиксный оператор ++ возвращает ЗНАЧЕНИЕ вектора ДО
                                // УВЕЛИЧЕНИЯ.
}

// Оператор сложения - бинарный, поэтому функция-член operator+ должна иметь помимо
// скрытого (this) один явно передаваемый параметр.
Vector Vector::operator+(const Vector& vec) const
{
// Создать вектор для хранения промежуточных результатов и присвоить ему значение
// ПЕРВОГО слагаемого...
Vector vec_tmp=*this;
// сложить...
```

```
vec_tmp.X+=vec.X;
vec_tmp.Y+=vec.Y;
return vec_tmp;          // и возвратить результат (но НЕ сам объект *this).
}
// Функции-друзья класса не являются членами этого класса и не получают скрытого параметра
// this, поэтому унарные операции перегружаются дружественными функциями с одним
// аргументом, а бинарные - с двумя. Передается ссылка, т. к. операция  уменьшения должна
// изменить сам операнд.
Vector& operator--(Vector&; vec) // Префиксный оператор  уменьшения.
{
vec.X--;
vec.Y--;
return vec;          // Возвращается САМ объект vec после уменьшения.
}
Vector operator-(const Vector& vecl, const Vector& vec2)
{ Vector vec_tmp=vecl;
vec_tmp.X-=vec2.X;
vec_tmp.Y-=vec2.Y;
return vec_tmp;          // Возвращается результат вычитания векторов.
}
const NUMBER_OF_VECTORS=5;
int main()
{
Vector v_array[NUMBER_OF_VECTORS], vl; // создать массив вектоов...
                                                    //...и еще один вектор.
for (int i=0; i<NUMBER_OF_VECTORS; i++).
        {
        v_array[i].Assign(i, i+1);
        printf (" Присвоили значение вектору #%d\n", i);
        v_array[i].print();
        }
for (i=0; i<NUMBER_OF_VECTORS ; i++).
        {
        --v_array[i] ;    // компилятор интерпретирует как operator- - (v_array[i])
        printf  (" Вычли из вектора #%d единичный вектор.\n", i);
        v_array[i].print() ;
        }
vl.Assign(0, 0) ;
printf ("Сумма всех векторов массива: ");
for (i=0; i<NUMBER_OF_VECTORS ; i++)
        {
        vl=v1+v_array[i] ;     // компилятор интерпретирует как
                                        // v1 .operator+(v_array[i])
        }
v1.print () ;
return 0 ;
}
```

Попробуйте «поиграть» с этим примером, используя все пере груженные операторы.

Определите постфиксный оператор --.

Обратите внимание, что не следует определять функции для перегрузки операторов + и - как возвращающие ссылку на **Vector.** В противном случае можно получить неожиданный и, очевидно, нежелательный результат, например:

```
// . . .
```

```
Vector v1, v2, v3, v4 ;
// . . .
vl=v2+v3+v4;      // Изменение значения слагаемых v2 и v3 явно не входило бы в
// наши планы! Выражение v2+v3+v4 эквивалентно:
(v2.operator+(v3)).operator+(v4) ;
// . . .
```

Тепер наконец-то, пора начать осмысленно пользоваться стандартными предопределенными (то есть не требующими описания и инициализации перед использованием) объектами потокового ввода-вывода Си++, аналогичными по своему назначению стандартным файлам ввода-вывода «обычного» Си. Вот эти объекты:

cin - стандартный ввод. по умолчанию клавиатура, аналог **stdin;**

cout - стандартный вывод, по умолчанию дисплей, аналог **stdoirt;**

cerr - стандартный вывод сообщений об ошибках, по умолчанию дисплеи, аналог **stderr;**

clog - буферизованная версия **cerr.**

Ввод-вывод с использованием его стандартных объектов может быть перенаправлен средствами операционной системы на другие устройства или файлы. Использовать эти объекты очень легко: достаточно включить в соответствующие модули программы стандартный заголовочный файл **iostream.h** — и можно пользоваться перегруженными операторами ввода и вывода Си++ « и » (которые, как должно быть хорошо известно знатокам Си, изначально были операторами побитового сдвига). Специальные операторы, называемые обычно манипуляторами **(manipulators),** часть из которых определена в **iostream.h,** а часть - в **iomanip.h,** предоставляют средства форматного ввода-вывода. В данном параграфе мы не будем подробно обсуждать классы, входящие в стандартную библиотеку классов ввода-вывода, а ограничимся примером, демонстрирующим их использование (см. *листинг 18}*. В дальнейшем мы изучим библиотеку потокового ввода-вывода более подробно.

Обратите внимание на то, как можно перегрузить оператор « для вывода в желаемом формате не только данных, принадлежащих встроенным в язык Си++ типам (таких как **int, double** и т. д.), но и принадлежащих к типу, определенному пользователем. Так как стандартный оператор вывода Си++ « , перегруженный для работы со встроенными типами данных, возвращает ссылку на определенный в файле **iostream.h** стандартный класс потокового вывода **ostream,** то определенная нами функция **operator<<** должна также возвращать ссылку на **ostream.** Для того, чтобы использовать «наш» оператор « в одной цепочке со стандартным, необходимо, чтобы определенная нами функция **operator<<** имела два аргумента: первый будет ссылкой на **ostream,** а второй должен либо относиться к определенному нами типу, либо быть ссылкой на этот тип.

Листинг 18. Использование стандартных классов и объектов потокового ввода-вывода.

```
#include <iostream.h>
#include <string.h>
struct ComputerInfo {
        char* type;
        char* processor;
        char* coprocessor;
        int RAM;
        char* operating_systern;
        };
// перегружаем оператор << для вывода данных определенного нами типа CompulerInfo
ostream& operator<< (ostream& os, const ComputerInfo& ci)
{
return os<< "Computer System Information:\n"<<
"Type :..... "<< ci.type <<endl
"CPU: ..... "<<ci.processor<<
"Co-processor:..... "<<ci.coprocessor <<endl<<
"RAM: ..... "<<ci.RAM<<" KB"<<endl<<
```

```
"Operating System:.."<<ci.operating_system<<endl;
}
int main() {
// инициализация при помоди списка значений
ComputerInfo MyComputerInfo=
        {
        "IBM PC AT",
        "Intel 80386",
        " NO ",
        2048,
        "DOS 5.0"
        };
char name[80];
int age;
// Манипулятор endl вызывает очистку буфера и переход на новую строку.
cout << "Hello, world! "<< "Привет! Как Вас зовут и сколько Вам лет? ";
cin>>name>>age ;
// Манипуляторы dec, oct и hex вызывают вывод целого числа соответственно в
// десятичном, восьмеричном и шестнадцатеричном форматах.
cout<<name<< " Вам, наверное, будет интересно узнать, как Ваш возраст \n"<<
" выражается в различных системах счисления?\n"<<
" В десятичной: "<<dec<<age<<endl<<
" В восьмеричной: "<<oct<<age<<endl<<
" В шестнадцатеричной: "<<hex<<age<<endl<<
// Обратите внимание, как используется "неперегруженный" оператор сдвига <<
" А если Ваш возраст сдвинуть влево на 2 бита, то получится "<<endl<<:
" В шестнадцатеричной системе "<< (age<<2) << "лет."<<endl<<
" В десятичной системе это будет "<<dec<< (age<<2)<< " лет! "<<endl ;
cout<< " А вот число пи с точностью до четвертого знака: "<<
    setprecision(4)<< 3.1415926 << endl;
// Используем оператор вывода, перегруженный для данных типа Computerlnfo.
cout<< " Вам, несомненно, любопытно, на каком компьютере я работаю? \n"<< MyCom-
puterInfo<< " Всего хорошего, "<<name<< ". До свидания!";
return 0 ;
}
```

Попробуйте написать функцию
istream& operator>> (istream& os, ComputerInto& ci)
для форматированного ввода информации о компьютере и проверить ее работу, «вставив» ее в пример из листингп 18.

Важное замечание: Чтобы уменьшить размер исполняемого файла *не* делайте подставляемыми функции, содержпщне операторы потокового ввода-вывода (поскольку эти операторы порождают код очень большого объема, а выполняются достаточно медленно), для того, чтобы практически свести к нулю выигрыш в быстродействии, который вы ожидаете от **inline**-подстановки! Однако, поскольку в наших примерах лаконичность текста важнее эффективности, мы часто будем игнорировать этот совет.

Еще раз заметим, что большинство компиляторов позволяет «отключить», если это необходимо, расширение подставляемых функций и превратить их в обычные.

Теперь рассмотрим перегрузку оператора присваивания. В примерах мы уже неоднократно встречались с выражениями типа **object2=object1**, где **object1** и **object2** — объекты одного класса, и до сих пор присваивание «срабатывало» надлежащим образом, то есть приводило к присваиванию значений всех членов-данных, принадлежащих **object1**, соответствующим членам-данным **object2**, или, другими словами, к побитовому копированию **object1** в **object2**. Но ведь

такое присваивание подходит далеко не всегда. Мало того, в некоторых ситуациях оно может быть даже опасно (см. *листинг 19*)!

Листинг 19. Перегрузка оператора присваивания.

```
#include <iostream.h>
#include <string.h>
// Макроопределение для условной компиляции: сделать ли оператор присваивания
// перегруженным.
#define OVERLOAD_ASSIGNMENT
class Str
        {
        char *s;        // указатель на строку
        int len;       // длина строки
public:
        void Initialize (const char*);          // функция инициализации
        void Initialize() {s=NULL; len=0;}  // еще одна - без параметров
        void ToUpper () {strupr(s);} // преобразовать все буквы в заглавные
        void Destroy () {delete s; s=NULL; len=0;} // освободить память
        void print!) (cout << s << endl;}  // напечатать
        #ifdef OVERLOAD_ASSIGNMENT
        Str& operator=(const Str&); // перегрузить оператор присваивания
#endif
};
// функция инициализации с параметром - указателем на строку.
void Str::Initialize(const char* InitStr)
{
len=strlen(InitStr);
s=new char[len+l];   // занять блок памяти под строку
strcpy(s, InitStr);  // скопировать строку
}
#ifdef OVERLOAD_ASSIGNMENT
// Перегруженный оператор присваивания берет в качестве аргумента ссылку на Str и
// возвращает также ссылку на Str, что, во-первых, позволяет избежать копирования объекта
// класса Str при вызове функции operator=, а, во-вторых, дает возможность осуществлять
// присваивание по цепочке вида str1=str2=str3 и т.д.
Str& Str::operator=(const Str& OtherStr)
{
len=OtherStr.len;
delete s; // Освободить массив, занятый под прежнее содержимое строки. Если строка
// была пустой, то s==NULL и применение delete также не приведет к ошибке.
s=new char[len+i];
strcpyfs, OtherStr. s);   // скопировать СОДЕРЖИМОЕ по ДРУГОМУ адресу.
Return *this;             // Возвратить результат.
}
#endif
int main()
{
Str strl, str2, str3;
Str1.Initialize("Hi! I am a string!!!");
// Инициализация str2 и str3 также необходима!
Str2.Initialize();
str3.Initializer;
cout<< " После инициализации: " << endl;
cout<< " Первый объект содержит: "; strl.print();
```

```
cout<< " Второй объект содержит: "; str2.print();
cout<< " Третий объект содержит: "; str3.print();
str3=str2=strl;
cout<<  " После присваивания:" << endl;
cout<< " Первый объект содержит: "; strl.print();
cout<< " Второй объект содержит: "; str2.print();
cout<<  " Третий объект содержит: "; str3.print();
str3.ToUpper() ;
cout<< " Все буквы в ТРЕТЬЕЙ     строке преобразованы в заглавные." << endl <<
          " Только ли ТРЕТЬЯ строка изменилась?"  << endl ;
cout<< "Первый объект содержит: "; strl.print();
cout<< "Второй объект содержит: "; str2.print();
cout<< "Третий объект содержит: "; str3.print();
strl.Destroy(); str2.Destroy(); str3.Destroy ();
cout<< "Для всех объектов вызвана функция Destroy. "« endl;
cout<< "Первый объект содержит: "; strl.print();
cout<< "Второй объект содержит: "; str2.print();
cout<< "Третий объект содержит: "; str3.print();
return 0 ;
}
```

Проанализируйте работу программы из *листинга 19* в исходном виде, затем закомментируйте директиву **#define OVERLOAD_ASSIGNMENT** и еще раз откомпилируйте и запустите программу. Сравните результат.

Инициализация и разрушение. Конструкторы и деструкторы.

Не возникла ли у вас при рассмотрении программы из *листинга 19* мысль о том, что было бы здорово, если бы функции-члены класса **Str, Initialize и Destroy,** вызывались бы автоматически при создании и разрушении объектов этого класса, поскольку любой объект класса **Str** необходимо перед использованием инициализировать, а когда в нем отпадет необходимость, имеет смысл освободить память, занятую под строки, хранимые в объектах?

В Си++ автоматически вызываемые при создании и разрушении объектов специальные функции-члены классов, выполняющие задачи по инициализации и уничтожению объектов, называются, соответственно, конструкторами **(constructors)** и деструкторами **(destructors).** Надо сказать, что вы уже имели дело с конструкторами и деструкторами, сами того не подозревая: ведь если класс не содержит явно определенных конструктора и деструктора, то компилятор генерирует и вызывает их сам, когда это необходимо.

Снабдить класс «самодельными» конструкторами, которых может быть несколько – каждый со своим, отличным от других, списком параметров (они могут быть заданы «по умолчанию»), - и деструктором, который всегда один и не имеет аргументов, - совсем не сложно. Конструктор формально отличается от обычных функций-членов тем, что его имя совпадает с именем его класса. Имя деструктора образуется путем добавления символа ~ (тильда) в начало имени класса. Тип возвращаемого значения для конструкторов и деструкторов *не* указывается. Конструктор вызывается *только* при создании объекта соответствующего типа, деструктор автоматически вызы вается при разрушении объектов, но при необходимости его можно вызвать подобно обыкновенной функции-члену. Если ни один из конструкторов класса не является *открытой* функцией-членом, то объекты такого класса не могут быть созданы. Несколько забегая вперед, скажем, что такие классы не бесполезны: они могут быть базовыми классами для других классов.

Заметим, что объект класса, содержащего только открытые члены и не имеющего конструктора (и лишь в этом случае!), может быть инициализирован подобно обычной структурной переменной при помощи списка значений.

Довольно «объемистый» пример, приведенный в *листинге 20* поможет вам разобраться в материале, связанном с инициализацией и разрушением объектов класса, имеющего конструкторы и деструктор. Материал весьма важный, поэтому будьте особенно внимательны!

Примечание: так как выдача программы **ex14** не умещается на экране дисплея, рекомендуется использовать фильтр **more** для ее разбиения на страницы:

ex14 | more

Листинг 20. Конструкторы и деструкторы.

```
#include <stream.h>
#include <stdlib.h>
// ***************************************************************************
// Макроопределения для условной компиляции, предназначенные для экспериментов с
// примером.
#define DEFINE_DEFAULT_CONSTRUCTOR // Определять ли конструктор,
                                                    // используемый по умолчанию.
#define DEFAULT_CONSTR_WITH_NO_ARG // Конструктор, используемый по
// умолчанию, - без аргументов, иначе - с одним аргументом, заданным по умолчанию.
#define DEFINE_COPY_GENERATOR  // Определять ли конструктор - генератор копий.
#define UNEXPECTED_EXIT   // Неожиданный выход из программы.
#define UNEXPECTED_RETURN // Неожиданный возврат из функции main.
// ***************************************************************************
typedef char Boolean;
const Boolean NO=0;
const Boolean YES=1;
class SimpleClass
        {
        static int ObjCounter; // счетчик созданных объектов
        int PrivateNumber; // "личный номер" объекта
        Boolean IsCopy;   // объект - копия другого?
        friend void WhoAreYou(const SimpleClass&);
                                                    // напечатать сведения о себе
        // Конечно, конструкторы и деструктор должны быть ОТКРЫТЫМИ членами
        // класса! Иначе к ним не будет доступа "извне" объектов
public:
#ifdef DEFINE_DEFAULT_CONSTRUCTOR
#ifdef DEFAULT_CONSTR_WITH_NO_ARG
      // Только один из этих двух конструкторов может быть определен во избежание
      // конфликта при вызове без параметров.
      SimpleClass();  // конструктор без аргументов.
      #else
         SimpleClass(const char* DefArg="Аргумент по умолчанию!");
      #endif
#endif
      SimpleClass(int);
      SimpleClass(char);
#ifdef DEFINE_COPY_GENERATOR
      SimpleClass (const Simple-Classic);  // конструктор - генератор копий
#endif
      ~SimpleClass () ;   // деструктор
);
void WhoAreYou(const SimpleClassSc obj)
{
char *ms;
if (obj.IsCopy)
```

```
        ms="Вас слышу! Я – копия объекта #";
else
ms="Вас слышу! Я – объект #";
cout << ms << obj.PrivateNumber << endl ;
}
#ifdef DEFINE_DEPAULT_CONSTRUCTOR
#ifdef DEFAULT_CONSTR_WITH_.NO_ARG
// После заголовка функции-конструктора и до начала ее тела может находиться список
// инициализаторов, начинающийся символом двоеточий. В данном примере необходимые
// присваивания можно было бы сделать и внутри тела конструктора, но далее мы
// встретимся с ситуациями, когда без списка инициализаторов не обойтись.
// IsCopy получит значение NO
SimpleClass::SimpleClass(): IsCopy(NO) (
cout << " Привет! Я - вновь созданный объект # "<<
          (PrivateNumber=++ObjCounter) << endl <<
          " Конструктор для меня был вызван без аргументов!" << endl;
}
       #else
SimpleClass::SimpleClass(const char* Arg): IsCopy(NO)
{
cout<< "Привет! Я - вновь созданный объект # " <<
         (PrivateNumber=++ObjCounter) << endl <<
         " Конструктор для меня был вызван с аргументом типа char* !" << endl <<
         " Эта строка содержит: " << Arg << endl;
}
       #endif
#endif
SimpleClass::SimpleClass(int Arg): IsCopy(NO)
{
cout<< "Привет! Я - вновь созданный объект # " <<
         (PrivateNumber=++ObjCounter) << endl <<
         " Конструктор для меня был вызван с аргументом типа int !"
         " Arg = " << Arg << endl;
}
SimpleClass::SimpleClass(char Arg): IsCopy(NO)
{
cout<< "Привет! Я - вновь созданный объект # " <<
         (PrivateNumber=++ObjCounter) << endl <<
         " Конструктор для меня был вызван с аргументом "
         " типа char! Arg ="  << Arg << endl;
}
#ifdef DEFINE_COPY_GENERATOR
SimpleClass::SimpleClass(const SimpleClass&
OtherObject):
IsCopy (YES),     // Элементы списка инициализаторов разделяются запятыми.
PrivateNumber(OtherObject.PrivateNumber) // Копия будет иметь тот же
                                                                  // "личный номер", что и оригинал.
{
cout << " Привет! Я - вновь созданная копия объекта #"
        << PrivateNumber << endl ;
}
#endif
SimpleClass::~SimpleClass()
{
```

```
char* ms ;
if (IsCopy)
   ms= " Я - копия объекта # ";
else
      ms= " Я - объект # ";
cout << ms << PrivateNumber << "Разрушаюсь …Прощайте! " << endl ;
}
int SimpleClass::ObjCounter=0; // инициализация статического члена
SimpleClass global_object;  // Создать глобальный объект. Конструктор для этого
// объекта будет вызван ДО функции main, а деструктор - ПОСЛЕ выхода из main.
int main()
{
cout << ">>>Вызвана функция main. Создаем три объекта...\n";
SimpleClass obj1, // Вызывается конструктор "по умолчанию" (default), т. е. либо
// конструктор без аргументов, либо со всеми аргументами, имеющими значения, заданные
// "по умолчанию".
            obj2(1992), // Вызывается конструктор с одним аргументом типа int.
            //можно и так: obj2=1992,
            obj3('A');
            // можно и так: obj3='A':
cout << ">>>Создаем копию объекта obj1...\n";
SimpleClass objla=objl;
// можно и так: SimpleClass obj1a(obj1);

cout << ">>>А теперь создаем объект без имени и сразу же его используем! \n";
WhoAreYou(SimpleClass('B'));
cout << ">>> Этот временный объект должен быть уже уничтожен, не правда ли? \n":
cout << ">>> Будет ли создан еще один объект без имени - копия obj1? \n";
WhoAreYou(SimpieClass(obj1)) ;
cout << ">>> Создаем автоматический массив из двух объектов ... \n":
SimpleClass obj_array1[2];
cout << ">>> Создаем динамический массив из двух объектов оператором new...\n";
// Для инициализации каждого элемента массива вызывается конструктор "по умолчанию".
SimpleClass* obj_array2=new SimpleClass[2];
cout << ">>>0ткрывается фигурная скобка. ..\n";
      {              // начало блока
      cout << ">>> Создаем автоматический массив из двух " "объектов...\n";
      SimpleClass obj_array3 [2];
      cout << ">>> Создаем динамический массив из двух объектов оператором new...\n";
      SimpleClass* obj_array4=new SimpleClass[2];
      cout << ">>> Создаем статический объект. ..\n";
      static SimpleClass stat_obj;
      cout << ">>>3акрывается фигурная скобка... \n";
      }              //конец блока
#ifdef UNEXPECTED_EXIT
      cout << ">>> Неожиданное завершение программы функцией exit \n";
      exit(O) ;
#endif

#ifdef UNEXPECTED_RETURN
      cout << ">>>Неожиданный выход из main оператором return \n";
      return 0;
#endif cout << ">>>Работает оператор delete...\n";
// ВНИМАНИЕ! Опасное место!
Delete[2] obj_array2;
```

```
// delete[2] obj_array4; НЕЛЬЗЯ! obj_array4 уже за пределами области видимости!
Cout << ">>> Заканчивается функция main...\n";
return 0 ;
}
```

Важное замечание: Если предполагается, что оператор **delete** должен уничтожить массив объектов, имеющих явно определенный деструктор, то необходимо после ключевого слова **delete** поставить угловые скобки [ ], в которых можно (не обязательно) указать размерность уничтожаемого массива. Угловые скобки дают понять компилятору, что уничтожается не один объект, а массив объектов, и для каждого объекта массива следует вызвать деструктор. В противном случае результат действия **delete** будет непредсказуем.

Откомпилируйте и запустите пример. Проанализируйте видимость и время жизни объектов. Обратите внимание на следующие детали: конструкторы вызываются в порядке создания объектов, деструкторы – в обратном порядке. Если объект или массив объектов был создан оператором **new,** то уничтожить его можно только оператором **delete** (соответственно, при этом деструктор вызывается для этого объекта или для каждого объекта массива). Если указатель на объект или массив, созданный оператором **new,** утерян (например, адрес хранился в автоматической переменной, вышедшей за пределы видимости), а объект или массив объектов не были уничтожены при помощи **delete,** то они, став недоступными, «повиснут в воздухе», занимая память и не выполнив действия, предусмотренные деструктором. Такое положение может привести к различным неприятностям: например, если объект рисовал что-либо на дисплее, то на экране не будет восстановлено прежнее изображение, а уж если конструктор объекта переустанавливал какой-либо вектор прерывания, вас наверняка ждут совсем не праздничные сюрпризы!

Уберите (закомментируйте) директиву **#define DEFINE_DEFAULT_CONSTRUCTOR** и попробуйте откомпилировать пример. Проанализируйте сообщения об ошибках компиляции. Конструктор без аргументов (или со всеми аргументами, имеющими значения по умолчанию) называется конструктором по умолчанию **(default constructor)** и играет особую роль. Во-первых, он используется для инициализации объектов, когда не заданы параметры инициализации, а, во-вторых, этот конструктор инициализирует каждый элемент создаваемого массива объектов. Следует отметить, что не все компиляторы Си++ позволяют использовать конструкторы с параметрами, заданными по умолчанию, при инициализации массивов! Если компилятор в вышеперечисленных случаях не может найти в классе конструктора по умолчанию, он генерирует сообщения об ошибках. Позвольте, можете спросить Вы, а ведь в примере из *листинга 17* мы уже создавали массив объектов класса **Vector,** для которого не было определено вообще ни одного конструктора – и ничего, компиляция прошла? Правильно: если в классе нет ни одного явно определенного конструктора (и только в этом случае!) компилятор сам генерирует конструктор по умолчанию.

Восстановите **#define DEFINE_DEFAULT CONSTRUCTOR** и уберите **#define DEFAULT_CONSTR_WITH_NO_ARG**. Теперь конструктор по умолчанию будет иметь аргумент, заданный по умолчанию. Пронаблюдайте работу модифицированной программы (если ваш компилятор ее «проглотит»).

Поэкспериментируйте с директивами:

#define UNEXPECTED_EXIT

и

#define UNEXPECTED_RETURN

Помните, что так как компилятор «ничего не знает» о предназначении функции **exit** или **abort,** то при «неожиданном» завершении программы путем вызова этих функций деструкторы не вызываются, и это, как уже указывалось выше, может во многих случаях привести к различным неприятным последствиям. Компилятор вызывает деструктор для объекта только тогда, когда он может отследить прекращение существования этого объекта, как это, например, произойдет при возврате из **main** при помощи оператора **return.**

Изучите, в каких случаях вызывается конструктор-генератор копий. Уберите директиву **#define DEFINE_COPY_GENE-RATOR** и пронаблюдайте, как работает генератор копий, создаваемый компилятором.

Теперь необходимо подробнее остановиться на инициализации и разрушении объектов, размещаемых в свободной памяти, и на способах, при помощи которых программист сможет взять на себя управление свободной памятью.

Когда с помощью операции **new** создается объект некоторого класса, то для получения необходимой памяти конструктор будет (неявно) использовать стандартную глобальную библиотечную функцию **:: operator new.** Старый подход, позволяющий управлять размещением объектов в свободной памяти, *запрощенный* многими современными компиляторами, заключался в том, что конструктор может осуществить свое собственное резервирование памяти посредством присваивания указателю **this** до каких-либо использовании любого нестатического члена его класса. С помощью присваивания **this** значения ноль деструктор может избежать стандартной операции дерезервирования памяти для объекта его класса. Например:

```
class C
        {
        int array [10] ;
public:
        C () {this=my_allocator (sizeof(C));}
        ~C() { my_deallocator (this); this=0;}
        };
```

На входе в конструктор **this** имеет ненулевое значение, если резервирование памяти уже имело место (как это имеет место для объектов с классом памяти **auto** или **static** и объектов, которые сами являются членами классов), и ноль в остальных случаях. Естественно это надо проверить, прежде чем пытаться выделять память, если известно, что объект может размещаться не только в свободной, но и в заранее зарезервированной памяти.

Гораздо более гибкий и безопасный способ управления резервированием памяти, чем присваивание **this,** состоит в перегрузке операторов **new и delete.** Определенный пользователем' оператор **new** должен возвращать значение типа **void*** и принимать один или более аргументов, первый из которых всегда типа **size_t** (тип **size_t** определен в заголовочных файлах **stdlib.h, stdio.h** и др., обычно это **unsigned int).** Если функция **operator new** является функцией-членом некоторого класса, то это обязательно *статическая* функция-член (то есть, она имеет доступ только к статическим членам-данным своего класса), независимо от того, объявлена она **static,** или нет. Причина этого очевидна: функция **operator new** резервирует память под объект, а сам объект создается только после возврата из нее. Определенный пользователем оператор **delete** не должен возвращать значения. Первый аргумент **delete** типа **void*,** а второй (необязятельный) — типа **size_t.**

А зачем, спросите Вы, вообще брать на себя управление свободной памятью? В ряде случаев это позволяет значительно ускорить выделение памяти и уменьшить ее требуемый объем при размещении объектов некоторых классов, особенно если размещаемые объекты имеют небольшой и заранее известный размер, например, являются элементами связанного списка. Возможны и менее тривиальные ситуации. Одну из них иллюстрирует *листинг 21*, где приводятся две простейшие реализации стека для работы с данными типа **int.**

Листинг 21. Управление выделением свободной памяти.

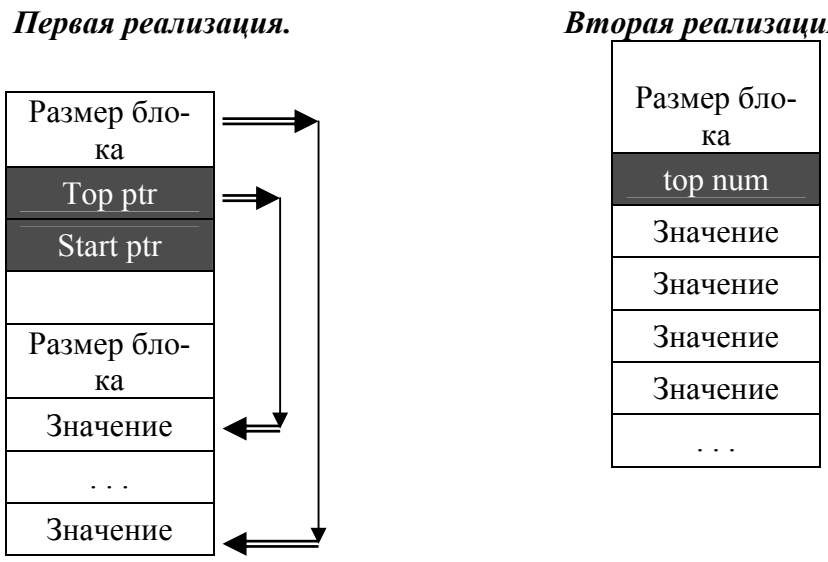
```
#include <stream.h>
#include <stdio.h>          // определение типа size_t
#define FIRST_VERSION    // макроопределение для условной компиляции
7const DEF_SIZE=10;          // вместимость стека по умолчанию
#ifdef FIRST_VERSION
// Первая (традиционная) реализация.
class Stack
{
int* top_ptr;       // указатель на вершину стека
int* start_ptr;      // указатель на начало стека
public:
// Конструктор резервирует память под sz элементов типа int.
Stack(int sz=DEF_SIZE)
```

```
{
top_ptr=start_ptr=new int[sz];
}
void push(int i)
{
*top_ptr++ = i;         // "Затолкнуть" в стек.
}
int pop()
{
return *--top_ptr;    // Извлечь из стека.
}
~Stack() {delete start_ptr;}
};
#else
// Вторая реализация.
class Stack
{
// порядковый номер значения, помещаемого в вершину стека
int top_num;
public:
Stack () {top_num=l;} // вначале стек пуст...
void push(int i) {*((int*)this+top_num++) = i;}
int pop() {return *((int*)this+ --top_num);}
/* Определенный пользователем оператор new при размещении объектов класса Stack резервирует блок памяти размером def_sz+sz*sizeof(int) байт, где def_sz - неявно формируемый компилятором первый параметр функции, operator new - размер объекта типа Stack (который, как "думает" компилятор, равен sizeof(int)), а sz - количество значений типа int, которое можно будет поместить в стек. Обратите внимание, что оператор new, используемый внутри тела функции operator new, вызывает стандартную функцию ::operator new. функция-член operator new вызывается только тогда, когда операндом new является класс, к которому она принадлежит.*/

void* operator new(size_t def_sz, int sz=DEP_SIZE)
  {return new char[def_sz+sz*sizeof(int)];}

// Деструктор не нужен, так как каждый объект класса Stack теперь занимает один блок в свободной памяти, и этот блок может быть освобожден оператором delete stk_ptr!
};
#endif
int main()
{
int st_size;
cout<<"Сколько значений типа Int вы хотите хранить в стеке? ";
cin>>st_size;
#ifdef FIRST_VERSION
cout<<"Первая реализация. \n";
Stack* stk_ptr=new Stack(st_size) ;
#else
cout<<"Вторая реализация.\n";
// Первый аргумент new формируется компилятором и представляет собой размер
// создаваемого объекта (как считает компилятор, хотя в данном случае мы его провели).
// st_size - это второй аргумент.
Stack* stk_ptr=new(st_size) Stack;
#endif
cout<<"Заталкиваем в стек...\n";
```

```
for (int counter=l; counter<=st_size; counter++)
{
cout<<counter<<' ';
stk_ptr->push(counter) ;
}
cout<<"\n Извлекаем из стека...\n";
for (counter=l; counter<=st_size; counter++)
cout<<stk_ptr->pop()<<' ';
delete stk_ptr;
return 0;
}
```

Рисунок 1 показывает размещение в свободной памяти объектов класса **Stack** для первой и второй реализации. Черным цветом выделен сам объект, как он «виден» компилятору.

Рис. 1. **Размещение в свободной памяти объектов класса Stack.**



В чем преимущества и недостатки второй реализации класса **Stack,** использующей перегруженный оператор **new,** по сравнению с первой? Самое главное преимущество — это меньшее время помещения в стек и извлечения значений **из** стека. Действительно, при первой реализации доступ к значению в вершине стека при помощи функций **push и pop** осуществляется по схеме:
(stk_ptr->top_ptr) ->значение,
тогда как при второй реализации схема доступа содержит вместо двух всего одну косвенную адресацию:
((int*)stk_ptr+top_num)->значение

Кроме того, объекты класса **Stack** при второй реализации занимают меньше места в памяти. Одним словом, вторая реализация более эффективна. Однако, за эффективность, как обычно, приходится расплачиваться универсальностью. Объекты класса **Stack** при первой реализации могут быть размещены как в свободной памяти, так и в сегменте данных или в программном стеке (конечно, сами значения, помещаемые в объекты класса **Stack,** будут все равно находиться в свободной памяти!); можно также создать массив объектов, например:

```
. . .
Stack global_stk (st_size) ;
int main()
{
. . .
Stack local_stk(st_size);
Stack stk_array[100] ;
```

```
Stack* dynair_ic_stk_array=new Stack[100];
. . .
}
```

При второй реализации объекты класса **Stack** можно разместить только в свободной памяти, причем, даже в свободной памяти нельзя создать массив объектов, так как реальный размер объектов неизвестен компилятору. Заметим, однако, что все же можно создать массив *указателей* на объекты класса **Stack:**

```
. . .
Stack *stk_ptr_array[100];
for (int count=0; count<=100; count++)
  stk_ptr_array[count]=new(st_size) Stack;
. . .
```

Как вы теперь, наверное, убедились, контроль над динамическим размещением объектов – дело тонкое, и использовать его стоит лишь в особо необходимых случаях и с большой осторожностью.

Наверняка вы сможете ответить на вопрос, как сделать, чтобы при второй реализации объекты класса **Stack** осуществляли самоконтроль и не допускали своего размещения не в свободной памяти. Для этого следует добавить в класс **Stack** статический член – флаг, который будет проверяться и устанавливаться конструктором. Оставляем это вам в качестве упражнения.

Шаблоны классов.

Шаблоны классов называют также «генераторами классов», или «родовыми классами», или «обобщенными классами», или параметризованными типами **(class generators, or generic classes, or parameterized types).** Они позволяют определить структуру семейства классов, по которой компилятор создаст классы в дальнейшем, основываясь, как и в рассмотренном ранее случае шаблонов функций, на задаваемых параметрах настройки. Наиболее показательный пример в данном случае — это создание семейства «вмещающих» **(container)** классов, например, векторов для размещения объектов произвольных типов.

Листинг 22. Шаблоны классов

```
#include <iostream.h>
template <class T, int size> // T и size – параметры настройки: T - тип
                             // размещаемых элементов,  size - их максимальное
количество
class Vector
        {
        T *elements;  // указатель на массив элементов типа T
public:
        Vector() ;
        ~Vector() {delete elements;}
        T& operator[](int i) {return elements[i];}
        void print_contents() ;
        };
// обратите внимание на синтаксис определения функций-членов шаблонов классов
template <class T, int size>
Vector<T, size>::Vector()
        {
        elements= new T[size];
        for (int i=0; i<size; elements[i]=(T)0, i++);
        };
template<class T, int size>
void Vector<T, size>::print_contents()
{
```

```
cout<<"Всего элементов:"<<size<<"Элементы:";
for (int. i=0; i<size; i++)
        cout<<' ' <<elements [i];
cout<<'\n' ;
};
int main()
{
// Создать вектора с элементами типа int, double и char, вмещающие по 10 элементов.
// Требуемый тип элементов и размер вектора указывается в угловых скобках.
Vector <int, 10> i;
Vector <double, 10> x;
Vector <char, 10> ch;
// присвоить элементам векторов значения
for (int count=0; count<10; count++)
        {
        i[count]=count ;
        x[count]=0.1+count ;
        ch[count]='a'+count ;
        }
// распечатать содержимое векторов
i.print_contents() ;
x.print_contents() ;
ch.print_contents() ;
return 0 ;
}
```

Наследование. Иерархия классов.

Наследование **(inheritance)** — один из главных механизмов языка Си++. С его помощью можно разрабатывать очень сложные классы, продвигаясь от общего к частному, а также «наращивать» уже созданные, получая из них новые классы, немного отличающиеся от исходных.
Собираясь проектировать сложный класс, необходимо прежде всего спросить себя, какими наиболее общими свойствами должны обладать его объекты и нет ли «под рукой» похожего на него готового класса. Иными словами, следует вначале как бы набросать «крупными мазками» план вновь разрабатываемого класса, а затем переходить к постепенной детализации, создавая на основе уже построенных классов новые, которые наследуют **(inherit)** от них свойства и поведение (т. е. члены-данные и функции-члены), приобретая в то же время новые качества.

Иерархия наследования классов.

Для того, чтобы указать, по отношению к каким базовым классам (**base classes**) данный определяемый класс является производным (**derived class**), в определении класса перед списком его членов приводится список базовых классов (иногда говорят также о классах-предках — **ancestor classes** — и классах-потомках — **descendant classes**). В *листинге 23* приведен пример класса, наследующего члены двух базовых классов; этот пример иллюстрирует также очередность вызова конструкторов и деструкторов при инициализации и разрушении объектов такого класса. Для определения классов в примере использовано ключевое слово **struct**, что, как мы помним, делает все их члены открытыми — это позволит нам пока не акцентировать внимание на вопросах, связанных с режимами доступа к членам классов.

Примечание. Класс, определенный при помощи ключевого слова **union**, не может быть ни базовым, ни производным по отношению к какому бы то ни было другому классу.

При создании объектов производного класса в первую очередь вызываются конструкторы базовых классов, а затем — конструктор, определенный в производном классе. При разрушении объекта деструкторы вызываются в обратном порядке. Чтобы передать, если это требуется, аргументы конструкторам базовых классов, используется уже знакомый нам список ини-

циализаторов, помещаемый *в определении* конструктора производного класса сразу после его заголовка и символа двоеточия и продолжающийся до начала тела функции, как это показано в *листинге 23*.

Листинг 23. Наследование.

```
#include <iostream.h>
// Иерархия классов:
//
//
//
//
//
struct Name
        {
        char* Firstname;     // имя
        char* Secondname;   // в данном случае отчество
        char* Surname;        // фамилия
        Name (char* FN, char* SN, char* SurN)
                {
                cout<< "Вызван конструктор класса Name"<<endl;
                Firstname=FN; Secondname=SN; Surname=SurN;
                }
         ~Name () { cout<<"Вызван деструктор класса Name"<<endl;}
         } ;
struct Job      // сведения о работе
         {
         char* Company;     // компания
         char* Position;    // должность
         Job(char* C, char* P)
                {
                cout<< " Вызван конструктор класса Job "<< endl;
                Company=C; Position=P;
                }
         Job() {cout<< " Вызван деструктор класса Job" << endl; }
         };
struct Person: Name, Job   // Подробная информация о человеке. Person наследует
                                        // члены классов Name и Job.
         {
         int Age;       // возраст
         char* Sex;    // пол
         Person (char* IFirstname, char* ISecondname, char* ISurname,
              int IAge, char* ISex, char* Icompany, char* Iposition) ;
         ~Person()
                {
                cout<<"Вызван деструктор класса Person"<<endl;
                }
         };
              Person::Person(char* iFirstname, char* Isecondname, char*Isurname, intIiAge, char*
              ISex,
              char* ICompany, char* Iposition):
// Список инициализаторов позволяет передать параметры конструкторам базовых классов.
// В данном случае, в отличие от примера 14, без него не обойтись!
              Name(IFirstname, Isecondname, I Surname),
              Job(ICompany, IPosition)
```

```
    {
    cout << " Вызван конструктор класса Person" << endl ;
    Age=IAge;
    Sex=ISex;
    }
    ostream& operator<<(ostream& os, Person& p)
    {
    return os<<" Firstname. .... " << p.Firstname << endl <<
                "Secondname. . . . " << p.Secondname << endl <<
                "Surname. ..... . " << p.Surnam << endl <<
                "Age..... " << p.Age << endl <<
                "Sex.. ..... " << p.Sex << endl <<
                "Company. ..... " << p.Company << endl <<
                " Position. ..... " << p.Position << endl ;
    }
    int main()
    {
    Person p1("Ivan", "Ivanovitch", "Sidorov", 30, "Male",
                    "SoftSci", "Programmer");
    cout << " Employee : \n" << p1 << "OK! \n" ;
    return 0 ;
}
```

Доступ к членам базовых классов.

К открытому члену базового класса, не переопределенному в производном классе, можно обращаться точно так же, как если бы он был просто членом производного класса. А вот как получить доступ к открытому члену базового класса, если в производном классе также определен член с тем же именем? Как видно из *листинга 24*, для этой цели необходимо использовать оператор разрешения области видимости **:: .**

Листинг 24. Доступ кчленам базового класса из производного класса.

```
    #include <iostream.h>
    void f()
    (
    cout << "\n Вызвана внешняя функция f !\n";
    }
    struct Base1
    {
    int a;
    void f(int i) {a+=i;}
    int get_a() {return a;}
    void print_a()
      {cout << "Base1::a=" << a << endl;}
    Base1(): a(0) {;}
    };
    struct Base2
                {
                int a;
                void f(char c) {a+=int(c);}
                void print_a() {cout << " Base2 : :a=" << a << endl; }
                Base2(): a(0) (;}
                };
    struct Deriv: Base1, Base2
```

```
{
int a;
// Класс Deriv содержит три различных члена a:
//    - унаследованный от Basel,
//    - унаследованный от Base2,
//    - определенный в классе Deriv.
void f()
{
a++ ;
:: f () ; // Так вызывается функция-не-член класса с именем, совпадающим с именем
// функции-члена. Вывод: можно не бояться давать функциям-членам имена, совпадающие
// с именами распространенных библиотечных функций типа open, seek, write и т. п., так
// как последние останутся легко доступными внутри функций-членов при помощи опера-
// тора :: . То же самое относится к именам глобальных переменных.
}
void print_a()
    {cout << "Deriv::a =" << a << endl;}
Deriv() : a (0) {;}
};
int main()
{
Deriv obj
obj.Base1 :print_a();
obj.Base2 :print_a();
obj.print_a () ; // вызывается Deriv::print_a
obj.Base1 :a=obj.Base2::a=obj.a=1 ;
obj.Base1 :prinfc_a();
obj.Base2 :print_a();
obj.print_a() ;
// Обратите внимание, что несмотря на то, что три функции f, определенные в классах
// Basel, Base2 и Deriv, имеют различные списки аргументов, и, казалось бы, компилятор
// мог бы их различить, тем не менее, чтобы вызвать одноименные функции из базовых
// классов для объекта производного класса необходимо использовать оператор ::, т. е.
// obj.Base1::f(1).
// А вот функция с именем get_a всего одна и принадлежит базовому классу Base1,
// компилятор ее и вызовет! Естественно, функция возвратит значение члена
// obj.Basel::a
cout << " Результат вызова get_a: " << obj.get_a() << endl;
obj.Base2::f('\x1');
obj.f();
obj.Base1::print_a();
obj.Base2::print_a();
obj.print_a();
return 0;
}
```

Таким образом, для объектов производного класса доступны все открытые члены-данные и открытые функции-члены базовых классов, а также функции, не являющиеся членами классов, и глобальные переменные.

Теперь обсудим другой вопрос. Как мы уже видели в *листинге 23*, если в базовом классе все члены открытые, а при определении производного класса используется ключевое слово **struct,** то все унаследованные им члены базового класса так и остаются открытыми. Но возможны и иные ситуации, как это показано в *листинге 25*.

Листинг 25. Модификаторы доступа.

```
class Bl
{
private: // Метка private здесь избыточна, так как члены класса, определенного при
// помощи ключевого слова class, являются закрытыми по умолчанию. Напоминаем, что
// личные (private) члены класса могут использоваться только функциями-членами и
// дружественными функциями своего класса.
      int i1;
protected: // А что такое protected?  Protected означает защищенный. Защищенные
// члены класса могут использоваться только функциями-членами и дружественными
// функциями своего И ПРОИЗВОДНЫХ классов.
      int j1;
public:
      int k1;
};
class B2
{
      int i2;,
protected:
      int j2;
public:
      int k2 ;
};
// Чтобы указать, какие атрибуты доступа получат члены базового класса в производном
// классе, в списке базовых классов используются ключевые слова private и public.
// Если производный класс определяется при помощи ключевого слова class, то по
// умолчанию принимается private, а если struct - то public.
class D: private B1, public B2
{
// . . .
// i1 и i2 для функций-членов и дружественных функций класса D недоступны.
// j1 и k1, унаследованные от класса B1, получают в классе D атрибут доступа "закрытый".
// j2 и k2 унаследованные от класса B2, не изменят в классе D своих атрибутов и остану-
// тся, соответственно, "защищенный" и "открытый".
// . . .
};
```

В этом фрагменте программы наряду с уже известными нам атрибутами доступа **private** и **public** используется не встречавшийся еще в наших примерах атрибут **protected** - «защищенный». Защищенные члены класса подобны закрытым, но, в отличие от них, доступны функциям-членам и дружественным функциям производных классов. Из *листинга 25* видно также, что при помощи модификаторов доступа **(access modifiers) private и public,** которые ставятся в списке базовых классов перед их именами, можно управлять тем, какие права доступа членов базовых классов будут «делегированы» производному классу. Соотношения атрибутов доступа приводятся в *таблице 1*.

Как вы уже, наверное, заметили, в производном классе доступ к унаследованным членам базового класса может быть либо оставлен прежним, либо еще более ограничен. Еще раз подчеркнем, что «скрывать как можно больше» — признак хорошего стиля ООП.

Таблица 1. **Соотношение атрибутов доступа в базовом и производном классе.**

Доступ в баз. классе	Модификатор доступа	Модификатор доступа

открытый	public	открытый
закрытый	public	недоступен
защищенный	public	защищенный
открытый	private	закрытый
закрытый	private	недоступен
защищенный	private	закрытый

Виртуальные базовые классы.

Синтаксис Си++ запрещает непосредственно передавать базовый класс в производный более одного раза, т. е. имя класса в списке базовых классов не может повторяться:

```
class A {...};
class B: A, A {...};        // ОШИБКА!
```

Интересная ситуация иногда возникает при множественном наследовании, когда базовый класс может быть косвенно унаследован производным классом более одного раза, как например (конечно, можно придумать еще более сложные схемы):

```
class A
{
public:
  int i ;
// . . .
};
class B: public A {/*...*/};
class C: public A, public B {/*...*/};
```

При этом получается, что каждый объект класса **C** будет иметь в своем составе как бы два «подобъекта» класса **A,** один из которых унаследован непосредственно, а другой — косвенно, через класс **B**. Какие это может вызвать проблемы и как их избежать? Самое очевидное затруднение заключается в невозможности для объектов класса **C** получить доступ к членам, унаследованным от класса **A**. Действительно:

```
C obj_c;
obj_c.i=l;        // Какой член i использовать: унаследованный непосредственно или
                  // через класс B?!
                  // Компилятор не может решить...
```

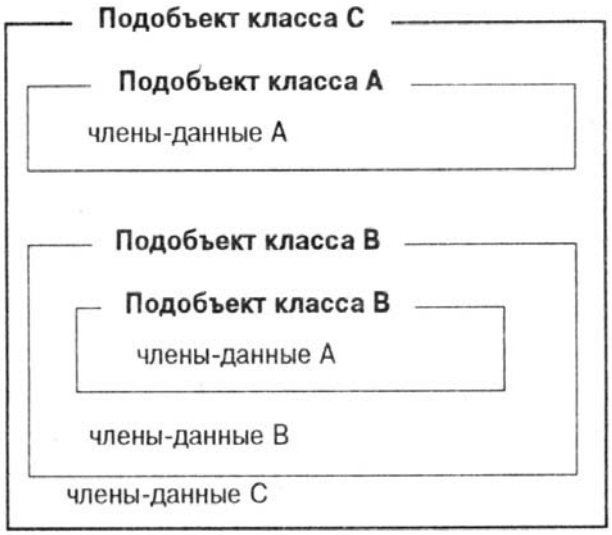
«Множественность» подобъектов базового класса в составе объекта производного класса в ряде случаев приводит также к неразрешимым противоречиям при вызове конструкторов в процессе инициализации объекта и к другим неприятным последствиям.

Трудностей, связанных с неоднократным наследованием производным классом одного базового класса можно избежать, используя виртуальные базовые классы **(virtual base classes).** При «обычном» наследовании объект производного класса содержит в своем составе *подобъект* базового класса, тогда как при виртуальном наследовании объект производного класса содержит скрытый *указатель на подобъект* виртуального базового класса. Этот указатель компилятор неявно использует при работе с объектом для доступа к членам-данным, унаследованным от виртуального базового класса. На рис. 2 приведена схема строения объекта класса C при обычном и виртуальном наследовании.

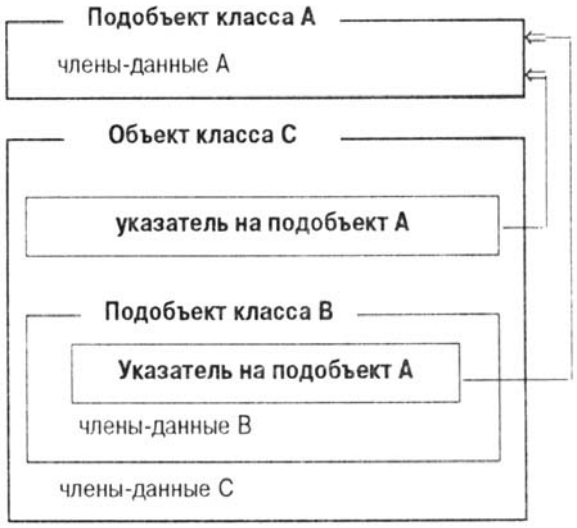
Конечно, при создании объекта класса, имеющего как виртуальные, так и обычные базовые классы, конструкторы виртуальных базовых классов вызываются до вызова конструкторов невиртуальных базовых классов, независимо от расположения базовых классов в списке.

Рис. 2. **Строение объектов при обычном и виртуальном наследовании.**

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




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



В *листинге 26* содержится пример, показывающий виртуальные базовые классы «в действии». К нему рекомендуется выполнить следующие задания:

1. Удалите из программы все встречающиеся ключевые слова **virtual** и попробуйте ее откомпилировать. Проанализируйте сообщения об ошибках компиляции.
2. Изучите опции используемого вами компилятора, связанные с обработкой виртуальных базовых классов. Выясните, каков размер (в байтах) скрытого указателя на подобъект виртуального базового класса в вашей системе программирования в зависимости от установленных опций и модели памяти.

Листинг 26. Виртуальные базовые классы.

```
#include <iostream.h>
class A
        {
        long i;
public:
        long get_i () {return i;}
        A(long init_i) {cout << " Вызван констр. A "; i=init_i;}
        ~A() {cout << " Вызван дестр. A ";}
        };
class B: virtual public A
```

```
        {
public:
        B(long i): A(i) {cout << " Вызван констр B ";}
        ~B() {cout << " Вызван дестр. B ";}
        };
class B1: public A
        {
        public:
        B1(long i):A(i) {cout << "Вызван констр. B1";}
        ~B1() {cout << " Вызван дестр. B1";}
        };
class C: public B, virtual public A
        {

public:
        C(long j): B(j), A(j)
        {cout << " Вызван констр. C ";}
        ~C() {cout << "Вызван дестр. C ";}
        };
        int main()
        {
        cout  << " Создаем объект класса A\n";
        A a ( i ) ;
        cout  << "\n Размер объекта a=" << sizeof (a) << " байт. \n";
        cout  << "\nСоздаем объект класса B1\n";
        B1 b1(1) ;
        cout  << "\n Размер объекта b1=" << sizeof (b1) << " байт. \n";
        cout  << "\n Создаем объект класса B \n";
        B b (1) ;
        cout  << "\n Размер объекта b=" << sizeof (b) << " байт. \n";
        cout  << "\n Создаем объект класса C\n";
        C c (1) ;
        cout  << "\n Размер объекта c=" << sizeof (c) << " байт. \n";
        return 0 ;
        }
```

Преобразования указателей на объекты.

Если производный класс **Derived** имеет открытый базовый класс **Base,** то указатель на **Derived** можно присваивать переменной типа «указатель на **Base**» без явного преобразования типа. Обратное преобразование указателя на **Base** в указатель на **Derived** должно быть явным. Например:

```
class Base { ... };
class Derived : public Base { ... };
Derived d;
Base* b_ptr = &d;  // неявное преобразование Derived* d_ptr;
d_ptr = b_ptr;             // ОШИБКА!
d_ptr = (Derived*) b_ptr;  // явное преобразование
```

Преобразования указателей на объекты «вверх и вниз по иерархической лестнице» широко используются, в частности, когда некоторый вмещающий класс (например, массив, стек, очередь) содержит указатели на объекты разных классов, имеющих общий базовый класс. С такой ситуацией мы еще не раз встретимся в дальнейшем.

Соглашение об именах производных типов.

Для улучшения читаемости текстов программ на Си++ рекомендуется следовать общепринятым соглашениям об именах производных типов, таких как ссылки и указатели. По этим соглашениям имя производного типа образуется из имени основного типа, к которому добавляется соответствующий префикс согласно таблице 2.

Таблица 2. **Стандартные имена для производных типов данных.**

Тип	**Имя**	**Макрос для генерации имени**
aType	aType	- -
aType*	PaType	#define_PTRDEF(name)\ typedef name * P##name;
Const aType*	PCaType	#define_PTRCONSTDEF(name)\ typedef const name & RC##name;
aType&	RaType	#define_REFDEF(name)\ typedef name & R##name;
Const aType&	PCaType	#define_REFCONSTDEF(name)\ typedef const name & RC##name;
aType*&	RP aType	#define_REFPTRDEF(name)\ typedef name * & RP##name;

В таблице также приведены содержащиеся в заголовочном файле **defs.h** Borland C++ 3.0 макросы с параметрами, предназначенные для автоматической генерации имен производных типов. Макрос **_CLASSDEF,** также имеющийся в **_defs.h,** генерирует все вышеперечисленные имена типов, производных от некоторого класса.

```
#define _CLASSDEF(name) class name;\
_PTRDEF (name)\
_REFDEF(name)\
_REFPTRDEF(name)\
_PTRCONSTDEF(name)\
_REFCONSTDEF(name)
```

Если вы пишете сложную программу, в которой многократно встречаются производные типы от многих классов, имеет смысл включить в соответствующие модули заголовочный файл **_defs.h** и пользоваться макросом **_CLASSDEF** при определении классов. Например:

```
#include <_defs.h>
```

```
_CLASSDEF(MyClass)
class MyClass {/* определение класса...*/};
// . . .
// используем имена типов, производных от MyClass:
RMyClass func(int n, RMyClass obj);
// . . .
MyClass obj;
PMyClass obj_ptr=bobj;
RCMyClass ref:_const_obj=obj ;
// . . .
```

В примерах, приводимых в данной книге, мы также будем придерживаться соглашения об именах производных типов.

Полиморфизм.

Полиморфизм **(polymorphism)** — последний из трех главных «китов», на которых держится объектно-ориентированное программирование. Слово это можно перевести с греческого как «многоформенность». Применительно к ООП на языке Си++ этот термин обычно трактуют как способность объекта отреагировать на некоторый запрос (т. е. вызов функции-члена) сообразно своему типу, даже если на стадии компиляции тип объекта, к которому направлен запрос, еще неизвестен. В языке Си++ полиморфизм реализуется при помощи механизма виртуальных функций-членов.

Виртуальные функции-члены.

При вызове функции-члена некоторого класса у компилятора есть несколько хорошо уже нам знакомых способов связать имя вызываемой функции с соответствующим ей машинным кодом на стадии компиляции. Однако, если для вызова функции-члена используется указатель на класс, у которого есть производные и базовые классы, имеющие функции с тем же именем и одинаковым набором аргументов, компилятор не в состоянии определить, к какому конкретно классу относится указываемый объект и какую функцию для него вызывать. В *листинге 27* представлена типичная ситуация, когда механизм раннего связывания не может обеспечить вызов функции-члена, соответствующей реальному типу указываемого объекта:

Листинг 27. Вызов функции-члена при помощи указателя на базовый класс.

```
class Base
        {
        void f();
        /*...*/
        };
class Deriv1: public Base
        {
        void f() ;
        /*...*/
        };
class Deriv2: public Base
        {
        void f() ;
        /*...*/
        };
class Deriv3: public Deriv2
        {
        void f() ;
        /*...*/
        };
```

```
// . . .
Base b_ptr[3] ;
b_ptr[0]=new Deriv1
b_ptr[1]=new Deriv2
b_ptr[2]=new Deriv3
/ / . . .
for (int count=0; count<3; count++)
        b_ptr -> f();  // для всех объектов будет вызвана Base::f()
```

Обыкновенно указатели на общий для нескольких классов базовый класс используются при разработке вмещающих классов: множеств, векторов, списков и т. п. При вызове функции-члена при помощи указателя необходимо решить проблему распознавания класса, к которому принадлежит указываемый объект, для того, чтобы вызываемая функция-член корректно выполнила над этим объектом требуемые действия. Эта проблема имеет три возможных решения.

Первое: обеспечить, чтобы всегда указывались только объекты одного типа. Это, конечно, радикальное решение, но оно вносит существенные ограничения. В частности, при создании вмещающих классов оно приводит к тому, что могут быть со зданы лишь однородные вмещающие классы, содержащие указатели на объекты только одного типа. При этом вообще лучше с точки зрения эффективности, чтобы однородный вмещающий класс содержал не указатели на объекты, а сами объекты (в таком случае для автоматической генерации вмещающих классов, соответствующих типу помещаемых в него объектов, весьма разумно использовать шаблоны).

Второе: поместить в базовый класс поле типа, которое смогут просматривать функции. Это решение уже несколько лучше, но только в том случае, если как сам базовый класс, так и классы, производные от такого базового класса, создает и использует один программист, да и то в пределах небольших программ (см. листинг 28).

Листинг 28. Можно поместить в базовый класс поле типа.

```
// каждый класс получит свой идентифицирующий номер
enum CLASS_ID (ID_Base, ID_Deriv1, ID_Deriv2);
class Base
        {
        protected:
        CLASS_ID class_TD;
        public:
        void f () ;
        Base() {class_ID=ID_Base; /*...*/}
        };
class Derivl: public Base
        {
        // . . .
        friend void Base:: f();
        Deriv1() {class_ID=ID_Derivl; /*...*/}
        };
class Deriv2: public Base
        {
        // . . .
        int some_ipember;
        friend void Base::f();
        Deriv2() {class_ID=ID_Deriv2; /*...*/}
        };
void Base::f()
{
switch (class_ID)
        {
```

```
        case ID_Base:
                // операторы...
                break;
        case ID_Deriv1:
                // операторы...
                break;
        case ID_Deriv2:
                // так можно добраться до члена класса Deriv2:
                ((Deriv2*)this) -> some_member++ ;
                // операторы...
                break;
        default:
                // обработка случая неизвестного идентифицирующего номера класса...
                ;
        }
}
```

Действительно, такая программа получится весьма запутанной... Вообразите мучения программиста, пожелавшего добавить в иерархию свой собственный класс, для объектов которого функция **f** также должна работать правильно. Он не сможет сделать это иначе, чем модифицировав исходный текст **f** – функции-члена класса **Base.**

И, наконец, **третье** решение – использование виртуальных функций – фактически перекладывает на компилятор все заботы о помещении в класс поля типа, генерации кода для его проверки (во время выполнения программы!) и вызова функции-члена, соответствующей реальному типу объекта.

Если в некотором классе имеется функция, описанная как **virtual,** то в такой класс компилятором добавляется скрытый член — указатель на таблицу виртуальных функций, а также генерируется специальный код, позволяющий осуществить выбор виртуальной функции, подходящей для объекта данного типа, во время работы, программы (а не во время компиляции!). Ситуация, когда имя функции связывается с соответствующим ей кодом во время работы программы, получила название позднего связывания **(late binding).** Конечно, использование виртуальных функций снижает быстродействие программы и увеличивает размер объектов классов, содержащих такие функции, но это — неизбежное противоречие: гибкость или эффективность? Фактически каждый объект класса, име ющего виртуальные функции, несет информацию о своем типе.

Если некоторая функция объявлена в базовом классе как виртуальная, то функция с тем же именем, с таким же списком аргументов и тем же типом возвращаемого значения, переопределенная в производном классе, автоматически становится виртуальной, и ключевое слово **virtual** при ее описании в производном классе уже можно не использовать. Нельзя переопределять в производном классе функцию, отличающуюся от виртуальной функции-члена базового класса только типом возвращаемого значения. Само собой, виртуальная функция должна быть функцией-членом некоторого класса, однако она не должна быть статической функцией-членом.

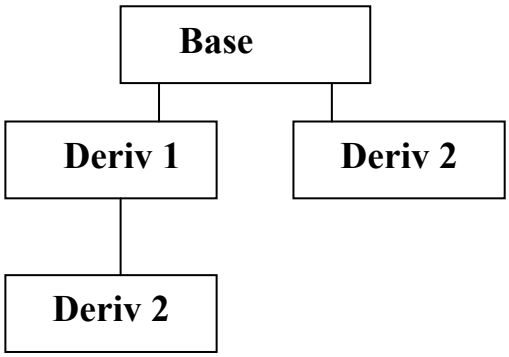
Если виртуальная функция не была переопределена в производном классе, то при ее вызове для объекта этого производного класса будет происходить обращение к соответствующей функции из ближайшего по иерархической лестнице базового класса, в котором она была определена.

Заметим, что членом производного класса вполне может быть функция с именем, совпадающим с именем виртуальной функции-члена базового класса, но с отличающимся списком аргументов, и она будет обыкновенной невиртуальной функцией.

В листинге 29 приведен пример программы, позволяющий сравнить виртуальные и невиртуальные функции, вызываемые различными способами.

Листинг 29. Использование виртуальных функций.

```
// иерархия классов:
//
//
//
//
//
//
//
//
//
//
//
```




```
#include <iostream.h>
const char *const f_msg =" Вызвана функция ";
class Base
        {
        int dummy;
public:
// каждая функция при вызове сообщает информацию о себе
void f() {cout << f_msg << "Base :: f()\n";}
virtual void f(int i)
        {cout << f_msg << " Base:: f (int)\n";}
void f(int i, int j)
        {cout << f_msg << "Base:: f(int, int)\n";)
};
class Derivl: public Base
        {
        char dummy1;
public:
        void f() {cout << f_msg << "Deriv1:: f()\n";}
        virtual void f(int i)
             {cout << f_msg << "Deriv1::f(int)\n";}
        };
class Deriv2: public Base
        {
public:
        void f() (cout << f_msg << "Deriv2:: f()\n";}
        virtual void f(int i)
             {cout << f_msg << "Deriv2::f(int)\n";}
        };
class Derivl2: public Deriv1
        {int dummy;
        // В этом классе функции f() и f(int) не переопределены.
        };
typedef Base* PBase;
typedef Deriv1* Pderiv1;
int main()
{
cout << "\n**********************************\n";
Base b_obj;
Deriv1 d1_obj;
```

```
Deriv2 d2_obj;
Deriv12 d12_obj;
cout << " Функции-члены вызываются непосредственно: \n";
b_obj.f();
b_obj.f(1);
d2_obj.f(1);
d2_obj.Base::f(1) ;
// d2_obj.f(1,1); Была бы ОШИБКА! См: коммент. в листинге 24.
d2_obj.Base::f(1,1) ;    //Так правильно!
PBase b_ptr[4]; //массив из трех указателей на Base
b_ptr[0]=&b_obj ;
b_ptr[1]=&dl_obj; // неявное преобразование Denv1* в Base*
b_ptr[2]=&d2_obj;  // неявное преобразование Denv2* в Base*
b_ptr[3]=&dl2_obj; //неявное преобразование Deriv12* в Base*
// имена классов
char* types[4]={"Base", "Deriv1", "Deriv2", "Deriv12"} ;
char* msg = "Тип указываемого объекта:";
cout << "Функции-члены вызываются через указатель на класс:\n";
cout << "Невиртуальная функция f() ********************\n";
int count ;
cout << "Тип указателя Base*\n";
// Программа при работе будет сообщать тип указателя, тип указываемого объекта и к
// какому классу принадлежит вызванная функция.
for (count=0; count<4; count++)
        {cout << msg << types[count]; b_ptr[count]->f();}
cout << "Тип указателя Deriv1 *\n";
for (count=0; count<4; count++)
        {
        cout << msg << types[count];
        (PDerivl(b_ptr[count]))->f();
        }
cout << "Виртуальная функция f(int)************************\n";
// Какая функция будет вызвана для b_ptr[3]?
cout << "Тип указателя Base*\n";
for (count=0; count<4; count++)
        {
        cout << insg << types[count] ;
        b_ptr[count]->f(1) ;
        }
cout << "Тип указателя Deriv1 *\n";
for (count=0; count<4; count++)
        {
        cout << msg << types[count];
        (PDerivl(b_ptr[count]))->f(1);
        }
cout << "************************************\n";
//Вот так все-таки можно заставить компилятор вызвать виртуальную функцию, НЕ
// соответствующую типу объекта, на который указывает указатель!
(Pderivl(b_ptr[2])) -> Deriv1::f(1);
return 0;
}
```

У внимательного читателя, вероятно, уже назрел такой вопрос: а как же реализуется механизм виртуального вызова, если виртуальные функции из листинга 29 являются подставляемыми

— ведь они же определены внутри определения класса? Компиляторы языка Си++ поступают весьма разумно: подстановка тела подставляемой виртуальной функции осуществляется только при тех вызовах, где механизм виртуального вызова игнорируется (иначе говоря, реализуется раннее связывание). Я думаю, вам нетрудно будет найти в листинге 29 такие вызовы.

Интересно отметить, что механизм виртуального вызова фактически позволяет нарушить запрет на обращение извне к закрытым функциям-членам. Действительно, представим себе такую ситуацию:

```
class Base
{
// . . .
public:
virtual void f();
// . . .
};
class Deriv: public Base
{
// . . .
private:
virtual void f() ; // Закрытая функция!
// . . .
};
// . . .
Deriv d_obj;
Base* b_ptr=&d_obj ; // Неявное преобразование типа.
b_ptr->f () ; // Будет вызвана закрытая функция Deriv::f()!
```

Виртуальные деструкторы

Конструктор класса, конечно, не может быть виртуальным, поскольку он вызывается только при создании объекта, и уж, разумеется, тип создаваемого объекта должен быть известен компилятору. С деструкторами же ситуация иная: вообразите, что оператор **delete** разрушает объект, адресуемый указателем типа «указатель на базовый класс». А вдруг этот указатель на самом деле указывает на объект производного класса, имеющего свой собственный деструктор? Проблема корректного разрушения указываемых объектов решается при помощи использования виртуальных деструкторов. В листинге 30 это показано на примере классов фигур, производных от базового класса **Figure.**

Листинг 30. Виртуальные деструкторы.

```
Class Figure
{
public:
Figure() ;
virtual ~Figure();
};
typedef Figure* PFigure;
class Circle: public Figure
{
public:
Circle(int CenterX, int CenterY, int Radius);
virtual ~Circle(); // Также будет виртуальным.
// Ключевое слово virtual здесь избыточно.
};
class Rectangle: public Figure
{
```

```
public:
Rectangle(int Left, int Top, int Right, int Bottom) ;
~Rectangle(); // Также будет виртуальным.
};
int main()
{
//  . . .
// .Программа создает массив указателей на фигуры:
const NallFigures=2
PPigure figures[NAllFigures];
figures [0] =new CircledOO, 100, 10);
figures[1]=new Rectangle(100, 100, 200, 250);
//  . . .
//  А теперь настало время их уничтожить:
for (int count=0; count<NAllFigures; count++)
   delete figures[count];
// Если бы деструкторы не были виртуальными, то для всех элементов массива вызывался
// бы деструктор ~Figured!
//  . . .
   }
```

Таким образом, использование виртуальных деструкторов позволяет обеспечить вызов соответствующего деструктора при разрушении объектов оператором **delete,** даже если тип разрушаемого объекта неизвестен на стадии компиляции.

Абстрактные классы.

Если при определении базового класса некоторая виртуальная функция объявлена членом этого класса, то она должна быть определена подобно обычным функциям-членам, либо же объявлена как абстрактная или «чистая» функция **(pure function)** при помощи спецификатора **=0**.

```
class Base
{
virtual void f(int) =0;
// . . .
};
```

Класс, содержащий хотя бы одну абстрактную функцию, называется абстрактным классом. Абстрактный класс может служить только в качестве базового для других классов – объект такого класса создать невозможно. В производных от него классах абстрактные функции должны быть либо определены, либо вновь объявлены как абстрактные. Абстрактный класс нельзя указать в качестве типа аргумента или возвращаемого значения функции. Однако разрешено (и это очень часто используется) создать указатель на абстрактный базовый класс, а также ссылку на такой класс, если для ее инициализации не требуется создания временного объекта.

Функции-члены абстрактного класса могут вызывать абстрактные функции-члены этого же класса (в таком случае будет вызвана соответствующая типу объекта функция, определенная в производном классе). Абстрактный класс может иметь конструкторы и деструктор. Конструктор абстрактного класса будет вызываться при создании объектов производных классов, а деструктор — при их разрушении. Следует помнить, что деструктор базового класса вызывается при разрушении объекта в последнюю очередь, когда уже разрушены «подобъекты», определенные в производных классах, поэтому деструктор базового класса не должен вызывать абстрактные функции-члены своего класса, так как такой вызов приведет к ошибке во время выполнения программы.

```
class Figure
{
// . . .
public:
```

```
Figure();
virtual void Show() =0;
virtual void Hide() =0;
virtual void Expand(int ExpandBy) =0;
void Contract(int ContractBy)
{
Expand(-ContractBy); // OK!
}
virtual ~Figure() {Hide();} // ОШИБКА при выполнении программы!
} ;
```

Библиотека потокового ввода-вывода.

В языке Си++, как и в Си, отсутствуют встроенные операторы ввода-вывода. В программах на Си и Си++ ввод и вывод обычно осуществляется при помощи функций ввода-вывода, содержащихся в библиотеках. Стандарт Си определяет набор функций ввода-вывода **(stdio),** которые обязательно должны входить в стандартную библиотеку Си. Прототипы стандартных функций ввода-вывода содержатся в заголовочном файле **stdio.h.** Хотя все стандартные функции Си также доступны в Си++, при программировании на Си++ во многих случаях предпочтительнее использовать специальную стандартную библиотеку потокового ввода-вывода **iostreams,** содержащую большой набор классов и функций для буферизованного и небуферизованного ввода-вывода данных как для файлов, так и для устройств.

Мы уже в некоторой степени знакомы с использованием потокового ввода-вывода и предопределенными объектами ввода-вывода **cin, cout, cerr и clog,** однако, в данный момент имеет смысл немного расширить наши представления о потоках и о возможностях библиотеки **iostreams.**

Что же такое, в сущности, потоки ввода-вывода **(input-output streams)?** В полном соответствии с логикой Си++ поток определяется как некоторый абстрактный тип данных, представляющий некую последовательность элементов данных, направленную от источ-ника **(source, or producer)** к потребителю **(consumer, or sink).** Количество элементов в потоке называют его длиной **(length),** порядковый номер доступного в некоторый момент элемента называют текущей позицией **(current position).** Каждый поток имеет один из трех возможных режимов доступа **(access mode):**

- только для чтения;
- только для записи;
- для чтения и записи.

Иерархия классов, входящих в библиотеку **iostreams,** довольно сложна, и для подробного ее изучения следует обратиться к соответствующим справочным руководствам. В данном параграфе мы весьма кратко рассмотрим назначение некоторых классов библиотеки и на простом примере убедимся, что ввод-вывод с использованием потоков действительно осуществляется проще и нагляднее, чем стандартными средствами Си.

Библиотека **iostreams** содержит два параллельных семейства классов. Первое ведет свое начало от класса **streambuf,** а второе – от **ios.** Класс **streambuf** и два класса, производных от него – **filebuf и strstreambuf,** – обеспечивают буферизацию потокового ввода-вывода и абстрагируют обращение к физическим устройствам ввода-вывода. Класс **ios** дает начало семейству классов, предназначенных для реализации форматированного и неформатиро-ванного ввода-вывода на высоком уровне с возможностью контроля и изменения состояния потока. **ios** содержит член **ios::bp** – указатель на **streambuf,** – посредством которого и «общается» с устройствами ввода-вывода. Среди классов, производных от **ios,** следует назвать следующие:

istream – ввод из стандартного устройства ввода. Содержит перегруженный оператор форматированного ввода » и ряд функций неформатированного ввода;

ostream – вывод на стандартное устройство вывода. Содержит перегруженный оператор форматированного вывода « и ряд функций неформатированного вывода;

iostream – «смесь» классов **istream и ostream;**
iostream_withassign – **iostream** с перегруженным оператором присваивания.

Объектами последнего класса являются стандартные объекты ввода-вывода Си++ **cin, cout, cerr** и **clog.**

ifstream – файловый ввод;
ofstream – файловый вывод;
istrstream – ввод из строки;
ostrstream – вывод в строку.

Форматирование при вводе-выводе данных, относящихся к встроенным в Си++ типам, определяется рядом флагов форматирования, являющихся членами-данными **ios**. Состояние флагов форматирования может быть изменено вызовом соответствующих функций-членов **ios** или, как мы уже видели в *листинге 18*, при помощи манипуляторов.

В *листинге 31* приводится пример программы, подсчитывающей, сколько раз заданная строка встречается в текстовом файле. Программа иллюстрирует чтение из потока, связанного с дисковым файлом.

Листинг 31. Чтение из потока, связанного с дисковым файлом.

```
#include <fstream.h> // автоматически включает iostream.h
#include <string.h>

int main()
{
const MaxLineLength=255; // максимальная длина строки, читаемой из файла
char inbuf [MaxLineLength+1]; //буфер ввода
cout << " Эта программа считает, сколько раз заданная \n"
            "строка встречается в текстовом файле.\n";
cout << " Введите имя файла:";
char fname[80];
cin >> fname;
if stream ifs(fname); // Создаем объект потокового вывода, ассоциированный с
// файлом filename.
if (!ifs)    // Файл открыт успешно?
        {cerr << "Невозможно открыть файл!"; return 1;}
cout << " Введите строку для поиска:";
char SearchString[MaxLineLength+1] ;
cin >> SearchString;
unsigned line_counter=0,      // счетчик числа прочитанных строк файла
str_counter=0; // счетчик встретившихся строк поиска
int SearchStrLen=strlen(SearchString) ;
while (!ifs.eof()) // пока не достишут конец файла ...
        {
        ifs.getline (inbuf, MaxLineLength, '\n'); // Прочитать строку из
        // потока в буфер inbuf. Чтение происходит, пока не достигнут символ '\n' или конец
        // файла, либо пока не прочитано MaxLineLength-1 символов.
        line_counter++ ;
        char* TmpStr=inbuf;
// Функция strstr находит в строке заданную подстроку, возвращая ее адрес или NULL в
// случае неудачи. Ее прототип содержится в string.h. В цикле while подсчитываются все
// случаи встречи SearchString в строке, прочитанной из потока.
        while ((TmpStr=strstr(TmpStr, SearchString))i=NULL)
                {
                str_counter++ ;
                if (strlen(TmpStr)< SearchStrLen*2) break;
// Больше строка поиска не встретится. Необходимо выйти из цикла, чтобы не
```

```
// перепрыгнуть '\x0'.
                TmpStr+=SearchStrLen; //Перескочить через найденную строку поиска.
                }
        }
cout << "Программа просмотрела" << line_counter << "строк(и) текста.\n";
cout<<"Строка \'"<<SearchString<<"'\ встретилась"<<str_counter<< "раз(а).";
return 0;
}
```