Конспект лекций по курсу "Обьектно-ориентированное программирование"

для студентов специальности 31.03.04 "Информатика"

Оглавление

Лекция 1. Концепция объектно-ориентированного программирования. Понятие объекта и фундаментальные характеристики ООП (инкапсуляция, наследование, полиморфизм).

Объектно-ориентированное программирование (ООП) - это новый подход к созданию программ. По мере развития вычислительной техники возникали разные методики программирования. На каждом этапе создавался новый подход, который помогал программистам справляться с растущим усложнением программ. Так язык программирования (даже высокого уровня), легко понимаемый в коротких программах, становился нечитабельным в более длинных. Ситуацию разрешило изобретение в 1960 году языков структурного программирования (к ним относятся Алгол, Паскаль и С). Структурное программирование подразумевает точно обозначенные управляющие структуры, программные блоки, отсутствие (минимальное использование) инструкций GOTO, автономные подпрограммы, в которых поддерживаются локальные переменные и рекурсия. Сутью структурного программирования является возможность разбиения программы на составляющие ее элементы. Хотя структурное программирование в свое время принесло выдающиеся результаты, чтобы написать более сложную программу, отвечающую современным требованиям, необходим новый подход к программированию. Объектно-ориентированное программирование позволяет разложить проблему на составные части, каждая из которых становится самостоятельным объектом. Каждый из объектов содержит свой собственный код и данные, которые относятся к этому объекту. Давайте обратимся к понятию “объект”. Технология ООП, прежде всего, накладывает ограничения на способы представления данных в программе. Любая программа отражает в них состояние физических предметов либо абстрактных понятий (назовем их объектами программирования) для работы, с которыми она предназначена. В традиционной технологии варианты представления данных могут быть разными. В худшем случае программист может “равномерно размазать” данные о некотором объекте программирования по всей программе. Например, имеется задача, условие которой гласит: “Определить принадлежит ли точка с заданными координатами заштрихованной области”, и приводится рисунок. Объектами программирования такой задачи выступают “точка” и “заштрихованная область”. Скорее всего, в решении этой задачи для структурного языка программирования как раз и наблюдается принцип “размазанности” данных: координаты точек приходиться хранить в независимых переменных X и Y, информация о заштрихованной области храниться на протяжении всей программы в виде операторов ветвления.

В противоположность такому подходу все данные об объекте программирования и его связях с другими объектами можно объединить в одну структурированную переменную. В первом приближении ее можно назвать объектом. Данные об объекте программирования объединены в большинстве случаев в структуру - структурированную переменную. Кроме того, с объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования набор действий или методов это функции, получающие в качестве обязательного параметра указатель на объект и выполняющие определенные действия с данными объекта программирования. Технология ООП запрещает работать с объектом иначе, чем через методы, таким образом, внутренняя структура объекта скрыта от внешнего пользователя. Описание множества однотипных объектов называется классом.

Объект - это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс - это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Это определение можно проиллюстрировать средствами классического С:

*struct myclass /*класс - описание множества объектов*/*

```
{ int data1;
...
};
void method1(struct myclass *this,...) /*метод*/
{ ... this->data1 ... }
void method2(struct myclass *this,...) /*метод*/
{ ... this->data1 ... }
struct myclass obj1, obj2; /*объекты*/
... method1(&obj1,...); /*для работы с объектами*/
... method2(&obj2,...); /*применяются методы*/
```

Основные понятия объектно-ориентированного программирования: **инкапсуляция**, **наследование** и **полиморфизм**.

“Эпизодическое” использование технологии ООП заключается в разработке отдельных, не связанных между собой классов и использовании их как необходимых программисту базовых типов данных, отсутствующих в языке. При этом общая структура программы остается традиционной (“от функции к функции”). Однако, строгое следование технологии ООП предполагает, что любая функция в про грамме представляет собой метод для объекта некоторого класса. Это не означает, что нужно вводить в программу какие попало классы ради того, чтобы написать необходимые для работы функции. Наоборот, класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования. С другой стороны, каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих. В конце концов, вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить). Именно этот переход (а не понятия класса и объекта, как таковые) создает психологический барьер перед программистом, осваивающим технологию ООП.

Программирование “от класса к классу” включает в себя ряд новых понятий. Прежде всего, это - инкапсуляция данных. ***Инкапсуляция*** - это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

В ООП код и данные могут быть объединены вместе (в так называемый “черный ящик”) при создании объекта. Внутри объекта коды и данные могут быть закрытыми или открытыми. Закрытые коды или данные доступны только для других частей того же самого объекта и, соответственно, недоступны для тех частей программы, которые существуют вне объекта. Открытые коды и данные, напротив, доступны для всех частей программы, в том числе и для других частей того же самого объекта.

Для примера, рассмотрим такую структуру хранения информации как стек. Пользователя интересует выполнение операций: push(), top(), empty(), pop(), reset(), full(). Внутренняя реализация (адреса ячеек памяти, значения регистров процессора, имена массивов и т.д.) пользователя не интересует, то есть она должна быть скрыта от него. Пользователю нужны только указанные операции, и он должен получить в свое распоряжение набор методов для их выполнения. Объектом программирования в этом случае является структура хранения информации типа стек. Соответственно, внутренняя организация объекта стек должна быть недоступна (закрыта) для пользователя, то есть все его попытки обратиться непосредственно к элементам, обеспечивающим функционирование структуры хранения данных, должны отвергаться (выдавать сообщение об ошибке). А вызов методов, напротив, должен приводить к выполнению желаемого действия и является общедоступным.

Вторым по значимости понятием является ***наследование***. Новый, или производный класс может быть определен на основе уже имеющегося, или базового класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в

данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки. Если объект наследует свои свойства от одного родителя, то говорят об одиночном наследовании. Если же объект наследует данные и методы от нескольких базовых классов, то говорят о множественном наследовании. Простой пример наследования - определение структуры, отдельный член которой является ранее определенной структурой. Рассмотрим еще один пример. Например, базовый класс “животные” может иметь производные классы: “млекопитающие”, “рыбы”, “птицы” и т.д. Они будут наследовать все характеристики базового класса, но каждый из них может иметь и свои собственные свойства:
“рыбы – плавники”, “птицы – крылья” и т.д.

Третьим по значимости понятием является полиморфизм. ***Полиморфизм*** - это свойство, которое позволяет один и тот же идентификатор (одно и то же имя) использовать для решения двух и более схожих, но технически разных задач. Целью полиморфизма, применительно к ООП, является использование одного имени для задания действий, общих для ряда классов объектов. Такой полиморфизм основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится “самодостаточным”. Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о его типе, он всегда может корректно вызвать свойственные ему методы. Таким образом, полиморфная функция - это семейство функций с одним и тем же именем, но выполняющие различные действия в зависимости от условий вызова.
Например, нахождение абсолютной величины в языке С требует трех разных функций:
int abs(int);
long labs(long);
double fabs(double);
Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. С точки зрения полиморфизма, каждую из этих функций может быть названа abs(), а тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется.

Лекция 2. Не связанные с объектами расширения C++ относительно C. Новое в описании типов и переменных. Перегружаемые функции и функции с аргументами по умолчанию.

Язык C++ не появился на "голом" месте, на его синтаксические структуры определенное влияние оказали различные конструкции других языков программирования. В основу C++, безусловно, был положен язык программирования C, разработанный Денисом Ричи. Кроме того, предшественником C++ явился язык BCPL. Из языка Simula67 были позаимствованы концепция классов вместе с производными классами и функциями-членами. Перегрузка операций и свобода в расположении описаний взяты из Алгол68.

Название C++ появилось летом 1983 г. Более ранние версии этого языка использовались с 1979 г. под общим названием “C с классами”. Название “C++” придумал Риск Маскитти (Rick Mascitti). Это название отражает эволюционный характер изменения языка C. Согласно одной из интерпретаций названия языка “++” - это оператор инкрементации в языке C.. Существуют и другие интерпретации названия, тем не менее язык не называется D, потому что он - расширение C. Фактически язык C++ представляет собой полный язык C с надстройкой для реализации идей ООП, а также некоторыми дополнительными синтаксическими структурами, включенными для удобства пользователей. Все основные операции, операторы, типы данных языка C присутствуют в C++. Некоторые из них усовершенствованы и добавлены принципиально новые конструкции, которые и позволяют говорить о C++ как о новом языке, а не просто о новой версии языка C. К настоящему времени язык C++ претерпел несколько существенных модернизаций, последняя из которых связана с процедурой стандартизации (1998 г.). Вследствие этого более ранние реализации языка отличаются друг от друга. При этом мы будем рассматривать те возможности языка C++, которые верны для большинства реализаций, и их принято считать общими, полагая что Standard C++ является надмножеством традиционного C++.

Дополнительные ключевые слова. C++ расширен следующими ключевыми словами:

class	delete	friend	inline
new	operator	private	protected
public	template	this	virtual

Комментарии. Символы /* открывают комментарий, оканчивающийся символами */. Вся такая последовательность эквивалентна игнорируемому символу (например, пробелу). Это наиболее удобно для написания многострочных комментариев, и является единственным видом комментариев в языке C. Но как только что было сказано, все, что относится к C справедливо и для C++. Кроме того, для написания коротких комментариев, в языке C++ используются символы //

```
int a; /* целая переменная */
float b; // вещественная переменная
```

Переменные. Язык C++ является блочно сконструированным. Переменные, описанные внутри блока, вне блока недоступны. Переменные внешнего блока доступны во внутреннем, если он их не переопределяет. Приведем несколько определений, касающихся идентификаторов переменных (объектов) C++. Контекстом идентификатора называется часть программы, в которой данный идентификатор может быть использован для доступа к соответствующему объекту. Если идентификатор объявлен в блоке, то он имеет контекст блока, который начинается в точке объявления и распространяется до конца блока. Поскольку тело функции

представляет собой блок, то это правило распространяется на локальные переменные функции.

Параметры функции имеют контекст тела функции, например:

```
void func(char ch, int i)  // начало контекста ch, i
{
        int x = 5;            // начало контекста x
        x++;
        int y = x * x + 8;      // начало контекста y
        . . . . . .
        int z = x + y;          // начало контекста z
        . . . . . .
  }  // конец контекста ch, i, x, y, z
```

Если идентификатор объявлен вне всех блоков и функций, он имеет контекст файла, начинающийся в точке объявления и заканчивающийся в конце исходного файла. Объявленные таким образом переменные называются глобальными, например:

```
// файл my_func.c
 float r;          // r, d и h имеют контекст файл my_func.c
double d;
 int h;
 void func() {...}
```

Видимостью идентификатора называется область исходного текста программы, из которого возможен нормальный доступ к соответствующему объекту. Обычно видимость и контекст идентификаторов совпадают. Исключение составляют случаи, когда переменную внешнего (охватывающего) блока временно скрывает переопределение во внутреннем блоке переменной с тем же именем, например:

```
 {
      int i = 3;
      . . . . . .
      for (int k = 0; k<5; k++)
      {
            int i = k;
            // здесь i - новая переменная, изменяющаяся от 0 до 4,
            // скрывает i = 3 внешнего блока
            . . . . . .
      }
      // здесь по-прежнему i = 3
   }
```

Связанные с объявлением идентификаторов объекты характеризуются также продолжительностью. В C++ различают статическую (static), локальную (local) и динамическую (dynamic) продолжительности.

Объектам со статической продолжительностью память обычно выделяется в фиксированной области в начале выполнения программы и сохраняется до выхода из программы. К объектам со статической продолжительностью относятся все функции, независимо от того, где они определены, переменные с файловым контекстом и переменные, имеющие спецификаторы памяти static или extern.

Память объектам с локальной продолжительностью (или локальным переменным) выделяется в стеке при входе в ближайший охватывающий их блок и уничтожается при выходе из блока.

Объекты с динамической продолжительностью создаются и разрушаются в свободной области памяти (куче) путем вызова функций (malloc(), calloc(), free()) или с помощью операций new и delete в процессе выполнения программы, например:

```
float r;       // r, d u h имеют статическую продолжительность
double d;
int h;
void func(char ch, int i)
{
      int x = 5;            // ch, i u x - локальную
      char *str;
      str = (char*)malloc(81*(sizeof(char)));
      . . . . . .
      free(srt);            // srt - динамическую
  }
```

В С переменные инициализируются константными выражениями. В С++ переменные можно инициализировать обычными выражениями. Последнее носит название динамической инициализации переменных, поскольку позволяет инициализировать значения переменных в процессе выполнения программы. Например:

```
void  main(void)
{
   char s[81];
    …….
    int k = strlen(s);          // динамическая инициализация
    …….
}
```

В С++ односимвольные константы имеют тип char (в С они автоматически преобразуются к типу int). С++ поддерживает также двухсимвольные константы типа int , например:

```
'AB', '\n\t', '\t\v'
```

при этом первый символ располагается в младшем байте, а второй - в старшем байте.

Если в С++ символьной константе предшествует L, то она называется широкой символьной константой и имеет тип wchar_t, определенный в stddef.h, например:

```
x = L'abc';
```

Константы. Константы в языке С++ аналогичны константам в С. Отличие состоит в том, что для символьного представления константы в С использовалась директива препроцессора #define. В С++ для символьного представления константы рекомендуется использовать объявление переменной с начальным значением и ключевым словом const:
const <тип> <имя переменной>=<начальное значение>;
Например,
const int n=10;

Область видимости константы такая же, как у обычной переменной. С помощью ключевого слова const можно объявить указатель на константу

const <тип> *<имя переменной>;
Например,
*const int *m;*
m=&n;
m - указатель на константу типа int.

Еще одна возможность const состоит в возможности создавать постоянный указатель на величину указанного типа
<тип> *const <имя переменной>=<значение>;
Например,
int i;
*int *const ptri=&i;*

Операции. В C++ введены следующие новые операции для работы с компонентами класса:

:: - операция доступа к контексту или разрешения контекста;
.* и ->* - операции обращения через указатель к компоненте класса;
new - динамическое выделение памяти объектам;
delelte - освобождение памяти, выделенной операцией new.

Операция :: доступа к контексту позволяет обращаться к глобальным переменным, когда они скрыты объявлением локального имени, например :

```
int  x = 5;                 // объявление глобальной переменной x
main (void)
{
    int  x = 3;             // объявление локальной переменной,
                            // скрывающей глобальную x
    ::x++;                  // изменение значения глобальной x
    ……….
}
```

В C++ предусмотрены специальные операции new и delete для динамического выделения и освобождения памяти объектам в свободной области памяти (куче). Эти операции могут использоваться для выделения памяти переменным и массивам, аналогично функциям malloc() и free(). Формат операций:

<указатель> = new <тип>;
или
<указатель> = new <тип> (<инициализирующее_значение>);
delete <указатель>;
где указатель - переменная типа указатель на заданный тип;
тип - тип переменной, которой выделяется память;
инициализирующее_значение - константа, определяющая начальное значение переменной.

Операция new возвращает указатель на блок в свободной области памяти размером sizeof (<тип>). Если выделить память не удалось, то возвращается нулевой указатель. Созданный объект будет существовать в памяти до тех пор, пока память не будет освобождена с помощью операции delete или до окончания работы программы.

Например :

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

```
#include <stdlib.h>
 main (void)
{
    int *pi;
    if ( !(pi = new int (5)) )   // выделение памяти
    {                            // и инициализация
       printf("Не хватает памяти для pi\n");
       exit(1);
     }
    printf ("pi= %d\n",*pi);
    delete pi;                   // освобождение памяти
 }
```

В случае успеха будет выведено *5*, в противном случае появляется строка:
Не хватает памяти для pi

Операция new отличается от функции malloc(), во-первых, тем, что операция new автоматически вычисляет размер типа и нет необходимости явно указывать операцию sizeof и, во-вторых, операция new позволяет инициализировать вновь создаваемый объект (в отличие от функции calloc(), которая только обнуляет память).

Важное замечание: если память объекту в программе отводилась с помощью операции new, то освобождаться она должна только с помощью операции delete (и ни в коем случае не функцией free() !). Нельзя также применять операцию delete для освобождения памяти, ранее выделенной функциями malloc() или calloc().

Ввод-вывод. В C++, как и в C, нет встроенных в язык средств ввода-вывода. В C для этих целей используется стандартная библиотека stdio.h, но она имеет несколько недостатков:

- функции ввода-вывода сложны в употреблении, что приводит к частым ошибкам;
- затруднителен ввод-вывод абстрактных типов данных (например, структур - struct).

В C++ разработана новая библиотека ввода-вывода (iostream.h), написанная на C++, и использующая концепцию объектно-ориентированного программирования. Библиотека iostream.h определяет три стандартных потока:
cin стандартный входной поток (stdin в C)
cout стандартный выходной поток (stdout в C)
cerr стандартный поток вывода сообщений об ошибках (stderr в C)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:
>> получить из входного потока
<< поместить в выходной поток

Ввод значений переменной может быть осуществлен следующим способом:
cin >> идентификатор;

По этому выражению из входного потока читается последовательность символов до пробельного символа, затем эта последовательность преобразуется к типу <идентификатора> и получаемое значение помещается в <идентификатор>.
Вывод информации осуществляется с использованием потока cout:
cout << значение;

Здесь <значение> преобразуется в последовательность символов и выводится в выходной поток. Возможно многократное назначение потоков:
cin >> 'переменная1' >> 'переменная2' >>...>> 'переменная n';
cout << 'значение1' << 'значение2' << ... << 'значение n';

Программа на С	Программа на С++
# include <stdio.h>	*# include <iostream.h>*
void main()	*void main()*
{ int n;	*{ int n;*
char str[80];	*char str[80];*
printf ("Ввести число\n");	*cout << "Ввести число\n";*
scanf("%d",&n);	*cin >> n;*
printf ("Ввести строку\n");	*cout << "Ввести строку\n";*
gets(str);	*cin >> str;*
printf("n=%d\n",n);	*cout << "n="<<n<<endl;*
printf("str=%s\n",str);	*cout << "str="<<str <<endl;*
}	*}*

Позднее мы еще вернемся к рассмотрению библиотеки ввода-вывода языка С++.

Ссылки. В С++ введен новый тип данных - ссылка. Ссылка позволяет определять альтернативное имя переменной. Объявляются ссылки добавлением символа & перед именем переменной. Формат объявления ссылки:

<тип> &<имя_ссылки> = <инициализатор>;

Ссылку нельзя объявить без инициализатора, т.е. некоторого объекта, связанного с сылкой, например :

```
main (void)
 {
      int  x = 3;                    // объявление переменной x
      int  &rx = x;                  // объявление ссылки на переменную x
     ……..
      rx = 5;                         // изменяется значение x
     ……..
  }
```

Ссылка может использоваться везде, где используется переменная, на которую она указывает - от нее можно брать адрес, как от переменной, и т.д. В данном примере сылка rx используется как псевдоним переменной x и всякое обращение к rx равносильно обращению к x.

Ссылки могут также указывать на константу. В этом случае компилятор создает временный объект, инициализированный значением константы :

```
main (void)
 {
    int  &ry = 7;                  // создается временный объект с именем ry
                                   // и значением 7
    ry = 8;                       // изменяется значение ry
    ……..
  }
```

При новой инициализации временного объекта адрес его не изменяется. Необходимость создания временного объекта при инициализации ссылки константой вызвана тем, что большинство компиляторов оптимизируют код, создавая только одну копию одинаковых констант. Если допустить установку ссылки на константу, а не на временный объект, это может вызвать изменение констант в других частях программы, например:

```
int &rz = 1;
rz++;
int y = rz + 1;
```

вызовет инициализацию y значением 2 вместо 3.

Временный объект также создается, когда инициализатор представляет собой объект нессылочного типа, например:

```
main (void)
{
   float  f = 1.5;                  // переменная типа float
   int    &rf = f;                  // создается временный объект rf
   ……
   rf = 2;                          // изменяется только rf, а f остается
                                    // без изменения
   …….
}
```

В данном примере создается временный объект rf, поскольку тип float инициализатора f не совпадает с типом ссылки int, т.е. значение ссылки rf определяется инициализатором нессылочного типа.

Отметим некоторые особенности ссылок:

1) ссылки не являются указателями;
2) нельзя ссылаться на битовые поля, так как отдельные биты не имеют адресов;
3) нельзя создать массив ссылок :

```
int  &rm[5];       // недопустимо
```

4) можно образовывать ссылку на ссылку, при этом временный объект не создается :

```
int  &rx = 5;
int  &ry = rx;
```

5) можно также устанавливать указатель на ссылку:

```
int  &rx = 5;
int  *p = &rx;
```

Основное назначение ссылок - это передача аргументов функции "по ссылке", а не "по значению". Другими словами ссылки в аргументах функций C++ аналогичны параметру VAR в языке Pascal. При этом нет необходимости передавать функции в качестве аргументов адреса фактических параметров, а достаточно объявить формальные параметры функции как ссылки на соответствующий тип. Для примера рассмотрим функцию swap(), меняющую местами значения своих аргументов :

```
#include <stdio.h>
void swap(int &a, int &b)
{
   int c;
   c = a;
   a = b;
   b = c;
}
```

```
main(void)
{
    int i = 5, j = 7;
    printf("Начальное значение: i = %d  j = %d\n", i, j);
    swap(i,j);
    printf("После обмена: i = %d  j = %d\n", i, j);
 }
```

Ссылки в качестве аргументов функции также полезны для сокращения области локальной памяти программы при передаче больших параметров. В этом случае функции будет передаваться не сам параметр, а ссылка на него. Чтобы предотвратить возможное изменение параметра в теле функции, он может быть описан со спецификатором const, например:

```
void func(const  large& arg)   // здесь  large - некоторый
                               // пользовательский тип большого
                               // объема
{
   . . . . . .          // значение  arg  не изменяется
}
```

Кроме того, в С++ функции могут не только принимать ссылку в качестве аргумента, но и возвращать ссылку на переменную. Выражение вызова такой функции может появиться в любой части операции присваивания. При этом необходимо учитывать, что

- если возвращаемое значение - указатель, то нельзя оператором return возвращать адрес локальной переменной.
- Если возвращаемое значение - ссылка, то нельзя оператором return возвращать локальную переменную. (Так как после выхода из функции переменная не существует, и мы получим повисшую ссылку).

Таким образом, использовать ссылки нужно крайне осторожно. Чаще всего потребность в ссылках возникает при перегрузке операций.

Функции. Прототипы функций. Определение функции в программе выглядит следующим образом:

<заголовок_функции>
{
<тело функции>
}

Заголовок функции имеет следующий вид:

<тип_функции> <имя_функции> (<спецификация_формальных_параметров>)

Если функция не возвращает значения, то ее тип void. Если тип возвращаемого значения не указан, то по умолчанию используется тип int.

При обращении к функции, формальные параметры заменяются фактическими, причем соблюдается строгое соответствие параметров по типам. В отличие от своего предшественника - языка С С++ не предусматривает автоматического преобразования в тех случаях, когда фактические параметры не совпадают по типам с соответствующими им формальными параметрами. Говорят, что язык С++ обеспечивает <строгий контроль типов>. В связи с этой особенностью языка С++ проверка соответствия типов формальных и фактических параметров выполняется на этапе компиляции.

Строгое согласование по типам между формальными и фактическими параметрами требует, чтобы в модуле до первого обращения к функции было помещено либо ее определение, либо ее описание (прототип), содержащее сведения о ее типе, о типе результата (то есть возвращаемого значения) и о типах всех параметров. Именно наличие такого прототипа либо

полного определения позволяет компилятору выполнять контроль соответствия типов параметров. Прототип (описание) функции может внешне почти полностью совпадать с заголовком ее определения:
<тип_функции> <имя_функции> (<спецификация_формальных_параметров>);

Основное различие - точка с запятой в конце описания (прототипа). Второе отличие - необязательность имен формальных параметров в прототипе даже тогда, когда они есть в заголовке определения функции.

Функции. Значения формальных параметров по умолчанию. Спецификация формальных параметров - это либо пусто, либо void, либо список спецификаций отдельных параметров, в конце которого может быть поставлено многоточие. Спецификация каждого параметра в определении функции имеет вид:
<тип> <имя_параметра>
<тип> <имя_параметра> = <значение_по_умолчанию>
Как следует из формата, для параметра может быть задано (а может отсутствовать) умалчиваемое значение. Это значение используется в том случае, если при обращении к функции соответствующий параметр опущен. При задании начальных (умалчиваемых) значений должно соблюдаться следующее соглашение. Если параметр имеет значение по умолчанию, то все параметры, специфицированные справа от него, также должны иметь начальные значения.
Пусть нужно вычислить n^k, где k чаще всего равно 2.

```
int pow(int n, int k=2)    // по умолчанию k=2
{
     if( k== 2) return( n*n );
      else return( pow( n, k-1 )*n );
}
```

Вызывать эту функции можно двумя способами:

```
t = pow(i+3);
q = pow(i+3, 5);
```

Значение по умолчанию может быть задано либо при объявлении функции, либо при определении функции, но только один раз.

Функции. Перегрузка функций. Цель перегрузки функций состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с разными по типам и количеству фактическими параметрами. Например, может потребоваться функция, возвращающая максимальное значение элементов одномерного массива, передаваемого ей в качестве параметра. Массивы, использованные как фактические параметры, могут содержать элементы разных типов, но пользователь функции не должен беспокоиться о типе результата. Функция всегда должна возвращать значение того же типа, что и тип массива - фактического параметра. Для обеспечения перегрузки функций необходимо для каждого имени определить, сколько разных функций связано с ним, т.е. сколько вариантов сигнатур допустимы при обращении к ним. Предположим, что функция выбора максимального значения элемента из массива должна работать для массивов типа int, long, float, double. В этом случае придется написать четыре разных варианта функции с одним и тем же именем. Распознавание перегруженных функций при вызове выполняется по их сигнатурам. Перегруженные функции, поэтому должны иметь одинаковые имена, но спецификации их параметров должны различаться по количеству и (или) по типам, и (или) по расположению. При использовании перегруженных функций нужно с осторожностью задавать начальные значения их параметров.

Функции. Встраиваемые функции. В базовом языке С директива препроцессора #define позволяла использовать макроопределения для записи вызова небольших часто используемых конструкций. Некорректная запись макроопределения может приводить к ошибкам, которые очень трудно найти. Макроопределения не позволяют определять локальные переменные и не

выполняют проверки и преобразования аргументов. Если вместо макроопределения использовать функцию, то это удлиняет объектный код и увеличивает время выполнения программы. Кроме того, при работе с макроопределениями необходимо тщательно проверять раскрытия макросов:

```
#define SUMMA(a, b) a + b
rez = SUMMA(x, y)*10;
```

После работы препроцессора получим:

```
rez = x + y*10;
```

В C++ для определения функции, которая должна встраиваться как макроопределение используется ключевое слово inline. Вызов такой функции приводит к встраиванию кода inline-функции в вызывающую программу. Определение такой функции может выглядеть следующим образом:

```
inline double SUMMA(double a, double b)
{
  return(a + b);
}
```

При вызове этой функции

```
rez = SUMMA(x,y)*10;
```

будет получен следующий результат:

```
rez=(x+y)*10.
```

При определении и использовании встраиваемых функций необходимо придерживаться следующих правил:

1. определение и объявление функций должны быть совмещены и располагаться перед первым вызовом встраиваемой функции.
2. Имеет смысл определять inline только очень небольшие функции.
3. Различные компиляторы накладывают ограничения на сложность встраиваемых функций. Компилятор сам решает, может ли функция быть встраиваемой. Если функция не может быть встраиваемой, компилятор рассматривает ее как обычную функцию.

Таким образом, использование ключевого слова inline для встраиваемых функций и ключевого слова const для определения констант позволяют практически исключить директиву препроцессора #define из употребления.

Лекция 3. Абстрактные типы данных. Обращение к компонентам класса. Статические члены. Защита элементов класса и атрибуты доступа. Конструкторы и деструкторы.

Объявление класса. *Синтаксически классы в C++ подобны структурам, компонентами которых помимо данных могут являться также и функции. При объявлении классов используется ключевое слово class :*

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

```
class Array                          // имя класса
{
  public:                            // спецификатор степени доступа

 // компоненты-данные класса
     int  *m;                        //указатель на массив целых чисел
     int  size;                      // размерность массива

 // компоненты-функции класса
 Array(int n = 0);                   // описание конструктора
 ~Array() { delete m; }              // определение деструктора
  int  get(int i);                   // извлечение элемента массива
  void print(char *);                // функция вывода
};

Array:: Array(int n)          // определение конструктора
{
  size = n;
   m = new int[size];
  for (int i=0; i<size; m[i++]=0);
}

int Array:: get(int i)         //определение функции get()
{
     if (i < 0 || i >= size)
     {
        cout << "Выход за границы массива\n";
        exit(1);
     }
    return m[i];
}

void Array:: print(char *s)          // определение printf()
{
   cout << s << ": ";
   for (int i=0; i<size; cout << get(i++) << " ");
    cout << "\n";
}

main(void)
{
   Array x(5);                       // определение объекта класса Array
   Array *py;                        // объявление указателя на объект
   py = new Array(10);               // создание нового объекта
                                     // и установка на него указателя
  // вызов компонент-функций:
   x.print("x");                     // прямой
   py->print("y");                   // косвенный
}
```

В этом примере опеределяется класс Array, который имеет две компоненты данных:

*m - указатель на массив целых чисел и
size - размерность массива.

Здесь же объявлены четыре компоненты-функции:
Array(), ~Array(), get() и print().

Компоненты-функции могут определяться как внутри класса (~Array()), так и вне его (Array() и print()). В последнем случае имени функции должно предшествовать имя класса с операцией доступа к контексту (::). Например, конструкция Array:: print() говорит о том, что далее следует описание функции print(), относящейся к классу Array.

Язык C++ допускает использование одинаковых имен компонент в различных классах. Чтобы иметь возможность доступа ко всякой компоненте произвольного класса в любой точке программы служит синтаксическая конструкция, называемая квалификационным именем, вида:

<класс>:: <компонента>

где класс - имя класса; компонента - имя компоненты класса.

Напомним, что применение операции "::" перед некоторым именем без имени класса позволяет обращаться к глобальным переменным. В данном случае использование квалификационного имени позволяет большие описания компонент-функций выносить вне тела определения класса.

Описание класса определяет специальный тип данных, в котором наряду с собственно данными описываются также и функции для работы с этими данными. В последующем имя класса (Array) может использоваться подобно другим пользовательским типам для определения переменных, указателей, ссылок и т.д. Каждый конкретный экземпляр (переменная) класса называется объектом.

Обращение к компонентам класса осуществляется так же, как к компонентам структур с помощью операций прямого (.) и косвенного (->) выбора компонент:

```
Array  x(5),                // опеределение объектов
*py = new Array(10);
. . . . . .
j = x.m[i];                 // прямой выбор компоненты
k = py->m[i];               // косвенный выбор компоненты
```

Обращение к компонентам-функциям осуществляется аналогично:

```
x.print();                  // прямой выбор компоненты-функции
py->print();                // косвенный выбор компоненты-функции
```

Если обращение к компоненте осуществляется внутри тела другой компоненты, то обращение выполняется без указания объекта и, следовательно, без использования операторов прямого либо косвенного выбора, например:

```
void Array:: put(int i, int x)    // компонента-функция
{                                 // класса Array
. . . . . .
  m[i] = x;                       // обращение к компоненте
                                  // этого же класса
  print();                        // … к компоненте-функции
}
```

Встроенные функции и спецификатор inline. В программах на языке C++ большинство компонент-функций являются очень короткими, содержащими простейшие обращения к компонентам данных. В связи с этим возникает опасность низкой эффективности программы в

целом из-за частых обращений к функциям. Чтобы устранить этот недостаток в C++ предложен механизм встраиваемых функций. Компоненты-функции, определенные внутри тела класса, называются встраиваемыми. Компилятор вместо вызова таких функций в объектный модуль стремиться вставить непосредственно код этой функции. Указания компилятору на подобные действия можно сделать и для компонент-функций, описанных вне класса, с помощью спецификатора inline, например):

```
inline int Array:: get(int i)
{
   if (i < 0 || i >= size)
   . . . . . .
   return m[i];
}
```

что эквивалентно

```
class Array {
. . . . . .
 int get(int i)
 {
    if (i < 0 || i >= size)
    . . . . . .
    return m[i];
 }
. . . . . .
};
```

Указатель this. В функции - члене на данные - члены объекта, для которого она была вызвана, можно ссылаться непосредственно. Например:

```
class x {
      int m;
  public:
      int readm() { return m; }
  };

x aa;
x bb;
void f()
{
   int a = aa.readm();
   int b = bb.readm();
   // ...
}
```

В первом вызове члена readm() m относится к aa.m, а во втором - к bb.m.

Указатель на объект, для которого вызвана функция член, является скрытым параметром функции. На этот неявный параметр можно ссылаться явно как на this. В каждой функции класса x указатель this неявно описан как

```
x* this;
```

и инициализирован так, что он указывает на объект, для которого была вызвана функция член. this не может быть описан явно, так как это ключевое слово. Класс x можно эквивалентным образом описать так:

```
class x {
      int m;
  public:
      int readm() { return this->m; }
  };
```

При ссылке на члены использование this излишне. Главным образом this используется при написании функций членов, которые манипулируют непосредственно указателями.

Статические члены: функции и данные. Иногда необходимо реализовать класс таким образом, чтобы все объекты этого типа могли совместно использовать (разделять) некоторые данные. Предпочтительно, чтобы такие разделяемые данные были описаны как часть класса. Типичные случаи: требуется контроль общего количества объектов класса или одновременный доступ ко всем объектам или части их, разделение объектами общих ресурсов. Тогда в определение класса могут быть введены (посредством ключевого слова static) статические элементы - переменные. Ключевое слово static может быть использовано при объявлении членов-данных и функций-членов. Такие члены классов называются статическими и, независимо от количества объектов данного класса, существует только одна копия статического элемента, поэтому к нему можно обращаться с помощью оператора разрешения контекста и имени класса

<имя_класса>::<имя_элемента>

Если x - статическое данное-член класса cl, то на него можно ссылаться cl::x, и при этом не имеет значения количество объектов класса cl. Аналогично можно обращаться к статической функции-члену:

```
class X
{
  public:
       int n;

        void  memfunc( int n );
        static void statfunc(int n, X *ptr);       // статическая функция
.....................
};

void prog()
{
   X obj;

   obj.memfunc(20)       // вызов для обычной функции
   X :: func(10, &obj);   // ... для статической функции
}
```

Статические данные-члены класса можно рассматривать как глобальную переменную класса. Но в отличии от обычных глобальных переменных на статические члены распространяются правила видимости private и public. Поместив статическую переменную в часть private, можно ограничить ее использование. Объявление статического члена в объявлении класса не является определением, то есть это объявление статического члена не обеспечивает распределения памяти и инициализацию. Статические члены-данные нельзя инициализировать в теле класса, а

так же в функциях-членах. Статические члены должны инициализироваться аналогично глобальным переменным в области видимости файла:

```
class example
{
public:
 static int mask;
}
int example::mask = 0; // определение и инициализация
```

Защита компонент классов. Степень доступа компонент класса определяется с помощью меток (спецификаторов): public (общий), protected (защищенный) и private (скрытый) . Каждая метка определяет атрибут доступа на последующие описания компонент до тех пор, пока не встретиться другая метка. По умолчанию принимается атрибут private, например:

```
class X
{
      int a;
      char   ch;              // a, ch - private по умолчанию

   protected:
       float  f;
       double d;              // f, d - protected

   public:
       void   func(void);     // func - public
};
```

Здесь a и ch имеют по умолчанию атрибут доступа private, f и d - protected и func() - public. Все компоненты некоторого класса доступны любой компоненте-функции этого же класса. Например, функции func() доступны как компоненты a, ch, так и f,d.

Поясним теперь значение атрибутов доступа компонент:

- public определяет, что компонента может быть использована любой функцией;
- private - компонента может быть использована только компонентами-функциями и "друзьями" данного класса;
- protected - компонента может быть использована только компонентами-функциями и "друзьями" данного класса, а также компонентами-функциями и "друзьями" производных из него классов.
-

Поскольку данные объектов предостерегают от случайных изменений, а доступ и манипулирование данными стараются осуществлять через определенные функции, то на практике компоненты-данные классов обычно по умолчанию скрываются, а компоненты-функции определяются с атрибутом public. Если предполагается некоторые компоненты-данные использовать в производных классах, то им присваивается атрибут protected.

Дружественные функции. Иногда желательно, чтобы функция - не член класса, имела доступ к скрытым элементам класса. Это противоречит принципу инкапсуляции данных, поэтому вопрос об использование таких функций спорно. Основная причина использования таких функций состоит в том, что некоторые функции нуждаются в привилегированном доступе более, чем к одному классу. Такие функции получили название дружественных. Для того, чтобы функция - не член класса имела доступ к private-членам класса, необходимо в

определение класса поместить объявление этой дружественной функции, используя ключевое слово friend. Объявление дружественной функции начинается с ключевого слова friend и должно находиться только в определении класса:

```
void func() {...}

class A
{
    //.....................
    friend void func();
     //.....................
 };
```

Дружественная функция, хотя и объявляется внутри класса (класс A в данном примере), функцией-членом не является. Поэтому не имеет значения, в какой части тела класса (private, public) она объявлена. Функция-член одного класса может быть дружественной для другого класса

```
class A
{.................
     int func();
   .................
};
class B
{.................
   friend int A :: func();
   .................
};
```

Функция-член (func) класса A является дружественной для класса B. Если все функции-члены одного класса являются дружественными функциями второго класса, то можно объявить дружественный класс

friend class имя_класса;

Например,

```
class A
{
  .................
};
class B
{.................
    friend class A;
  .................
};
```

Все функции-члены класса A будут иметь доступ к скрытым членам класса B.

Дружба классов не транзитивна: если X является другом класса Y, а Y - другом класса Z, это еще не означает, что X является другом класса Z.

Использование функций-друзей классов значительно увеличивает гибкость программ, разрабатываемых на языке С++. Они также удобны для перегрузки операций, например, только с помощью функций друзей можно перегрузить операции ввода-вывода .

Структуры и объединения. В С++ структуры (struct) и объединения (union) рассматриваютя как классы с умалчиваемым атрибутом доступа public, причем для объединений другие атрибуты запрещены, например:

```
struct X
{
    int        a;
    char   ch;                    // a и ch по умолчанию - public
private:
    float  f;
    double d;                     // f и d  - private
protected:
    void   func(void);            // func() - protected
};
```

Поскольку компоненты структур по умолчанию имеют атрибут public, то часто классы, имеющие только общие компоненты, описываются в виде структур. Например, описание

```
class X
{
public:
    int k;
    void funcX(int);
};
```

эквивалентно

```
struct X
{
    int k;
    void funcX(int);
};
```

Лекция 4. Конструкторы и деструкторы. Инициализация объектов. Автоматические, динамические и статические объекты.

Конструкторы и деструкторы. Среди компонент-функций класса важное место занимают конструкторы и деструкторы. Конструкторы определяют как объект создается и инициализируется, а деструктор - как объект разрушается. Если конструкторы или деструктор класса не определены явно, C++ генерирует их самостоятельно. При определении и разрушении объектов компилятор осуществляет вызов конструкторов и деструкторов автоматически. Имя конструктора совпадает с именем класса, а для деструктора спереди добавляется символ '~', например:

```
class X
{
  int a;
 public:
   X (int i=0)  {a = i; }       // конструктор класса X
  ~X()  {   }                   // деструктор класса X
};
```

Отметим некоторые особенности конструкторов и деструкторов:
1) они не имеют возвращаемого значения (даже void);
2) они не наследуются производным классом, хотя и могут быть вызваны из него;
3) нельзя работать с их адресами;
4) к конструкторам нельзя обращаться как к обычной функции, однако можно вызвать деструктор, указав полностью квалификационное имя :

```
main(void)
{
   X *p = new X(3);
    p -> X:: ~X();                  // допустимый вызов деструктора
   p -> X:: X();                    // недопустимый вызов конструктора
}
```

Автоматические, динамические и статические объекты. Конструктор вызывается всякий раз при создании объекта, а деструктор - при его разрушении. Поэтому важно знать какие объекты когда создаются и когда разрушаются. Объект может быть:

- автоматическим, когда он определяется в теле функции, уничтожается при выходе из блока;
- статическим, создается один раз при запуске программы и уничтожается при ее завершении;
- динамическим, создается в свободной области памяти с помощью операции new, уничтожается операцией delete.

Каждый конструктор начинает свою работу с проверки указателя this для определения находится объект в работе или нет. Если this равен null, то объект не в работе и конструктор вызывает new для выделения памяти объекту.

Статические объекты создаются путем вызова своих конструкторов до начала работы функции main(), автоматические - при входе в блок, в котором они определены. Компоненты базового класса образуются при создании объектов производного класса. Таким образом, зная

момент создания объектов, программист может планировать действия, выполняемые даже до вызова функции main()!

Конструктор по умолчанию. Конструкторы, в отличие от деструкторов, могут принимать аргументы. Конструкторы, не имеющие параметров, называются конструкторами по умолчанию. Рассмотрим пример класса String, определяющего строку символов :

```
class String
{
    char  *str;
  public:
     String()            // первый конструктор без аргументов
     {
        str = new char[1]; *str = 0;
     }
     String (char *);     // объявление второго конструктора
     . . . . . .
};

// определение второго конструктора
String:: String (char *s)
{
   str = new char[strlen(s)+1];
   strcpy(str,s);
}

void main( void )
{
    String  str1;
    String  str2("abc");
    .....
}
```

Первый конструктор в этом примере определяет пустую строку символов длиной 1, содержащую единственный символ конца строки. Второй конструктор позволяет создавать строку определенной длины и инициализирует ее константой или другой строкой.

Если конструктор класса не задан, компилятор автоматически формирует конструктор по умолчанию (без аргументов). Конструкторы, как и любые функции языка C++, могут перегружаться (полиморфизм конструкторов). В случае определения в одном классе нескольких конструкторов вызов конкретного конструктора определяется компилятором из контекста вызова. Поэтому конструкторы должны отличаться либо количеством, либо типом своих аргументов.

Существует опасность неоднозначности вызова конструктора, когда используется передача аргументов по умолчанию, например:

```
class X
{
  . . . . . .
  public:
      X();                // конструктор по умолчанию
      X(int i = 0);       //  конструктор с параметром
   . . . . . .
};
```

```
void main( void )
{
    X obj1;
   ... . . . .
}
```

Здесь явная неоднозначность, поскольку не ясно какой конструктор будет вызван: первый без аргументов или второй с аргументом i=0 ?

Конструктор копирования X(X &). В программах на языке С++ часто возникает необходимость копирования одного объекта в другой. Например, явное копирование объектов выполняется при использовании операции присваивания, а неявное копирование осуществляется, когда объект передается функции в качестве параметра, или возвращаемое функцией значение представляет собой некоторый объект. Если пользователем не определена процедура копирования, то компилятор осуществляет побитное копирование объектов:

```
class complex
{
   . . . . . .
    complex (double r=0, double i=0)        // конструктор инициализации
     {
        re = x.re;
        im = x.im;
     }
  . . . . . .
};

main(void)
{
  complex    x(2,1),         // инициализация x списком
             y = x;          // инициализация y
                            // побитным копированием
……
}
```

Однако побитное копирование объектов, автоматически выполняемое компилятором, подходит далеко не для всех объектов. Рассмотрим класс String строк символов :

```
class String
{
     char *str;
   public:
      String( char* s = "\0" )
      {
          str = new char[strlen(s)+1];
          strcpy(str,s);
       }
      ~String()
        {
          delete str;
        }
```

```
    void print(char *s)
    {
        cout << s << ": " << str << "\n";
    }
};
```

Выполнение следующей функции может привести к неприятностям:

```
void f()
{
    String  s1("abc"),
            s2 = s1;

    s1.print("s1");
    s2.print("s2");
}
```

Здесь определяется строка s1 и инициализируется константой "abc". Затем образуется вторая строка s2 нулевой длины. В результате инициализации, выполняемой компилятором побитно, значение указателя *str строки s2 станет равным указателю строки s1. При выходе из функции дважды будет вызван деструктор для s1 и s2, а поскольку указатели *str обеих объектов показывают на одну и ту же область памяти, то результат такого освобождения непредсказуем.

В подобных ситуациях пользователь должен сам определять способы копирования объектов. Отметим различие между присваиванием и инициализацией объекта. Присваивание, как правило, предполагает, что память объекту уже выделена и необходимо поместить туда некоторое значение. При инициализации в памяти образуется новый объект и определяется его значение. Здесь мы рассмотрим инициализацию объекта значением другого объекта, осуществляемую с помощью конструкторов копирования.

Конструктором копирования называют конструктор, который в качестве аргумента принимает ссылку на собственный класс. По этой причине для обозначения конструкторов копирования часто используется аббревиатура X(X&).

Для класса String конструктор копирования может быть определен следующим образом:

```
class String
{
   . . . . . .
    String(String& s)
    {
      str = new char[strlen(s.str)+1];
      strcpy(str,s.str);
    }
  ……….
};
```

В этом случае выполнение функции f() приведет к вызову двух конструкторов: для s1 - String(char*), а для s2 - String(String&). Причем конструктор для s2 предварительно отводит память нужного размера, куда и выполняется копирование строки s1. В результате каждый из указателей *str будет адресовать свою область памяти. Вызов деструкторов при завершении функции f() выполнится корректно.

Напомним, что конструкторы копирования вызываются компилятором неявно, когда объект является параметром функции или возвращаемым функцией значением, например:

```
main()
{
   String      s = "abc";
    . . . . ..
    s = f(s);
    . . . . ..
}
```

В данном примере вначале вызывается конструктор String(char*) для инициализации s строкой "abc". Затем для копирования значения s в параметр x функции f() вызывается конструктор String(String&). Для возврата этой копии из f() требуется еще один вызов конструктора String(String&), но на этот раз инициализируется временная переменная, которая затем присваивается s. Подобные временные переменные, автоматически формируемые компилятором, уничтожаются путем вызова деструктора при первой возможности. В нашем примере деструктор вызывается трижды перед завершением функции main().

Конструкторы копирования кроме аргумента X& могут также содержать и другие аргументы, однако аргумент X& должен быть первым, а для остальных аргументов обязательно должны быть определены их умалчиваемые значения.

Деструкторы. Противоположностью конструкторов являются деструкторы. Деструкторы вызываются для освобождения членов объекта до разрушения самого объекта. Их главное назначение - освободить память, отводимую объекту и, возможно, выполнить некоторые другие действия.

Деструкторы не могут иметь параметров. Они вызываются автоматически при выходе переменной из объявленного контекста: для локальных переменных - когда перестает быть активным блок, в котором объявлена переменная; для глобальных переменных - при выходе из процедуры main(). Однако, это правило, не относится к указателям на объект: при выходе указателя объекта за пределы контекста автоматический вызов деструктора не производится. Для разрушения объекта в этом случае следует пользоваться операцией delete, например:

```
class complex
{
      double *re, *im;
   public:
      complex(double r=0, double i=0)
       {
          re = new double(r);
          im = new double(i);
        }

       ~complex()
         {
            delete re;
            delete im;
         }
       void print(void) { ……}

};

void main()
{
```

```
    complex  a(1.0);                          // определение первого числа
    complex  *pb = new complex(2.0);          // второго
      {
         complex c(3.0);                      // третьего
         complex  *pd = new complex(4.0); // четвертого
          c.print();
          pd->print();
          delete pd;                          // косвенный вызов деструктора для pd
      }                                       // неявный вызов деструктора для c
     a.print();
     pb->print();
     delete pb;                               // косвенный вызов деструктора для pb
}                                            // неявный вызов деструктора для a
```

Здесь правило простое: если в программе объект образован с помощью операции new , то занимаемая им память должна освобождаться явным указанием операции delete.

При освобождении памяти, занимаемой объектами классов, следует обращать внимание на использование библиотечных функций abort() и exit(). Дело в том, что при выполнении функции abort() никакие деструкторы классов не вызываются, а при выполнении exit() вызываются деструкторы только глобальных переменных.

Имеется также два способа явного вызова деструктора объекта: прямо, задавая полное квалификационное имя деструктора, и косвенно, через операцию delete.

В качестве примера рассмотрим класс, определяющий двумерный массив целых чисел как массив указателей на одномерные массивы :

```
class ArrayTwo
{
     int  **m;
     int  nstr, ncol;
   public:
     ArrayTwo (int n_str, int n_col); // объявление конструктора
     ~ArrayTwo ();                    // объявление деструктора

     void print(char *);
};

// определение конструктора
ArrayTwo:: ArrayTwo (int n_str, int n_col)
{
    nstr = n_str;
    ncol = n_col;
    // выделение памяти массиву указателей
    m = (int **) calloc (n_str, sizeof (int *));
    // выделение памяти одномерным массивам
    for (int i=0; i < n_str; i++)
        m[i] = (int *) calloc (n_col, sizeof (int));
}

ArrayTwo:: ~ArrayTwo ()
{
    // освобождение памяти от одномерных массивов
```

```
    for (int i=0; i < n_str; i++)
          free (m[i]);
    // освобождение памяти от массива указателей
    free (m);
}

main(void)
{
     ArrayTwo     a(2,4),  b(6,3);     // определение объектов

      a.print("a");
      a.ArrayTwo::~ArrayTwo();       // явный вызов деструктора

      b.print("b");
      b.ArrayTwo::~ArrayTwo();      // явный вызов деструктора
}
```

В данном примере нельзя использовать деструктор, состоящий из одного вызова функции free(m), так как в этом случае память останется занятой одномерными массивами и доступ к ним будет потерян.

Может возникнуть вопрос: когда в программе необходимо определять деструктор класса явно и когда можно положиться на деструктор, формируемый компилятором автоматически? Здесь ответ таков: если размеры компонент-данных класса заранее известны и класс не содержит в качестве компонент данных указатели, которым в конструкторах выделяются блоки памяти, то можно принять деструктор по умолчанию. Во всех остальных случаях в программе необходимо явно определять деструктор класса, указав порядок освобождения памяти.

Лекция 5. Базовые и производные классы. Ограничение доступа. Наследование свойств и модификаторы доступа. Множественное наследование. Конструкторы базовых и производных классов.

Наследование классов и производные классы. Наследование - это механизм создания нового класса на основе уже существующего.

В языке C++ для этого служит механизм наследования. Рассмотрим более общий формат определения класса:

class <имя_класса>[:<список_базовых_классов>] <описание_компонент_класса>

Если в определении класса присутствует список базовых классов, то такой класс называется производным (derived), а классы в базовом списке - базовыми (base) классами. В исходных предпосылках ООП предполагалась возможность наследования каждым производным классом только одного базового класса, что предполагает срогую иерархию классов. В языке C++ эти возможности расширены: каждый производный класс может наследовать несколько базовых классов, задаваемых списком базовых классов. При таком подходе к наследованию можно строить классы, отражающие структуру более сложную, чем простая иерархия.

Производные классы “получают наследство” - данные и методы своих базовых классов, и, кроме того, могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически в базовом классе.

При наследовании некоторые имена методов (функций-членов) и (или) данных-членов базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа из производного класса к компонентам базового класса, имена которых повторно определены в производном, используется операция разрешения контекста '::'.

Производному классу доступны все компоненты базовых классов, как если бы это были его собственные компоненты. Исключение составляют компоненты базового класса с атрибутами доступа private. При определении производного класса можно также влиять на атрибуты доступа компонент базовых классов. Для этого перед именем базового класса записываются спецификаторы доступа public или private. Спецификатор public используется для сохранения атрибутов доступа компонент базового класса в производном классе без изменения. Спецификатор private делает недоступными компоненты базового класса для внешних функций, т.е. все компоненты базового класса с атрибутами public и protected принимают значение private в производном классе, и в следующем уровне иерархии классов не могут быть унаследованы другими производными классами. Использование спецификатора доступа protected перед именем базового класса запрещено.

Умалчиваемым значением спецификатора доступа для базовых классов считается private, а если в качестве базового класса выступает структура (struct), то умалчиваемым значением является public. Объединения не могут быть базовыми классами. Например:

```
class D : public B1, B2 {...};
```

В производном классе D компоненты класса B1 с атрибутами public и protected сохранят свое значение атрибута доступа, а компоненты класса B2 будут иметь атрибут доступа private и в последующем не смогут быть унаследованы другими производными классами.

Действие спецификаторов доступа в базовом списке можно скорректировать при помощи квалификационного имени в объявлениях производного класса, например:

```
class B
{
     int  a;    // a  по умолчанию  private
  public:
     int b, c;  // b  и  c - public
};

class D: B       // b  и  c  стали  private, поскольку
{                // класс  B  по  умолчанию private
     int d;
   public:
     B:: c;      // теперь  c  стала  public
     int e;
};
```

Напомним, что все компоненты базовых классов с атрибутами доступа private недоступны и их атрибуты доступа не могут быть скорректированы в производном классе (для нашего примера это B::a).

Доступ к компонентам базовых классов из функций производного класса осуществляется так же, как если бы это были собственные компоненты, т.е. нет никаких различий в обращении к компонентам базовых или производного класса, например:

```
class B
{
   protected:
        int a;
    pubcic:
        B() { a = 5; }
};

class D: B
{
     int b;
   public:
     D() { b = 3; }
      void print() ;
};

main(void)
{
   D d;
   d.print();
}
```

Однако может возникнуть неоднозначность, если компоненты различных базовых классов имеют одинаковые имена. В этом случае для обращения к компонентам базовых классов следует использовать полное квалификационное имя, например :

```
class  X
{
```

```
  public:
      int a;
      X() {  a = 5; }
};

class  Y
{
  public:
     double a;
     Y() { a = 10; }
};

class  D: X,Y
{
   public:
        void print(void) {…}
};

main(void)
{
   D d;
   d.print();
}
```

C++ допускает одинаковые имена компонент в производном и базовых классах, однако следует быть внимательным при их использовании. Если возникают сомнения в правильности интерпретации имени компоненты, следует употреблять полное квалификационное имя. Компилятор, встретив любое имя в функции производного класса, в первую очередь ищет соответствующую компоненту в производном классе, а затем - в базовом.

Инициализация объектов при множественном наследовании Кроме рассмотренного ранее механизма инициализации объекта в теле конструктора, например:

```
class X
{
    int  a,b;
   public:
     X(int i, int j)  { a = i; b = j; }
};
```

язык C++ предусматривает также возможность присваивания значений компонентам класса через список инициализаторов. Список инициализаторов записывается перед телом конструктора, а каждый элемент списка представляет собой имя компоненты, за которым в круглых скобках следует инициализирующий параметр. Предыдущий пример в новой концепции может быть представлен следующим образом:

```
class X
{
      int  a,b;
   public:
   // инициализация компонент с помощью
   // списка инициализаторов
    X(int i, int j) a(i), b(j) {…}
```

```
};
```

В обоих случаях определение конструктора X инициализация объекта будет работать правильно, например выражение

```
  X  x(1,2);
```

вызовет инициализацию x.a значением 1, а x.b - 2.

Эти два способа инициализации конструкторов можно комбинировать в пределах одного класса. Основное назначение второго способа (список инициализаторов) - обеспечить механизм передачи значений конструкторам базового класса из производного класса. В этом случае в список инициализаторов производного класса включаются конструкторы базовых классов (для обеспечения такой возможности последние должны быть объявлены с атрибутами public или protected). Вначале инициализируются базовые классы в порядке их объявления, затем инициализируются компоненты производного класса, также в последовательности их объявления независимо от их расположения в списке инициализации. Рассмотрим пример :

```
class B1   // первый базовый класс
{
      int x;
  public:
      B1 (int i) { x = i;}   // инициализация присваиванием
};

class B2  // второй базовый класс
{
       int y;
   public:
       B2 (int j)  y(j) { }     // инициализация списком
                                // инициализаторов
};

// производный класс
class D: public B1, public B2
{
     int  a, b;
   public:
      // комбинированная инициализация
     D(int i, int J): B1(i*3),  B2(i+j),  a(i)
         { b = j; }
      void print() {... }
};

main(void)
{
    D d(3,4);
    d.print();
}
```

Здесь конструктор класса B1 инициализируется по первому способу, а B2 - по второму. В инициализации конструктора класса D применяется комбинация этих способов:

конструкторы B1, B2 и компонента a инициализируются по первому способу, а компонента b - по второму.
Если конструктор производного класса определяется вне тела класса, то список инициализаторов записывается при его определении.

Указатели на производные классы. C++ позволяет использовать указатель на базовый класс для обращения к компонентам базового класса, например:

```
class  B          // базовый класс
{
    int x;
   ...
};

class  D: public B       // производный класс
{
  int y;
  ..
 };
 . . . . . .
 B  *p;    // указатель на базовый класс
 D   d;    // определение объекта производного
           // класса
 p = &d;   // установка указателя базового класса
           // на объект производного класса,
           // такое допустимо в C++
 . . . . . .
 i = p->x;  // разрешенный доступ к компоненте
           // базового класса
 j = p->y;  // так нельзя, поскольку  y
           // компонента производного класса,
           // а p - указатель базового класса
```

Чтобы получить доступ к компонентам производного класса через указатель на базовый класс, необходимо выполнить преобразование типа вида:

```
j = ( (D *)p ) -> y;     // теперь так можно
```

Таким образом, с помощью указателя на базовый класс можно обращаться к любым компонентам производного класса: как унаследованным из базового, так и специфичным для производного класса. Обратное утверждение неверно, т.е. нельзя с помощью указателя на производный класс обратиться к компонентам базового класса.

Отметим также, что инкремент и декремент указателя относится к его базовому классу, т.е. увеличивая указатель базового класса на единицу нельзя обратиться к следующему элементу производного класса.

Виртуальные базовые классы. В C++ запрещено множественное наследование одного и того же базового класса:

```
class  B  {...};
class  D: B, B {...};  // недопустимо
```

Однако это может произойти через косвенное наследование:

```
class  B {...};          // базовый класс
class  X: B {...};
class  Y: B {...};       // X и Y разные классы, однако они
                         // наследуют один базовый класс B
class  D: X, Y {...};    // производный от X и Y
```

В данном случае каждый объект класса D будет содержать два подобъекта класса B. Чтобы этого не происходило, класс B объявляют в классах X и Y как виртуальный со спецификатором virtual:

```
class  B {...};                 // базовый класс
class  X: virtual B {... };  // производный от B
class  Y: virtual B {...};   // производный от B
class  D: X, Y {...};        // производный от X и Y
```

Теперь объекты класса D будут содержать только один подобъект класса B.
В отношении виртуальных базовых классов в C++ имеется два ограничения:

1) виртуальные базовые классы не могут точно инициализироваться списком инициализации конструктора, поэтому их конструкторы должны быть без аргументов, либо иметь умалчиваемые значения параметров;

2) нельзя использовать указатель на виртуальный базовый класс для обращения к компонентам производного класса.

Конструкторы и деструкторы при наследовании. Базовый класс, производный класс или оба могут иметь конструкторы и/или деструкторы. Если и у базового и у производного классов есть конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы - в обратном порядке. То есть если A - базовый класс, B - производный из A, а C - производный из B (A-B-C), то при создании объекта класса C вызов конструкторов будет иметь следующий порядок: конструктор класса A - конструктор класса B - конструктор класса C. Вызов деструкторов при разрушении этого объекта произойдет в обратном порядке: деструктор класса C – деструктор класса B - деструктор класса A. Понять закономерность такого порядка не сложно, поскольку базовый класс "не знает" о существовании производного класса, любая инициализация выполняется в нем независимо от производного класса, и, возможно, становится основой для инициализации, выполняемой в производном классе. С другой стороны, поскольку базовый класс лежит в основе производного, вызов деструктора базового класса раньше деструктора производного класса привел бы к преждевременному разрушению производного класса.

Конструкторы базовых классов не наследуются производным классом, однако могут из него вызываться. При образовании объектов производного класса первыми вызываются конструкторы базовых классов в порядке их объявления. Рассмотрим следующий фрагмент:

```
class  D: X, Y {...};
. . . . . .
D  one;
```

для создания объекта one компилятором будет сформирована следующая последовательность вызовов конструкторов:

```
X();   // конструктор первого базового класса
```

Y(); // конструктор второго базового класса
D(); // конструктор производного класса

В свою очередь виртуальные классы имеют приоритет при вызове конструкторов базовых классов, например для следующего фрагмента

```
class  B {...};
class  X: virtual B {...};
class  Y: virtual B {...};
class  D: X, Y {...};
. . . . . .
D one;
```

будет сформирована последовательность вызовов:

B(); // конструктор виртуального базового класса
X();
Y();
D();

В случае множественных виртуальных базовых классов конструкторы запускаются в порядке их объявления, например:

```
class  B1 {...};
class  B2  {...};
class  X: B2, virtual B1 { ...};
class  Y: B2, virtual B1 {...};
class  D: X, virtual Y {...};
. . . . . .
D one;
```

образует следующий порядок вызова конструкторов:

B1(); // конструктор общего виртуального класса
B2(); //этот конструктор необходим для запуска Y()
Y(); //конструктор виртуального базового класса Y
B2(); //этот конструктор необходим для запуска X()
X(); //конструктор базового класса X
D(); //конструктор производного класса

В этом примере первым будет запускаться конструктор B1() как самый старший виртуальный базовый класс. Затем должен быть сформирован конструктор виртуального базового класса Y, но он содержит не виртуальный базовый класс B2, поэтому вначале запускается конструктор B2() для класса Y. Теперь может быть запущен конструктор Y(). Вновь запускается конструктор B2(), но уже для класса X. И, наконец, последовательно запускаются конструкторы базового класса X и производного класса D.

Из этого примера можно также видеть, что конструкторы виртуальных классов (B1 и Y) запускаются только один раз, а конструкторы не виртуальных классов (B2) вызываются при создании каждого наследуемого объекта.

При разрушении объекта деструкторы вызываются аналогично конструкторам, но только в обратном порядке: вначале самый младший в иерархии деструктор, последним

вызывается деструктор базового класса. Для нашего примера вызов деструкторов будет выглядеть следующим образом:

```
~D();
~X();
~B2();
~Y();
~B2();
~B1();
```

Лекция 6. Полиморфизм. Перегрузка операций. Общие правила переопределения операций. Дружественные функции и особенности их использования для переопределения операторов.

Примером полиморфизма в языке C++ является перегрузка (overload) операций. Она позволяет манипулировать объектами классов используя обычный синтаксис языка C. Для обеспечения такой возможности в перегружаемых операциях должно сохраняться количество аргументов соответствующей операции. Например, операция деления, обозначаемая "/", в C имеет два аргумента, поэтому перегружаемая одноименная операция "/" также должна принимать два аргумента. Как правило, в качестве аргументов перегружаемых операций и возвращаемых ими значений выступают собственные классы. Так, если перегрузить операцию сложения для класса complex, то можно использовать естественный синтаксис языка C для их сложения:

```
class complex
{
    double re, im;
  public:
    . . . . . .
}
. . . . . .
complex a, b, c;
 . . . . . .
c = a + b;
```

Все перегружаемые операции имеют тот же приоритет и правила ассоциативности, что и предопределенные операции языка. По этой причине их можно использовать для построения цепочек операций, например:

x = a + b + c + d;

Отметим, что в C++ нельзя перегрузить операции . :: .* и ?. Язык C++ не позволяет отличать постфиксную и префиксную формы перегруженных операций, поэтому, например, для перегруженной операции ++ выражения

++x и x++

имеют одно и то же значение. В общем случае предполагается, что перегруженные унарные операции используются в префиксной форме, поэтому для выражения "x++" компилятор выдаст предупреждение. Нельзя также определять новые лексические символы операций.

Язык C++ обеспечивает достаточную гибкость в перегрузке операций и их наследовании. Так, все перегруженные операции базовых классов, за исключением операции присваивания, наследуются производным классом. Перегруженная операция базового класса может затем вновь перегружаться в производном классе и т.д. Операции перегружаются с помощью функций-операторов, имеющих следующий формат:

<тип> operator <символ> (<список_аргументов>) {...}

где тип - это тип возвращаемого операцией значения, а символ - символьное обозначение перегруженной операции.

Функции-операторы могут вызываться таким же образом, как и любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функции-оператора, например для комплексных чисел сокращенная запись

```
c = a + b;
```

эквивалентна вызову функции

```
c = operator+(a,b);
```

В отношении действий, выполняемых перегружаемыми операциями, C++ не накладывает никаких ограничений. Можно, например, перегрузить операцию "+" для выполнения вычитания комплексных чисел, а "-" - для их сложения, но подобное использование механизма перегрузки операций не рекомендуется с точки зрения здравого смысла.

Сложные операции C++, равносильные комбинации нескольких операций, автоматически не раскрываются и, если пользователь их не определил, остаются компилятору неизвестными. Например, если для комплексных чисел перегружены операции сложения "+" и присваивания "=", то запись вида

```
a += b;
```

совсем не означает, что будет выполняться

```
a = a + b;
```

Чтобы можно было применять операцию "+=" с комплесными числами, она должна быть соответствующим образом перегружена в классе complex.

В C++ имеются различия при перегрузке операций с помощью функций-"друзей" и функций-компонент, поэтому рассмотрим их отдельно.

Перегрузка операций с помощью функций-"друзей" Функции-операторы, являющиеся "друзьями" класса, всегда имеют число аргументов, равное числу аргументов перегружаемой операции. Для унарных операций - это один аргумент, для бинарных - два и т.д. Отметим, что с помощью функций-"друзей" в Turbo C++ нельзя перегружать операции =, (), [] и ->. В качестве примера рассмотрим перегрузку операции сложения класса complex :

```
class  complex
{
    double re, im;
  public:
    complex (double r=0, double i=0)
      { re = r; im = i; }
    void print(char *s)
      {...); }
    friend complex operator + (complex, complex);
};
 complex operator + (complex a, complex b)
  { return complex(a.re + b.re, a.im + b.im); }
```

или :

```
class  complex{...}
 complex operator + (complex a, complex b)
{
  complex c;
```

```
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
```

Теперь для сложения комплексных чисел можно использовать естественный синтаксис языка C++:

```
  complex  x(2.0,1.0), y(3.14,1.0), z;
   z = x + y;
```

Использование ссылок для перегрузки унарных операций

Если попытаться для класса complex аналогичным образом перегрузить унарную операцию инкремента следующим образом :

```
class  complex
{
. . . . . .
    public:
. . . . . .
    friend complex operator ++(complex);
};
complex operator ++(complex a)
 { return complex(a.re++, a.im++); }
```

или :

```
class  complex{...};
complex operator ++(complex a)
{
  a.re++,
  a.im++;
  return a;
}
```

то можно убедиться, что перегруженная операция работать не будет. Дело в том, что в C++ аргументы передаются по значению и, следовательно, возвращаемая величина не изменяет своего значения. Можно использовать в качестве аргумента указатель :

```
class  complex
{
. . . . . .
  public:
    friend complex operator ++ (complex *);
};
 complex operator + (complex *p)
{
   p->re++,
   p->im++;
   return *p;
}
```

Однако в этом случае компилятор выдаст ошибку, поскольку язык C++ требует, чтобы операнд операции "++" имел тип класса объекта. Выходом из этой ситуации является использование ссылок в качестве аргумента функции-оператора:

```
class  complex
{
. . . . . .
    public:
  friend complex operator ++(complex &);
};
 complex operator + (complex &r)
 {
    r.re++,
    r.im++;
    return r;
 }
```

Невозможность перегрузки унарных операций функциями "друзьями" послужило одной из причин введения ссылок в язык C++. Ссылки в качестве аргументов перегружаемых операций позволяют передавать реализующей ее функции не копии объектов, а их адреса, что позволяет изменять значения этих аргументов. Кроме того, использование ссылок в качестве аргументов для больших объектов позволяет значительно сократить объем копируемой информации, что приводит к повышению эффективности программы.

Рассмотрим теперь вопрос каким должно быть возвращаемое значение функции-оператора. Оно не может быть ссылкой на автоматическую переменную, так как после выхода из функции возвращаемое значение указывает на несуществующую переменную. Оно также не может быть статической переменной. Допускается ссылка на переменную в "куче", однако это усложняет программирование. Лучшим решением с точки зрения простоты реализации и эффективности считается копирование возвращаемого значения . Можно также возвращать ссылку, но только на существующий объект, например, когда эта ссылка передается функции-оператору в качестве параметра:

```
 class complex
{
 . . . . . .
    public:
 . . . . . .
    friend complex& operator ++(complex& r);
};
complex& operator ++(complex& r)
{
    r.re++;
    r.im++;
    return r;
}
```

Перегрузка операций с помощью компонент-функций. Для функций-операторов, являющихся компонентами класса, первым аргументом по умолчанию является указатель this на тот объект, к которому она относится. Поэтому число аргументов для таких операторов на единицу меньше, по сравнению с числом аргументов перегружаемой операции. Например,

для бинарных операций - один аргумент, для унарных - аргументы отсутствуют и т.д. Перегрузка операций + и ++ для класса complex, реализованная с помощью операторов-компонент имеет вид:

```
class  complex
{
 . . . . . .
   public:
     complex operator + (complex b);
     complex operator ++ (void);
};
complex complex:: operator + (complex b)
    { return complex(re + b.re, im + b.im); };
 complex complex:: operator ++()
     { return complex(re++, im++); };
```

Отметим, что при перегрузке операций с помощью компонент функций широко используется указатель this. Однако при этом следует соблюдать особую осторожность. Соблазнительной может показастья идея использовать для получения результата вместо промежуточной переменной первый аргумент, например:

```
class  complex
{
. . . . . .
   public:
      complex operator +(complex b);
      complex operator ++(void);
};
complex complex:: operator + (complex b)
{
    re += b.re;
    im += b.im;
    return *this;
};
complex complex:: operator ++ ()
{
     re++;
     im++;
     return *this;
};
```

В последнем примере операция инкремента работает правильно, а операция сложения имеет побочный эффект, увеличивая значение первого операнда.

Напомним, что с помощью функций-операторов, являющихся компонентами класса, можно перегружать операции =, (), [] и ->, которые запрещены для перегрузки функциями-"друзьями".

Может возникнуть вопрос в каких случаях при перегрузке операций следует использовать функции-"друзья", а в каких - функции-компоненты? Рассмотрим пример:

```
main(void)
{
```

```
   complex a(2.0,1.0), b(3.14,1.0), c;
   double d = 1.5;
   c = a + b;
   c = a + d;
   c = d + b;
. . . . .
}
```

Выражение *c = a + b* будет верным, поскольку a и b оба типа complex, для которого переопределена операция "+". Выражение *c = a + d* также будет верным, поскольку оно интерпретируется как *a.operator+(complex(d))* Однако выражение *c = d + b* вызовет ошибку, если операция "+" перегружена с помощью функции-компоненты. Дело в том, что d + b по определению эквивалентно *d.operator+(b)*, но d не является объектом класса и не имеет компонент! В случае же перегрузки "+" как функции-друга выражение верно, поскольку здесь будет вызван конструктор инициализации и *d + b* будет эквиваленто *complex(d) + b*

Отсюда можно сделать следующий вывод: в том случае, если предполагается неявное преобразование операндов перегружаемой операции, то реализующую ее функцию лучше сделать "другом". К таким операциям можно отнести арифметические и логические операции (+, -, || и т.д.).

Если же операция изменяет состояние объекта, а это можно сделать только для ранее созданного объекта, то реализующая операцию функция должна быть компонентой класса. К подобным операциям относятся =, *=, ++ и др.

При прочих равных условиях для перегрузки операций чаще используются функции-компоненты. Это связано с тем, что они имеют на один аргумент меньше, допускают использование неявного параметра this, не чувствительны к модификациям функций преобразования типа и, наконец, описание функции-компоненты, как правило, короче описания функции-друга.

Многократная перегрузка операций

С++ позволяет несколько раз перегружать одну и ту же операцию в пределах одного класса. Важно только, чтобы однозначно определялся вызов необходимой операции, т.е. функции-операторы должны различаться количеством или типом своих аргументов, например:

```
class String
{
  char *str;
     public:
   String (char * s = "\0")          // конструктор
   { str = new char [strlen (s) +1]; strcpy (str,s); }
    String operator + (String t)       // первая перзагрузка +
     { return String(strcat(str,t.str)); }
     String operator + (char *s)         // вторая перзагрузка +
    { return String(strcat(str,s)); }
      void print(char* s)
     { cout << s << ": " << str << "\n"; }
};
main(void)
{
   String a("Минск"), b(" - город"), c;
```

```
    c = a + b + " -герой!";
    c.print("c");
}
```

В данном примере дважды перегружена операция + для класса String. В первом случае она позволяет нам связывать объекты типа String между собой, а во втором - тип String со строкой символов. Это дает возможность в одном выражении записывать как объекты типа String, так и обычные строки типа (char *).

Перегрузка операции присваивания =

Операция присваивания перегружается обычным образом. Особенностью этой операции является то, что она не может быть унаследована производным классом. Кроме того, в C++ операцию "=" можно перегружать только с помощью функции-компоненты.
Например:

```
class  complex
{
. . . . . .
      public:
   complex& operator =(complex &);
};
complex& complex:: operator =(complex& b)
{
   re = b.re;
   im = b.im;
   return *this;
}
```

Здесь первый операнд передается через указатель this, принимает значение второго операнда, полученного в качестве аргумента, и возвращается с помощью выражения *this. Не смотря на то, что в этом примере изменяется значение первого аргумента, это не является ошибкой, поскольку его значение как раз и нужно изменить. Данный пример также демонстрирует те немногие случаи, когда в программе необходимо явно использовать указатель this.

Если для некоторого класса X операция присваивания не определена, то при необходимости она формируется компилятором C++ автоматически в виде:

```
X & X:: operator = (const X & <источник>)
{
   // покомпонентное присваивание
   return  *this;
}
```

Рассмотрим на примере класса String более сложный случай перегрузки операции "=", чем простое покомпонентное присваивание. Первый вариант перегрузки операции "=" для класса String на первый взгляд вообще не возвращает никакого значения

```
class String
{
 . . . . . .
     public:
    void operator = (Srting &t){
```

```
        if (this == &t)
            return;              // копирование в ту же строку
        delete str;              // удаляется старая строка
         str = new char[strlen(t.str)+1]; // образуется новая
         strcpy(str,t.str);          // в нее копируется значение
      }
}
```

Здесь вначале проверяется, чтобы строка не дублировала саму себя. Затем уничтожается старое значение строки, отводится память необходимого объема и туда копируется принимаемый аргумент.

Более корректной реализацией перегрузки операции "=" будет явный возврат сформированного объекта с помощью указателя this:

```
class String
{
 . . . . . .
     public:
   String& operator = (Srting &t)
   {
      if (this == &t)
           return *this;
      delete str;              // удаляется старая строка
      str = new char[strlen(t.str)+1]; // образуется новая
      strcpy(str,t.str);          // в нее копируется значение
       return *this;
      }
}
```

Если сравнить последнюю реализацию с конструктором класса String, то можно увидеть их значительное сходство. Поэтому можно воспользоваться уже готовым конструктором для формирования возвращаемого значения:

```
class String
{
 . . . . . .
     public:
  String operator = (Srting &t)
  {
    if (this == &t)  return *this;
    delete str;              // удаляется старая строка
    return String(t.str);
   }
}
```

Заметим, что здесь изменился тип возвращаемого значения, поскольку конструктор String::String() возвращает объект, а не ссылку на него.

Таким образом, для простых типов (например, complex) операцию присваивания можно не перегружать, в этом случае она формируется компилятором автоматически, осуществляя покомпонентное присваивание. Если объекты сложные и покомпонентное присваивание для них не работает, следует операцию присваивания перегружать явно. В качестве возвра-

щаемого значения перегруженной операции присваивания чаще используется ссылка на объект . Возможно в возвращаемом выражении обращаться к конструктору класса. В последнем случае возвращаемым значением будет сам объект.

Перегрузка операций [], () и ->

Прежде всего отметим, что эти операции перегружаются только с помощью компонент-функций и их нельзя перегрузить функциями- "друзьями".

Начнем с рассмотрения перегрузки операции []. Предположим, что мы хотим обращаться к действительной и мнимой частям некоторого комплексного числа h как к элементам массива, т.е. h.re соответствует h[0] и h.im соответствует h[1]. Для этого можно перегрузить операцию [] выделения элемента массива для класса complex следующим образом :

```
class  complex
{
   double re, im;
      public:
   . . . . . .
    double & operator [] (int i)
    { return *(&re + i);}
};
main(void)
{
  complex  one(1.0, 0.0);
   cout << "one.re=" << one[0] << "one.im=" << one[1] << "\n";
}
```

Перегрузка операции [] для класса X позволяет всякое выражение вида ob[i], где ob - объект класса X, интерпретировать как обращение к функции ob.operator[](i). Операция вызова функции вида:

<имя>(<список_аргументов>);

в языке C++ рассматривается как бинарная операция, где первым операндом является <имя>, а вторым - <список_аргументов>. При перегрузке операции () список аргументов вычисляется и проверяется в соответствии с обычными правилами передачи аргументов. В общем случае список аргументов может быть пуст.

Перегрузка операции () позволяет рассматривать выражение вида

ob(<список_аргументов>)

как обращение к функции

ob.operator()(<список_аргументов>).

Рассмотрим перегрузку операции () для класса Array. Иногда целесообразно считать, что индекс массива начинается не с нуля, а с единицы (подобно тому, как это реализовано в языке PL/1). Для этого перегрузим операцию () :

```
class Array
{
```

```
  . . . . . .
      public:
   int operator()(int i) { return m[i-1]; }
};
void main()
{
    Array x(5);
    cout << "x= ";
     for (int i=1; i<=5; i++)
        cout << x(i) << " ";
        cout << endl;
}
```

Операцию () обычно перегружают для операций, требующих большого числа операндов, для классов с единственно возможной операцией, а также когда некоторая операция используется особенно часто. Перегружаемая операция () не может быть статической компонентой-функцией класса.

C++ предоставляет пользователю возможность выполнять некоторую предварительную обработку до обращения к компонентам классов. С этой целью используется перегузка операций ->, * и &. Операция -> рассматривается как унарная операция и ее перегрузка позволяет выражение вида

ob->m

трактовать как

(ob.operator->())->m.

Причем функция-оператор, реализующая операцию -> должна либо возвращать указатель на объект данного класса, либо возвращать объект этого класса, т.е. ее описание для класса X должно иметь вид:

*X *operator -> () {...}*

или

X operator -> () {...}.

Рассмотрим случай, когда возвращаемым значением перегуженной операции -> является тип этого же класса:

```
class X
{
    public:
    int a;
    X(int i) { a = i; }
     X operator ->()
     {
        cout << "Доступ к компонентам класса X\n";
        return *this;
      }
};
main(void)
{
   X x(5);
   X *px = new X(10);
    cout << "a= " << x->a << "\n";  // перегруженная операция
    cout << "a= " << px->a << "\n"; // предопределенная операция
}
```

В данном примере перед обращением к компоненте в стандартный поток выводится сообщение.Наибольший интерес представляет случай, когда возвращаемым значением перегруженной операции -> класса X является указатель на некоторый другой класс Y. В этом случае операция -> вначале применяется к своему левому операнду для получения указателя p на класс Y, а затем указатель p используется как левый операнд бинарной операции -> для доступа к компоненте класса Y, например:

```
class Y
{
     public:
   int b;
   Y(int j) { b = j; }
};
class X
{
   int a;
     public:
   Y  *p;
    X(int i, int j)
   {
      a = i;
      p = new Y(j);
    }
   Y* operator ->()
   {
      cout << "Доступ к компонентам класса Y\n";
       return p;
     }
};
main(void)
{
    X x(3,5);
    cout << "b= " << x->b << "\n";     // перегруженная операция
    cout << "b= " << x.p->b << "\n";   // базовая операция
}
```

Если класс Y в свою очередь имеет перегруженную операцию ->, то p будет использован в качестве левого операнда унарной опеации -> и вся процедура повторится для класса Y.

Унарные операции * и & перегружаются аналогично, причем сохраняется соответствующая семантика между операцией -> и операциями * и &.

Перегрузка операций new и delete

Язык C++ предоставляет две возможности для перегрузки операций new и delete. Они могут быть перегружены глобально, т.е. любое обращение в программе к new и delete будет вызывать перегруженные операции, или в пределах класса. В последнем случае вызов перегруженных операций будет осуществляться только для объектов данного класса, а все другие обращения к new и delete будут вызывать переопределенные операции C++.

Глобальная перегрузка операций new и delete имеет вид:

```
void * operator  new (size_t size)
{
   // выполнение необходимых выделений памяти
   retutn p;
}
void * operator  delete (void *p)
{
   // освобождение памяти, указываемой  p
}
```

Здесь size_t - целый тип, определенный в stdlib.h; p - указатель на выделенную память.

Известно, что стандартная операция new при выделении память не обнуляет и там находится случайный "мусор". Следующий пример показывает вариант глобальной перегрузки предопределенных операций new и delete, обнуляющих память при ее выделении:

```
extern void * operator new(size_t  size)
{
    return calloc(1, size);
};
extern void operator delete(void *p)
{
    free ((char *)p);
};
main(void)
{
    int *m = new int[5];
    cout << "m= ";
    for (int i=0; i < 5; i++)
    {
       cout << m[i] << " ";
       cout << "\n";
     }
}
```

Теперь всякое обращение к операции new будет возвращать область памяти, заполненную нулями, а операция delete позволяет освободить выделенный блок с помощью функции free().

Чтобы перегрузить операции new и delete только в отношении объектов некоторого класса X, они должны быть определены в теле этого класса. Перегруженные операции new и delete будут вызываться только для объектов класса X. Для других объектов будут вызываться глобальные (либо собственные) new и delete.

Если в вызове операции new указан любой другой тип, отличный от X, то будет вызываться глобальная операция new, даже если вызов осуществляется в теле компоненты-функции класса X. Так, например, следующая попытка перегрузить операцию new для класса Array будет безуспешна, поскольку в конструкторе вызывается глобальное new :

```
class  Array
{
   int size;
   int *m;
```

```
        public:
   Array (int n)                    // конструктор
   { size = n; m = new int[size]; } // вызывается глобальная
                                    // new, которая и работает
   ~Array() { delete m; }           // деструктор
    void * operator new(size_t n);
    void operator delete(void *p);
};
 // переопределение операции new
void *Array:: operator new(size_t n)
{
    int *p;
    p = (int *) malloc(n*sizeo(int)); // выделяем и
    for (int i=0; i<n; i++)          // заполняем память
        p[i] = i;
    return (void *)p;
}
 // переопределение операции delete
 void *Array:: operator delete(void *p)
{
    free(p);
}
main(void)
{
    Array  a(5);                    // объект класса Array
    a.print();                      // вывод заполненного массива
    Array  *pb = new Array(7); // указатель на объект
     pb->print();                   // вывод массива
}
```

В качестве примера перегрузки операции new только для отдельного класса рассмотрим класс Point, определяющий точку на экране дисплея. Каждой точке соответствует блок, представляющий собой структуру из двух целых чисел. Множество таких блоков объединяется в один массив blocks, который расположен в статической области памяти. Каждое обращение к Point:: new возвращает указатель на свободный блок памяти, определяемый переменной top :

```
const max = 512;
class Point
{
   int x,y;
   static stract Block
    {   // блок из двух точек
        int xy[2];
     }  blocks[max];            // массив блоков точек
   static int top;              // счетчик блоков
        public:
    Point(int a=0, int b=0)     // конструктор
    { x=a; y=b; }
    void * operator new(size_t n)
    {
```

```
        n = n;
        if (top < max) return blocks + top++;
        else    return 0;
      }
     void print(char *s)
     {
       cout << s << "= (" << x << "," << y << ")\n";
      }
};
main(void)
{
    Point *pone = new Point(1,1);
    pone->print("one");
     Point *ptwo = new Point(2,2);
     ptwo->print("two");
}
```

Отметим, что в C++ функции-операторы new и delete некоторого класса по умолчанию имеют спецификатор static, поэтому они не могут быть виртуальными функциями.

Преобразование типа

Механизм перегрузки операций можно применять для преобразования типов данных. Функции-операторы, обеспечивающие преобразование типов, должны быть компонентами класса и имеют следующий вид:

```
operator <тип>();
```

где тип - спецификация типа, которая является результатом преобразования.

В качестве примера введем в класс String функцию-оператор для преобразования объектов класса String к типу char*:

```
class String
{
   char *str;
     public:
. . . . . .
    operator char *()
    {       // объявление оператора
            // преобразования типа
        char *p = new char[strlen(str)+1];
        strcpy(p,srt);
         return p;
     }
};
void main()
{
    String *s("Минск - город-герой!\n");
    char *t;
     t = s;                    // теперь так можно !
     cout << t;
}
```

Напомним, что преобразования типов могут выполняться в конструкторах, конструкторах копирования и в операциях присваивания. Поэтому при разработке классов сложных объектов следует предусматривать весь набор возможных преобразований типа. Для этого определяются конструктор копирования, перегружается операция присваивания и определяются необходимые преобразования типов.

Лекция 7. Виртуальные функции. Особенности разработки и использования виртуальных функций. Чистые виртуальные функции. Абстрактные классы.

Виртуальные функции. Виртуальные функции - специальный вид функций-членов класса. Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы. Для объявлениявиртуальной функции используется ключевое слово virtual. Функция-член класса может быть объявлена как виртуальная, если

- класс, содержащий виртуальную функцию, базовый в иерархии порождения;
- реализация функции зависит от класса и будет различной в каждом порожденном классе.

Другими словами, виртуальная функция - это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс. Определение того, какой экземпляр виртуальной функции вызывается по выражению вызова функции, зависит от класса объекта, адресуемого указателем или ссылкой, и осуществляется во время выполнения программы. Этот механизм называется динамическим (поздним) связыванием или разрешением типов во время выполнения.

Указатель на базовый класс может указывать либо на объект базового класса, либо на объект порожденного класса. Выбор функции-члена зависит от того, на объект какого класса при выполнении программы указывает указатель, но не от типа указателя. При отсутствии члена порожденного класса по умолчанию используется виртуальная функция базового класса.

```
// Выбор виртуальной функции
#include <iostream.h>
class X
{   protected:
  int i;
     public:
  virtual void print()  {cout << "класс X: " << i;}
};
class Y : public X
{
    public:
   void print() {cout << "класс Y: " << i;}
};
main()
{ X x;
  X *px=&x; // Указатель на базовый класс
  Y y;
   x.i=10;
   y.i=15;
   px->print(); // класс X: 10
   px=&y;
   px->print(); // класс Y: 15
```

```
}
```

В каждом случае выполняется различная версия функции print(). Выбор динамически зависит от объекта, на который ссылается указатель. В терминологии ООП <объект посылает сообщение print и выбирает свою собственную версию соответствующего метода>. Виртуальной может быть только нестатическая функция-член класса. Для порожденного класса функция автоматически становится виртуальной, поэтому ключевое слово virtual можно опустить.

Существует ограничение, связанное с применением виртуальных функций: деструкторы класса могут быть виртуальными, а конструкторы - нет.

Рассмотрим программу, которая вычисляет площади различных фигур. Каждая фигура может порождаться от базового класса figure

```
class figure
{
  protected:
    double x, y;
  public:
    virtual double area() {return(0);} // по умолчанию
};
class rectangle : public figure
{
   double height, width;
 public:
   double area() {return(height*width);}
};
class circle : public figure
{
  double r;
    public:
  double area() {return(PI*r*r);}
};
```

При такой иерархии порожденные классы соответствуют типам фигур. Вычисление площади является локальной ответственностью порожденных классов. Вычислить общую площадь фигур проекта можно следующим образом:

```
figure *p[N]; // массив указателей на базовый класс
              // элементы массива могут ссылаться на
              // различные производные классы
for(i=0; i<N; i++)
   double tot_area+=p[i]->area();
```

Код пользователя не нуждается в изменении, даже если к системе добавляются новые типы фигур.

Абстрактные базовые классы. Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придается лишь в порожденных классах. Такие функции называются чистыми виртуальными функциями.

Чистая виртуальная функция - это функция-член класса, тело которой не определено. В базовом классе такая функция записывается следующим образом:

virtual прототип функции = 0;

Например

virtual void func() = 0;

Чистая виртуальные функции используются для того, чтобы отложить решение задачи о реализации функции на более поздний срок. В терминологии ООП это называется отсроченным методом. Класс, имеющий по крайней мере одну чистую виртуальную функцию, называется абстрактным классом. Для иерархии типа полезно иметь абстрактный базовый класс. Он содержит общие свойства иерархии типа, но каждый порожденный класс реализует эти свойства по-своему.

Лекция 8. Параметризация классов и шаблоны функций. Оператор "template". Методы использования шаблонов.

Шаблоны функций.

Рассмотрим некоторую функцию, например, max(x,y), аргументы которой могут быть любого типа. Один путь решения этой задачи - перегузка функций:

```
int max(int x, int y)     {retutn (x > y) ? x : y;};
long max(long x, long y)  {retutn (x > y) ? x : y;};
. . . . . .
```

Однако это требует своего кода для каждой перегруженной функции. Общим здесь является только имя функции. Второй путь - использование макросов:

```
#define  max(x,y)  ((x>y)?x:y)
```

В этом случае не работает механизм проверки типов, что может привести к попытке сравнения несравнимых типов (например, int и struct). Далее, замена в исходном тексте макроса выполняется даже тогда, когда это не требуется, например:

```
class My_class
{
  . . . . . .
   int max(int,int);    // синтаксическая ошибка !
  . . . . . .
}
```

Решением проблемы может являться использование шаблонов (template). При таком подходе тип аргументов функции обозначается некоторой буквой. В определении функции эта буква используется в качестве типа данных. В последующем компилятор автоматически сгенерирует необходимую функцию, соответствующую типу передаваемых аргументов :

```
#include <iostream.h>
template <class T>          // определение буквы шаблона
T max(T x,T y)              // определение функции-шаблона
{
  return (x > y) ? x : y;
}
void main()
{
    int    a=3, b=5;
    float  x=2.1, y=0.15;
    long   r=754, t=3561;
    cout << "max(a,b)= "
         << max(a,b)  // генерируется max() для int
         << "max(x,y)= "
         << max(x,y)  // генерируется  max() для float
         << "max(r,t)= "
         << max(r,t)  // генерируется  max() для long
```

```
        << endl;
}
```

В приведенном примере функция-шаблон max() применима к любым типам, для которых определена операция ">". Можно отвергнуть генератором функции-шаблона для специальных типов, явно определив функцию для конкретного типа:

```
char * max(char *s, char *t)
{
  return (strcmp(s,t) > 0) ? s : t;
}
```

Теперь обращение к функции max(s,f) для строковых аргументов не будет генерировать новую функцию, а сформирует вызов необходимой функции:

```
#include <iostream.h>
#include <string.h>
template <class T>          // определение буквы шаблона
T max(T x,T y)              // определение функции-шаблона
{
  return (x > y) ? x : y;
}
char * max(char *s, char *f) // определение max для (char*)
{
 return (strcmp(s,f) > 0) ? s : f;
}
void main()
{
  int    a=3, b=5;
  float  x=2.1, y=0.15;
  long   r=754, t=3561;
  char   s[]="Минск",f[]="Москва";
  cout << "max(a,b)= "
       << max(a,b)  // генерируется max() для int
       << "max(x,y)= "
       << max(x,y)  // генерируется  max() для float
       << "max(r,t)= "
       << max(r,t)  // генерируется  max() для long
       << "max(s,f)= "
       << max(s,f)  // вызывается  max() для char*
       << endl;
}
```

При использовании функций-шаблонов необходимо внимательно следить за типом передаваемых аргументов, поскольку для функций-шаблонов предопределенные преобразования типов не выполняются, например:

```
#include <iostream.h>
template <class T>          // определение буквы шаблона
T max(T x,T y)              // определение функции-шаблона
{
```

```
  return (x > y) ? x : y;
}
void main()
{
   int  i=5;
   char c=3;
    int  x1,x2,x3,x4;
    x1=max(i,i);    // все в порядке
    x2=max(c,c);    // также все в порядке
    x3=max(i,c);    // не работает, поскольку аргументы
                    // разного типа
    x4=max(c,i);    // не работает по той же причине
   cout << "x1= " << x1 << "x2= " << x2
        << "x3= " << x3 << "x4= " << x4 << endl;
}
```

Чтобы допустить предопределенные преобразования типов при обращении к функции-шаблону, достаточно явно определить ее прототип:

```
#include <iostream.h>
template <class T>          // определение буквы шаблона
T max(T x,T y)              // определение функции-шаблона
{
  return (x > y) ? x : y;
}
int max(int,int);           // определение прототипа max()
void main()
{
   int  i=5;
   char c=3;
   int  x1,x2,x3,x4;
   x1=max(i,i);    // все в порядке
   x2=max(c,c);    // также все в порядке
   x3=max(i,c);    // теперь уже работает
   x4=max(c,i);    // теперь уже работает
  cout << "x1= " << x1 << "x2= " << x2
       << "x3= " << x3 << "x4= " << x4 << endl;
}
```

Ограничением на использование функций-шаблонов является то, что для них нельзя указывать умалчиваемые значения аргументов.

Шаблоны классов

Шаблон класса (class template), называемый также обобщением классов (generic class) или генератором классов (class generator), задает образец для определений классов. Рассмотрим шаблон класса Array, который позволяет определить классы массивов с любым типом элементов:

```
#include <iostream.h>
template <class T>      // определение буквы шаблона
class Array
```

```
{                          // определение шаблона классов Array
    T *m;              // тип массива задается шаблоном
    int size;
       public:
    Array(int n=0)
    {
       size = n;
        m = new T[size];  // тип памяти задается шаблоном
     }
    ~Array() {delete m;}
     void set(int i, T x)  { *(m+i) = x; }
     T & operator [] (int i)  { return m[i]; }
     void print(char*);
};
// определение внешней функции
 template <class T>      // шаблона классов
void Array<T>:: print(char *s)
{
    cout << s << ":";
    for (int i=0; i<size; cout << m[i++] << " ");
       cout << "\n";
};
```

Теперь можно определять объекты различных классов Array, задавая в качестве параметра шаблона конкретный тип элементов массива:

```
void main()
{
   Array<int>   mi(5);  // определение класса Array для int
   Array<float> mf(3);  // определение класса Array для float
   for (int i; i<5; i++)   mi.set(i,i);
      mi.print("mi");
   for (i; i<3; i++)       mf.set(i,(float)i);
      mf.print("mf");
}
```

В общем случае определение шаблона классов имеет вид:

```
template <список_аргументов>
class имя_класса
{
   // определение компонент класса
};
```

Внешние функции шаблона классов определяются следующим образом:

```
template <список_аргументов>
имя_класса<список_аргументов>:: имя_функции() {...};
```

Объекты конкретного класса задаются следующим образом:

имя_класса<параметры_шаблона> список_объектов;

Аргументы шаблона классов могут быть двух видов: типовые и не типовые. Типовым аргументам предшествует ключевое слово class, они, как правило, обозначаются буквой и представляют параметр типа, т.е. изменяемые типы данных. Не типовые аргументы шаблона классов аналогичны параметрам конструктора класса, для них можно задавать умалчиваемые значения параметров, определяемые константными выражениями. При этом каждый генерируемый класс будет получать собственную копию статических компонент.

В следующем примере тип элементов массива и размер массива передаются в качестве параметров шаблона классов:

```
#include <iostream.h>
template <class T, int n=1> // определение шаблона
class Array
{                                  // определение шаблона классов Array
    T *m;                          // тип массива задается шаблоном
     int size;
          public:
     Array()
     {
        size = n;         // размер массива задается параметром
                          // шаблона
        m = new T[size];  // тип памяти задается шаблоном
      }
      ~Array() {delete m;}
       T & operator [] (int i)  {return m[i];}
       void print(char*);
};
void Array<T>:: print(char *s)
{
   cout << s << ":";
   for (int i=0; i<size; cout << m[i++] << " ");
        cout << "\n";
};
void main()
{
   Array<int>   mi(5);  // определение класса Array для int
   Array<float> mf(3);  // определение класса Array для float
   for (int i; i<5; mi[i] = i++);
      mi.print("mi");
   for (i; i<5; mf[i] = (float)i++);
      mf.print("mf");
}
```

Если автоматическое определение класса пользователя не устраивает, можно явно определить шаблон класса подобно тому, как это делается для шаблонов функций:

```
#include <iostream.h>
#include <string.h>
template <class  T>  // определение  буквы  шаблона
class  Array
{                    //  определение   шаблона   классов   Array
```

```
   . . . . . .
};
class Array<char*>
{                        // определение  класса  Array  для  строк
    char    **m;         //    символов
     int size;
          public:
     Array<char*>(int n=0);
     ~Array();
      void set(int i, char *s) { strcpy(m[i],s);  }
      void print(char*);
};
Array<char*>:: Array(int n)
{
    size = n;
     m = new char*[size];
    for (int i=0; i<size; i++)  m[i] =  new  char[81];
}
Array<char*>:: ~Array()
{
   for (int i=0; i<size; i++)   delete  m[i];
        delete m;
}
void Array<char*>:: print(char *s)
{
    cout << s << ":";
    for (int i=0; i<size; i++) cout << m[i] <<  "  ";
         cout << "\n";
};
void main()
{
    Array<int> mi(5); // определение  класса  Array  для  int
    Array<float> mf(3); // определение класса Array  для  float
    Array<char*> mstr(2); // определение класса Array для char*
    for (int i=0; i<5; i++)  mi.set(i,i);
         mi.print("mi");
     for  (i=0;  i<3;  i++)   mf.set(i,(float)i);
        mf.print("mf");
     mstr.set(0,"Минск");     mstr.set(1,"Москва");
     mstr.print("mstr");
}
```

Лекция 9 .Потоки ввода-вывода. Иерархия классов ввода-вывода. Основные функции. Форматированный и неформатированный ввод-вывод. Функции. Поля управления форматированием, манипуляторы.

Си**стема ввода-вывода C++**

Может появиться вопрос: в связи с чем возникла необходимость

в языке C++ вводить собственную систему ввода-вывода, отличную от языка C ? Давайте обобщим уже известное. C++ позволяет оперировать объектами как обычными переменными языка программирования. Важное место здесь отводится механизму перегрузки операций. Было бы естественным предоставить такую же возможность и в отношении операций ввода-вывода. Для этого необходимо заменить стандартные функции ввода-вывода языка C (типа printf() и scanf()) операторами ввода-вывода.

Дальнейшее совершенствование системы ввода-вывода связано с ее реализацией через концепцию классов, задающих общий интерфейс ввода-вывода, зная который пользователь может создавать собственные библиотеки ввода-вывода.

Система ввода-вывода языка C++ основывается на концепции потока. Поток в C++ - это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. В качестве синонимов вывода данных используются термины извлечение, прием, получение, а синонимов ввода - вставка, помещение, заполнение.

В языке C++ вся передаваемая/принимаемая информация рассматривается как последовательность символов, поскольку любое двоичное представление может быть рассмотрено как неотображаемая последовательность байт.

Перенос данных от источника к приемнику редко осуществляется непосредственно, а обычно связан с временным их хранением в некоторой области памяти, называемой буфером

Система ввода-вывода C++ состоит из четырех основных библиотек: iostream, fstream, strstream и constream. Библиотека iostream обеспечивает ввод-вывод, связанный со стандартными потоками. Библиотека fstream поддерживает работу с файлами, strstream - со строками символов, а constream обеспечивает работу с консолью. Все эти библиотеки позволяют выполнять форматный ввод-вывод с контролем типов как для предопределенных, так и для определеяемых пользователем типов данных с помощью перегруженных операций и других объектно-ориентированных методов.

Система ввода-вывода языка C++ включает два параллельных семейства классов. Классы одного семейства являются производными из базового класса streambuf, а другого - из класса ois. Базовые классы streambuf и ios являются низкоуровневыми и каждый предназначен для решения своего круга задач.

Класс streambuf обеспечивает общие правила буферизации и обработки потоков без форматирования данных. Реализация компонент-функций класса streambuf направлена прежде всего на поддержку максимальной эффективности ввода-вывода. Этот класс используется всеми четырьмя библиотеками системы ввода-вывода языка C++, он также доступен для построения пользовательских производных классов ввода-вывода. Доступ из семейства классов, построенных на основе ios, к классу streambuf осуществляется через указатель на класс streambuf.

Характерными для класса streambuf являются компоненты-функции подсоединения потока к некоторому буферу (setbuf()), извлечения и помещения отдельного символа (sgetc(), sbumpc(),

snextc(), sputc()), нескольких символов (sgetn(), sputn()), перемещения потоковых указателей (seekoff(), seekpos(), stossc()), возврата символа в поток (sputbackc()), определения заполнения буферов (in_avail(), out_waiting). Класс streambuf содержит также ряд защищенных компонент- функций, доступных для производных классов ввода-вывода.

Из базового класса streambuf образуются три производных класса filebuf, strstreambuf и conbuf, соответственно специализирующих класс streambuf для работы с файлами, строками и экраном, которые определяются следующим образом:

class streambuf{...};
class filebuf: public streambuf{...};
class strstreambuf: public streambuf{...};
class conbuf: public streambuf{...};

Стандартные потоки ввода-вывода

Библиотека iostream объявляется с помощью файла заголовков iostream.h и имеет два класса: streambuf и ios.

Класс ios предназначен для форматного ввода-вывода через класс streambuf. Из него производятся три класса istream, ostream и iostream соответственно для ввода, вывода и одновременного ввода-вывода следующим образом:

class ios{...};
class istream: virtual public ios{...};
class ostream: virtual public ios{...};
class iostream: public istream, public ostream{...};

Кроме того, существуют три класса withassign, добавляющие в классы istream, ostream и iostream операцию присваивания:

class istream_withassign: public istream{...};
class ostream_withassign: public ostream{...};
class iostream_withassign: public iostream{...};

Последние классы обеспечивают четыре предопределенных стандартных потока C++: cin, cout, cerr и clog:

extern istream_withassign cin{...};
extern ostream_withassign cout{...};
extern ostream_withassign cerr{...};
extern ostream_withassign clog{...};

Эти потоки автоматически открываются при запуске всякой программы, содержащей файл iosrteam.h и соответствуют следующему:

cin - стандартный ввод (аналогичен stdin);
cout - стандартный вывод (аналогичен stdout);
cerr - стандартный вывод ошибок (аналогичен stderr);
clog - буфферизированная версия потока cerr, позволяющая сохранять протокол ошибок.

Названные стандартные потоки по умолчанию связаны с консолью.

Переназначение их на другие устройства или файлы обеспечивается обычными средствами языка С или операционной системы.

Потоки можно также копировать с помощью операции присваивания, например:

```
cout = cerr;
```

В результате две переменные будут обращаться к одному потоку. Это позволяет использовать стандартные имена (cin, cout, cerr, clog) для обращения к другим потокам.

Два различных потока в языке С++ можно определить как связанные. Это означает, что использование одного потока воздействует на другой поток. Например, если потоки cin и cout связаны, то перед использованием cin разгружается поток cout. По умолчанию стандартные потоки cin, cerr и clog связаны с потоком cout. Для определения потока, связанного с данным, используется функция tie() без аргументов:

```
ostream * tie();
```

возвращающая связанный поток, либо 0 в случае его отсутствия. Для привязки некоторого потока к данному служит функция tie() с аргументом:

```
ostream * tie(ostream * stream);
```

где stream - поток, привязанный к данному (для которого вызывается функция tie()), а возвращается предыдущий привязанный поток. Чтобы развязать поток вызывается tie(0).

Неформатный ввод-вывод

Компоненты-функции put() и write() класса ostream обеспечивают неформатный вывод данных в стандартные потоки. Функция put(), описываемая как

```
ostream & put (char);
```

позволяет вывод двоичных данных или отдельного символа, получаемого в качестве аргумента, в указанный поток, например:

```
int ch = 'a';
cout.put(ch);
```

выводит символ 'a' в поток cout.

Большие по размерам объекты выводятся с помощью функции write():

```
ostream & write (char *p, int n);
```

где p указывает на выводимые данные; n - размер в байтах. В отличие от вывода строк с помощью оператора <<, функция write() не прекращает работу, встретив пустой символ, например:

```
cout.write((char *)&ch, sizeof(ch));
```

пошлет непреобразованное представление ch на стандартное устройство вывода.

Подчеркнем еще раз, что функции put() и write() не выполняют форматирование данных при выводе, они не могут вызывать

разгрузку потоков, установку ширины поля и т.д. Их использование ограничено простыми примерами ввода-вывода, когда предъявляются жесткие требования к быстродействию и размеру абсолютного кода программы.

Для неформатного ввода данных из стандартного потока в C++ служит компонента-функция get() класса istream:

```
istream & get(char *str, int max, int term = '\n');
```

Функция get() считывает символы из входного потока в массив str до тех пор, пока не будет прочитано max-1 символов, либо пока не встретится символ, заданный терминатором (ограничителем) term. К прочитанным данным автоматически добавляется символ конца строки '\0'. По умолчанию значением терминатора является символ новой строки '\n', который в str не считывается и из istream не удаляется.

Для корректной работы программы массив str должен иметь размер не менее max символов, например:

```
char  s[81];
cin.get(s,81);                  // ввод строки с терминала
```

Функция get() в классе istream перегружена и имеет несколько реализаций, например,

```
istream &  get(streambuf & buf, char term = '\n');
```

позволяет читать символы в некоторый буфер потока buf.

Следующие две функции

```
istream &  get(char & ch);
int  get();
```

позволяют читать из потока единственный символ независимо от того, является он пробельным или нет, например:

```
#include <iostream.h>
void main(){
  char c1, c2;
  cout.write("Введите два символа\n", 20);
  cin.get(c1);          // ввод символа через параметр
  c2 = cin.get();       // ввод символа через
                        //возвращаемое значение
  cin.get();            // выборка из потока лишнего символа
  cout.write("Были введены символы: ", 23);
  cout.put(c1); cout.put(' '); cout.put(c2); cout.put('\n');
}
```

Для неформатного ввода блока данных служит функция read(), определенная в классе istream:

*istream & read(char *p, int n);*

где p указывает на выводимые данные, а n - размер данных в байтах. Следующий пример позволяет вводить и выводить непреобразованное значение последовательности символов:

```
void main(){
  int s;
  cout << "Введите пять символов\n";
  cin.read(s, sizeof(s));
  cout << "Было введено: ";
  cout.write(s, sizeof(s));
}
```

Стандартные потоки cin и cout осуществляют последовательный символьный ввод-вывод. Функции же read() и write() наиболее эффективны при вводе-выводе двоичных блоков данных, поэтому их часто используют для работы с бинарными файлами

В библиотеке iostream имеются еще несколько полезных функций неформатного ввода-вывода:

*istream & getline(char *str, int max, int term = '\n');*
- подобна функции get() за исключением того, что ограничивающий символ term извлекается, но не копируется в str (удобна для чтения строк);

int peek();
- возвращает следующий символ без извлечения из потока (просмотр вперед);

int gcount();
- возвращает число символов последнего извлечения;

istream & putback(char ch);
- возвращает обратно во входной поток символ ch;

istream & ignore(int n = 1, int term = EOF);
- пропускает n символов во входном потоке, останавливается, когда встретится term;

ostream & flush();
- разгружает (т.е. выводит содержимое) поток на связанное с ним устройство.

В качестве примера рассмотрим функцию компилятора для ввода индентификаторов языка C++

```
void getid(char *s)

{
  char c = 0;          // защита от конца файла
```

```
    cin >> c;              // пропуск пробельных символов
    if (isalpha(c) || c == '_')    // если буква или '_'
      do{
        *s++ = c;        // введенный символ помещаем в s
        c = 0;           // защита от конца файла
        cin.get(c);      // ввод следующего символа
      } while (isalnum(c) || c == '_'); // до тех пор, пока
                                // буквы, цифры или '_'
    *s = 0;              // устанавливаем символ конца строки
    if (c)
      cin.putback(c);    // возвращаем лишний символ в поток
  }
```

Суть защиты от конца файла заключается в том, что если при вводе будет обнаружен конец файла, значение переменной c останется нулевым и функция getid() корректно завершит свою работу.

Форматный ввод-вывод

В языке C++ форматный вывод в поток выполняется с помощью перегруженной операции сдвига влево "<<", называемой при выводе оператором вставки или помещения. Для форматного ввода используется перегруженная операция сдвига вправо ">>", которая при вводе называется оператором извлечения или просто извлечением. Левый операнд оператора вставки представляет собой объект класса ostream, а оператора извлечения - istream. Правый операнд операторов << и >> может быть любого типа, для которого определен ввода-вывод потоком (стандартные типы языка C++, а также типы, определяемые пользователем и имеющие возможность вывода потоком). В библиотеке iostream ввод-вывод потоком определен для всех арифметических типов (char, short, int, long, float, double), строки символов (char *), значений указателей (void *), а также для буфера потока (streambuf *).

Операция << ассоциативна слева и возвращает ссылку на объект ostream, для которого она вызывалась. Это позволяет связывать операцию вставки в цепочки, например:

```
cout << "i= " << i << "j= " << j << "\n";
```

что равносильно следующей записи:

```
cout << "i= ";
cout << i;
cout << "j= ";
cout << j;
cout << "\n";
```

По умолчанию извлечение опускает пробельные символы ('\v', '\t', '\n' и пробел), а затем считывает символы, соответствующие типу объекта ввода. Как и в случае оператора вставки, извлечение ассоциативна слева и возвращает ссылку на объект istream, для которого она вызывалась. Это позволяет объединить в одном

операторе несколько операций ввода, например:

```
int   k;
float y;
cin >> k >> y;
```

Последний оператор вызовет пропуск пробельных символов, считывает цифры со стандартного устройства ввода (консоли) до тех пор, пока не встретится символ, не соответствующий формату целого числа, преобразует введенное значение в двоичное представление целого и записывает в переменную k. Затем вновь пропускаются пробельные символы, считывается число с плавающей точкой, преобразуется во внутренний формат и записывается в переменную y.

Для других предопределенных типов C++ действие оператора извлечения аналогично: пропускаются пробельные символы, выполняется необходимое преобразование и осуществляется запись значения в указанную переменную. Строка символов считывается до первого встретившегося пробельного символа, затем добавляется нулевой символ '\0'.

Отметим, что в операторе извлечения указываются не адреса переменных (как это требует функция scanf()), а сами переменные. Следует также проявлять осторожность при считывании строки, так как контроль конца массива символов операция извлечения не выполняет. Например:

```
#include <iostream.h>
main(void)
{
  char str[81];
  cout << "Ведите строку\n";
  cin >> str;
  cout << "Была введена строка: " << str << endl;
}
```

Поскольку при перегрузки операций приоритет их не изменяется, то приоритет операторов форматного ввода-вывода совпадает с приоритетом операций сдвига. Поэтому при выводе операнд, заданный выражением, берется в скобки только в том случае, когда выражение содержит операции более низкого приоритета.

Форматирование ввода и вывода определяется форматирующими флагами, представляющими собой биты числа типа long int, определенного в классе ios следующим образом:

```
public:
  enum{
    skipws     = 0x0001,   // пропуск при вводе пробельных
                           // символов
    left       = 0x0002,   // левое выравнивание вывода
    right      = 0x0004,   // правое выравнивание вывода
    internal   = 0x0008,   // заполнение пробелами поля между
                           // знаком или основанием системы
                           // счисления и числовым значением
    dec        = 0x0010,   // десятичное представление чисел
```

```
    oct       = 0x0020,   // восьмеричное представление чисел
    hex       = 0x0040,   // шестнадцатеричное представление
                          // чисел
    showbase  = 0x0080,   // при выводе  чисел показывается
                          // основание системы счисления
    showpoint = 0x0100,   // при выводе плавающих чисел
                          // показывается позиция десятичной
                          // точки
    uppercase = 0x0200,   // вывод шестнадцатеричных значений
                          // буквами верхнего регистра
    showpos   = 0x0400,   // при выводе положительных целых
                          // чисел показывается знак "+"
    scientific = 0x0800,  // плавающие числа отображаются
                          // в экспоненциальной форме
    fixed     = 0x1000,   // плавающие числа отображаются
                          // с десятичной точкой
    unitbuf   = 0x2000,   // разгрузка всех потоков после
                          // вставки
    stdio     = 0x4000,   // разгрузка после вставки потоков
                          // stdout  и  stderr
  }
```

Отметим, что по умолчанию символ экспоненты "e" для чисел с плавающей точкой и символ "x" для шестнадцатеричных чисел отображаются на нижнем регистре. Если ни один из флагов dec, oct и hex не установлены, то числа выводятся в десятичном представлении. Установка флага stdio позволяет разгружать выходной поток на физическое устройство после каждой операции вывода. Флаг unitbuf выполняет усовершенствование системы ввода-вывода C++ и по умолчанию всегда установлен.

При отсутствии специальных действий со стороны пользователя флаги в C++ устанавливаются так, чтобы обеспечивать ввод-вывод, соответствующий умалчиваемым форматам функций printf() и scanf().

Ряд функций библиотеки iostream предоставляет пользователю возможность управлять форматным вводом-выводом. Для установки флагов пользователя используется функция setf():

```
long  setf(long flags);
```

Эта функция возвращает предыдущую установку флагов и изменяет флаги, определенные в flags. Обращение к функции имеет вид

```
stream.setf(ios::flag);
```

где stream - поток, на который вы желаете воздействовать; flag - изменяемый флаг. Например, следующая программа изменяет флаги hex и scientific:

```
#include <iostream.h>
main(void)
```

```
{
  cout.setf(ios::hex);
  cout.setf(ios::scientific);
  cout << 365 << " " << 365.78 << "\n";
}
```

В результате работы программы получим:

```
16d 3.6578e+02
```

В одном вызове функции setf() можно одновременно устанавливать несколько флагов, объединяя их с помощью поразрядного сложения:

```
cout.setf(ios::hex | ios::scientific);
```

Для сброса флагов используется функция unsetf():

```
long unsetf(long flags);
```

Функция возвращает предыдущую установку и сбрасывает флаги, определенные в flags. Совместить действия функций setf() и unsetf() можно с помощью функции setf(), принимающей два аргумента:

```
long setf(long setbits, long field);
```

которая вначале сбрасывает флаги, определенные в field, а затем устанавливает флаги, отмеченные в setbits.

Для определения значений флагов без их изменения используется функция flags() без параметров:

```
long flags(void);
```

Функция flags() с аргументом:

```
long flags(long bits);
```

работает точно так же, как и setf(long). Для установки флагов в умалчиваемое значение используется flags(0).

В качестве примера рассмотрим программу вывода умалчиваемых значений форматирующих флагов стандартных потоков:

```
#include <iostream.h>
// функция type_flags выводит значения форматирующих
// флагов f
void type_flags(long f)
{
  for (long i = 0x4000; i; i >>= 1) // формирование маски
    if (i & f)   cout << "1";      // определение значения
    else         cout << "0";      // флага
  cout << "\n";
}
main(void)
{
  long f;
  f = cout.flags();      type_flags(f);
  f = cin.flags();       type_flags(f);
  f = cerr.flags();      type_flags(f);
```

```
  f = clog.flags();      type_flags(f);
  cout.setf(ios::hex | ios::scientific);
  f = cout.flags();      type_flags(f);
}
```

Результатом работы программы будет:

```
010000000000001
000000000000001
010000000000001
000000000000001
010100001000001
```

Следующие три функции библиотеки iostream позволяют устанавливать ширину поля, заполняющий символ и число цифр, показываемых после запятой:

```
int  width(int len);
char fill(char ch);
int  precision(int num);
```

где len - ширина поля, ch - символ заполнения, num - число цифр после запятой в отображении числа с плавающей точкой. Все эти функции возвращают предыдущие значения соответствующих параметров. По умолчанию ширина поля равна нулю, это означает, что будет выведено минимальное число символов, которыми может быть представлено выводимое значение. В избыточных позициях выводится символ заполнителя, заданный функцией fill(), по умолчанию - это пробел. Если указанная ширина не достаточна для представления выводимого значения, то она будет проигнорирована и вывод выполняется как для нулевой ширины. Вызов функций width(), fill() и precision() без аргументов возвращает предыдущее значение соответствующих параметров без их изменения, например:

```
#include <iostream.h>
main(void)
{
  cout.setf(ios::hex);
  cout.setf(ios::scientific);
  cout << 365 << " " << 365.78 << "\n";
  cout.precision(2);
  cout.width(10);
  cout << 365 << " " << 365.78 << "\n";
  cout.fill('*');
  cout << 365 << " " << 365.78 << "\n";
}
```

Результат работы программы получим

```
16d 3.6578e+02
      16d 3.66e+02
16d 3.66e+02
```

Форматирование с помощью манипуляторов

Мы уже могли видеть как функции ввода-вывода языка С были заменены операторами ввода-вывода в языке С++. Однако в эту концепцию не вписываются функции управления флагами формата.

Возникает вопрос: нельзя ли их также заменить некоторыми конструкциями, подобными операторам ввода-вывода, с тем, чтобы не ломать общий стиль системы ввода-вывода языка C++ ?

С этой целью в язык C++ введено понятие манипулятора. Манипуляторы - это специальные функции, которые принимают в качестве аргументов ссылку на поток и возвращают ссылку на тот же поток. Поэтому они могут объединяться в цепочку вместе с операторами ввода-вывода. Сами манипуляторы никаких действий по вводу или выводу не выполняют, однако, осуществляют "побочный" эффект, воздействуя на флаги формата и другие параметры ввода-вывода. Например, для вывода переменной x в поле шириной 5, а переменной y в поле шириной 10, используется манипулятор setw() установки ширины поля:

```
cout << setw(5) << x << setw(10) << y << "\n";
```

Для возможности работы с манипуляторами C++ необходимо в программу включить файл iomanip.h.

Установленные с помощью манипуляторов режимы ввода-вывода не сохраняются на последующие операторы ввода-вывода, например, в результате выполнения следующего фрагмента :

```
int x = 561;
float f = 3.141529;
cout << setw(5) << x << setw(10) << y << 15 << "AAA" << "\n";
```

получим:

```
561  3.14152915AAA
```

В следующей таблице приведены все манипуляторы C++:

Манипулятор	Операторы	Действие
dec	<<,>>	десятичное представление чисел
oct	<<,>>	восьмеричное представление чисел
hex	<<,>>	шестнадцатеричное представление чисел
ws	>>	извлечение пробельных символов
endl	<<	вставка символа новой строки '\n' и разгрузка потока
ends	<<	вставка в строку символа '\n' конца строки
flush	<<	очистка потока ostream
setbase(int n)	<<,>>	установка системы счисления с основанием n, где n - одно из {0,8,10,16}; ноль означает десятичную систему счисления при выводе и правила C для литералов целых чисел при вводе
setiosflags(long f)	<<,>>	установка флагов, указанных в f
resetiosflags(long f)	<<,>>	сброс флагов, указанных в f
setfill(int ch)	<<,>>	установка символа-заполнителя в ch
setprecision(int n)	<<,>>	установка числа цифр после запятой в представлении чисел с плавающей запятой
setw(int l)	<<,>>	установка ширины поля в l

Например, оператор

```
cout << setw(10) << setprecision(2) << setfill('*') << 365.766;
```

позволяет вывести число 365.766 в поле шириной 10 позиций, после запятой отобразится 2 цифры (будет выполнено округление последней цифры) и заполняющий символ будет '*':

```
****365.77
```

Создание собственных манипуляторов

Как вы уже, наверное, догадались, что такая гибкая система, как C++, не может не предоставить средство пользователю для создания манипуляторов по своему усмотрению. Определение собственных манипуляторов пользователя для вывода имеет следующую структуру:

```
ostream & <имя_манипулятора>(ostream & stream)
{
  // необходимый код
  return stream;
}
```

Хотя здесь в качестве аргумента используется ссылка на поток, на самом деле этот аргумент не используется. Рассмотрим пример заказного манипулятора, определяющего вывод числа с плавающей точкой из предыдущего примера :

```
#include <iostream.h>
#include <iomanip.h>
ostream & my_manip(ostream & stream)
{
  stream << setw(10) << setprecision(2) << setfill('*');
  retutr stream;
}
main(void)
{
  cout << 365.766 << my_manip << 365.766 << endl;
}
```

Заказные манипуляторы являются полезными в двух случаях. Во-первых, когда осуществляется вывод на устройство, которое не выполняет применяемые манипуляторы, например, плоттер. В этом случае создание собственных манипуляторов позволяет выполнять необходимые преобразования во время вывода. Во-вторых, когда часто повторяется последовательность одних и тех же манипуляторов. В этом случае их можно объединить в один манипулятор, как было показано в предыдущем примере.

Определение собственных манипуляторов пользователя для ввода имеет следующую структуру:

```
istream & <имя_манипулятора>(istream & stream)
{
  // необходимый код
```

```
  return stream;
}
```

Например:

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

istream & manip_in_hex(istream & stream)
{
 cout << "Введите число, используя шестнадцатеричный формат";
 cin >> hex;
 retutr stream;
}
main(void)
{
  int i;
  cin >> manip_in_hex >> i;
  cout << i << endl;
}
```

Перегрузка операторов ввода-вывода

Наибольшая эффективность операторов ввода-вывода языка C++ наблюдается при их использовании для работы с объектами классов. С этой целью выполняется их перегрузка. Перегружаемый оператор ввода-вывода первым аргументом должен иметь ссылку на связанный с ним поток, вторым аргументом - объект, записываемый справа от оператора (обычно это ссылка на объект данного класса). Возвращаемым значением является ссылка на связанный с данным оператором поток.

Поскольку первый аргумент в операторе ввода-вывода не может быть объектом данного класса, а значит, и указателем this, то перегружаемые операции ввода-вывода не могут быть компонентами класса. Поэтому общий формат перегрузки операторов ввода-вывода для некоторого класса X имеет вид

```
class X{
  . . . . . .
  friend ostream & operator << (ostream &, X &);
  friend istream & operator >> (istream &, X &);
}
. . . . . .
ostream & operator << (ostream & stream, X x);
{
  // операторы вывода
  return stream;
}

istream & operator >> (istream & stream, X &x);
{
  // операторы ввода
  return stream;
}
```

Отметим, что рациональнее в качестве первого аргумента указывать абстрактный поток stream, а не cout или cin, для того, чтобы операторы могли работать с любыми потоками. В качестве примера рассмотрим перегрузку операторов ввода-вывода для класса complex:

```
#include <iostream.h>
class complex{
public:
  double re, im;
  complex (double r=0, double i=0)
        { re = r; im = i; };
  friend ostream & operator << (ostream &, complex &);
  friend istream & operator >> (istream &, complex &);
};
ostream & operator << (ostream & stream, complex &x);
{
  stream << "Комплексное число= ";
  stream << "(" << x.re << "," << x.im << ")" << endl;
  return stream;
};
istream & operator >> (istream & stream, complex &x);
{
  cout << "Введите действительную и мнимую части: ";
  stream >> x.re >> x.im;
  return stream;
};
main(void)
{
  complex a;
  cin >> a;
  cout << a;
}
```

Ошибки потоков ввода-вывода

С каждым открытым потоком в C++ связывается перечислимая переменная io_state, определяющая биты состояния потока. Она объявлена в классе ios и может рассматриваться как целая величина:

```
public:
  enum  io_state{
    goodbit  = 0x00,   // если бит не установлен,
                       // то ошибок нет
    eofbit   = 0x01,   // обнаружен конец файла
    failbit  = 0x02,   // сбой в последней операции
                       // ввода-вывода
    badbit   = 0x04,   // попытка недопустимой операции
    hardfail = 0x80    // в потоке невосстанавливаемая ошибка
  };
```

В случае установки eofbit игнорируются попытки выполнить операции извлечения; бит failbit может быть сброшен и продолжено использование потока; после сброса badbit не всегда можно

восстановить работоспособность потока; перед сбросом hardfail требуется установить причину, вызвавшую ошибку.

После того как для некоторого потока возникло состояние ошибки, все попытки ввода-вывода будут игнорироваться до тех пор, пока не будет устранена причина, вызвавшая ошибку, а биты ошибки не сброшены с помощью функции clear(), например:

```
in.clear(0);                    // очистка всех бит ошибок
in.clear(ios::eofbit);          // очистка бита eofbit
```

Хорошим стилем программирования считается проверка состояния ошибки в наиболее ответственных точках программы. Это можно выполнить с помощью следующих функций, возвращающих ненулевые значения, если:

good() - не было ошибки;
eof() - обнаружен конец файла;
fail() - был установлен один из битов failbit, badbit, hardfail;
bad() - был установлен бит badbit или hardfail.

Текущее состояние ошибки можно получить с помощью функции rdstate(), которая возвращает номер бита ошибки, например :

```
cout << "Состояние потока  cin: " << cin.rdstate() << endl;
cout << "Состояние потока  cout: " << cout.rdstate() << endl;
cout << "Состояние потока  cerr: " << cerr.rdstate() << endl;
cout << "Состояние потока  clog: " << clog.rdstate() << endl;
```

Кроме того, в классе ios имеются перегруженные операции

```
int operator !();
operator void *();
```

Операция void *() определяет преобразование потока в указатель, который будет равен нулю в случае установления бит failbit, badbit или hardfail и ненулевому значению в противном случае. Операция "!" наоборот возвращает ненулевое значение, если установлен один из бит failbit, badbit или hardfail, и возвращает нулевое значение в противном случае. Это позволяет рассматривать в логических выражениях в качестве переменной непосредственно сами потоки ввода-вывода, например:

```
#include <iostream.h>
int main(){
  int x;
  if (!cout)                    // ошибка вывода!
    return -1;
  cout << "Введите целое число\n";
  if (cin >> x)                 // все в порядке!
    cout << "Введено" << x << endl;
  else{                         // ошибка вывода!
    cout << "Ошибка ввода!\n";
```

```
    return -1;}
  return 0;
}
```

Лекция 10.Файловый ввод-вывод. Классы файлового ввода-вывода. Организация доступа к файлу. Основные функции.

Библиотека fstream и открытие файлов

Файлы в C++ рассматриваются как потоки, связанные с конкретными именами физических файлов, например, на диске. Библиотека fstream языка C++ определяется с помощью заголовочного файла <fstream.h>. Она содержит следующую структуру классов:

class filebuf: public streambuf {...};
class fstreambase: virtual public ios {...};
class ifstream: public fstreambase, public istream {...};
class ofstream: public fstreambase, public ostream {...};
class fstream: public fstreambase, public iostream {...};

Класс filebuf (буфер файла) специализирует низкоуровневый класс streambuf для управления файлами. Ключевую роль при этом играет дескриптор файла - 16-битовое значение, определяемое при открытии файла. Все последующие манипуляции с файлом (чтение, запись, позиционирование, закрытие и т.д.) так или иначе ссылаются на дескриптор файла. Характерными для класса filebuf являются компоненты- функции открытия и закрытия файла (open() и close()), подсоединения буфера к дескриптору открытого файла (attach()), определения дескриптора файла (fd()) и проверки открыт ли файл (is_open()). Кроме того, файл filebuf содержит ряд виртуальных функций (overflow(), seekoff(), setbuf(), sync() и underflow()), которые в случае использования должны переопределяться в производных классах.

Класс fstreambase образуется из класса ios и обеспечивает основные операции ввода-вывода для файловых потоков. Связь класса fstreambase с буфером файла осуществляется при помощи указателя на класс filebuf. Класс fstreambase кроме компонент- функций, наследуемых из класса ios, содержит также функции открытия и закрытия файла (open() и close()), подсоединения дескриптора к открытому файлу (attach()), задания определенного буфера (setbuf()) и определения используемого буфера файла (rdbuf()).

Остальные классы библиотеки fstream: ifstream, ofstream и fstream производятся из класса fstreambase и классов istream, ostream и iostream соответственно и новых компонент- функций не содержат.

Для того, чтобы начать работать с файлом, прежде всего необходимо определить файловый поток, который должен быть объектом соответствующего класса, например:

```
ifstream  in;    // in   - входной поток
ofstream  out;   // out  - выходной поток
fstream   inout; // inout - может быть как входным, так и
                 // выходным потоком
```

Уже созданный поток связывается с конкретным именем физического файла с помощию функции open(), которая имеет

вид:

```
void open(char *filename, int mode, int access);
```

где filename - имя физического файла, которое может включать определенный путь;

mode - режим открытия файла;

access - задает атрибут доступ к файлу.

Режим открытия файла определяется в классе ios следующим образом:

```
public:
  enum open_mode{
    in  = 0x01,      // открыть для чтения
    out = 0x02,      // открыть для записи
    ate = 0x04,      // переместиться в конец файла
                     // при первоначальном открытии
    app = 0x08,      // режим добавления в конец: все
                     // дополнения выполняются в конец файла
    trunc    = 0x10, // очистка содержимого: если файл
                     // существует, то его содержимое
                     // уничтожается
    nocreate  = 0x20, // не создавать: если файл не существует,
                     // то новый не создается и не открывается
    noreplace = 0x40, // не замещать: если файл уже существует,
                     // то новый не создается и не открывается
    binary    = 0x80  // создается бинарный файл (по умолчанию
                     // создается текстовый файл
  }
```

При задании режима открытия файла эти параметры могут объединяться с помощью операции поразрядного сложения (|).

Атрибут доступа к файлу соответствует атрибуту файла в операционной системе (MS-DOS, UNIX и др.) и может принимать следующие значения:

0 - обычный файл;

1 - файл доступен только для чтения;

2 - скрытый файл;

4 - системный файл;

8 - архивный файл.

Примеры открытия файлов:

```
in.open("INFILE",ios::in,0);              // файл для ввода
out.open("OUTFILE",ios::out | ios::app,0);   // файл для вывода
inout.open("IOFILE",ios::in | ios::out,0);   // файл для ввода
                                          // и вывода
```

Отметим, что умалчиваемым значением атрибута доступа access является 0; умалчиваемым значением режима открытия mode для объектов класса ifstream является ios::in, для ofstream - ios::out.

Открытие файла можно выполнять одновременно с объявлением переменной потока, передавая конструктору соответствующие параметры, например:

```
ifstream in("INFILE");                          // файл для ввода
ofstream out("OUTFILE",ios::out | ios::app); // файл для вывода
```

```
fstream  inout("IOFILE",ios::in | ios::out);     // файл для ввода
                                                 // и вывода
```

Если файл открыть не удалось, переменная потока будет иметь нулевое значение, поэтому проверить успешное открытие файла можно следующим образом:

```
ifstream  in("INFILE");
if (in == 0)
{
  cout << "Нельзя открать файл INFILE\n";
  . . . . . .
}
```

Закрытие файлов выполняется функцией close(), не имеющей параметров и не возвращающей никакого значения, например:

```
in.close();
out.close();
inout.close();
```

Не смотря на то, что при завершении программы на языке C++ все открытые файлы закрываются автоматически, хорошим стилем программирования считается явное закрытие файлов при выходе их программы. Не следует также в процессе выполнения программы держать длительное время файлы открытыми, если в этом нет необходимости. Рекомендуется также всегда проверять успешное открытие файла перед выполнением каких-либо манипуляций с файловым потоком (для стандартных потоков это не столь актуально).

Чтение и запись текстовых файлов

По умолчанию все файлы в языке C++ открываются в текстовом режиме. Поэтому нет необходимости в специальном указании того, что работа ведется с текстовым файлом. Чтение и запись текстовых файлов осуществляется с помощью операторов форматного ввода-вывода языка C++ >> и <<, определенных в классе ios, только вместо стандартных потоков cin и cout указываются имена файлов. При этом справедливы все механизмы C++, касающиеся форматного ввода-вывода (использование манипуляторов, перегрузка операторов ввода-вывода и т.д.). Например :

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
int main(){
  ofstream out("MY_FILE");
  if (!out){
    cout << "Нельзя открыть файл MY_FILE\n";
    return -1;
  }
  out << setw(5) << 3.15 << setw(5) << 15
      << endl << "Это текстовый файл\n";
  out.close();
  int i;
  float pi;
  char s1[81],s2[81],s3[81];
```

```
    ifstream in("MY_FILE");
    if (!in){
      cout << "Нельзя открыть файл MY_FILE\n";
      return -1;
    }
    in >> pi >> i                // чтение переменных
      >> s1 >> s2 >> s3;         // чтение строки
    cout << "pi= " << pi << "i= " << i      // вывод переменных
        << "s= " << s1 << s2 << s3 << endl; // вывод строки
    in.close();
    return 0;
  }
```

Напомним, что оператор извлечения >> предусматривает пропуск пробельных символов потока и строки читаются до первого пробельного символа. Поэтому в предыдущем примере строку из трех слов пришлось считывать в три разные переменные. Кроме того, при вводе текстовых файлов последовательность двух управляющих символов возврата каретки и перевода строки "x10x13" автоматически преобразуется в один символ новой строки "\n". При выводе выполняется обратное преобразование. Если подобные преобразования по какой-либо причине не приемлемы, то следует пользоваться бинарными файлами.

Чтение и запись бинарных файлов

В C++ бинарные и текстовые файлы определяются не способом их открытия (как может ожидаться), а используемыми функциями (операторами) ввода-вывода. Для простейшего чтения и записи символов бинарного файла используются функции get() и put(). Прочитать единственный символ ch из файла можно с помощью функции get(). В этом случае ее прототип имеет вид:

```
ifstream & get(char & ch);
```

Функция get() имеет несколько перегруженных версий , одна из которых объявляется как

```
ifstream & get(char *str, int n, char term = '\n');
```

и позволяет читать в строку str до тех пор, пока не встретится терминатор term, либо не будет прочитано n-1 символов.

Функция put() записывает в файл единственный символ ch, ее прототип:

```
ofstream & put(char & ch);
```

В качестве примера рассмотрим программу копирования двух файлов:

```
#include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){
  if (argc != 3){
    cout << "Использование:\n"
        << " исходный_файл принимающий_файл\n";
    return -1;
  }
  ifstream in(argv[1]);
  if (!in){
```

```
    cout << "Нельзя открыть исходный файл\n";
    return -1;
  }
  ofstream out(argv[2]);
  if (!out){
    cout << "Нельзя открыть принимающий файл\n";
    return -1;
  }
  char ch;
  while ( (ch=in.get()) != EOF)
    out.put(ch);
  return 0;
}
```

Чтение и запись блоков двоичных данных выполняется с помощью функций read() и write(), которые имеют вид:

```
istream & read(char *buf, int n);
ostream & write(char *buf, int n);
```

Функция read() читает n байт из потока и помещает их в буфер buf, причем buf может быть объявлена как знаковая, так и беззнаковая символьная переменная. Функция write() пишет n байт в поток из буфера, указанного buf. В случае, когда наступает конец файла до прочтения n символов, функция read() останавливается, а буфер будет содержать прочитанные символы. Определить число символов, прочитанных последней функцией read(), можно с помощью функции gcount(), имеющую прототип:

```
int gcount();
```

Если нас не устраивает посимвольное копирование файлов предыдущего примера, это можно сделать с помощью функций read() и write() :

```
#include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){
  if (argc != 3){
    cout << "Использование:\n"
         << " исходный_файл принимающий_файл\n";
    return -1;
  }
  ifstream in(argv[1]);
  if (!in){
    cout << "Нельзя открыть исходный файл\n";
    return -1;
  }
  ofstream out(argv[2]);
  if (!out){
    cout << "Нельзя открыть принимающий файл\n";
    return -1;
  }
  char s[512];
  while (in){
    in.read(s,512);
```

```
      out.write(s,gcout());
    }
    return 0;
  }
```

Кроме рассмотренных имеется еще ряд полезных функций, облегчающих работу с файлами:

ifstream & getline(char *str, int n, char term = '\n');

- работает также, как get(), только терминатор извлекается из потока;

ifstream & ignore(int n = 1, int term = EOF);

- пропускает в потоке n символов, останавливается, если встрерится term;

ostream & flush();

- разгружает выходной поток на физическое устройство.

-

Произвольный доступ к файлу

При открытии в С++ всякого файла образуются два указателя get - для определения позиции, откуда будет считываться очередная информация, и put - куда будут помещаться данные в следующей операции вывода.

На логическом уровне файл в языке С++ может рассматриваться как бинарный массив, расположенный в оперативной памяти Размер массива может быть достаточно большим, он определяется объемом физического устройства. Будем считать, что файловые указатели put и get могут свободно перемещаться по всей длине файла. В дествительности доступный участок файла расположен в буфере, который периодически загружается/разгружается на физическое устройство. Вопросы буферизации реализуются средствами операционной системы и здесь не рассматриваются. Программист может, при необходимости, только устанавливать размер буфера.

Для определения значений указателей get и put служат функции tellg() и tellp(), имеющие прототипы:

```
streampos  tellg();
streampos  tellp();
```

где streampos - тип, определенный в iostream.h, который в состоянии содержать большую величину. Функция tellg() позволяет получить позицию get, а функция tellp() - позицию put.

Перемещение файловых указателей get и put выполняется функциями seekg() и seekp():

```
istream & seekg(streamoff offset, seek_dir orgn);
ostream & seekp(streamoff offset, seek_dir orgn);
```

Типы streamoff и seek_dir также определены в iostream.h. streamoff аналогичен streampos и определяет смещение указателя на offset байт, а seek_dir является перечислением и имеет вид:

```
public:
  enum seek_dir{
    beg = 0,      // смещение от начала файла
    cur = 1,      // смещение от текущего положения указателя
    end = 2       // смещение от конца файла
  }
```

Функция seekg() управляет указателем get, а функция seekp() - указателем put.

Произвольный доступ к файлу продемонстрируем на примере программы, которая позволяет просматривать содержимое любого байта файла:

```
#include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){
  if (argc != 2){
    cout << "Использование:\n <имя_файла>\n";
    return -1;
  }
  ifstream in(argv[1]);
  if (!in){
    cout << "Нельзя открыть файл" << argv[1] << endl;
    retutn -1;
  }
  int offset;
  cout << "Введите номер просматриваемого байта\n";
  cin >> offset;
  in.seekg(offset,ios::beg);
  cout << in.get() << endl;
  return 0;
}
```

Строковый ввод-вывод

Иногда бывает удобно рассматривать в качестве потока массив символов, расположенный непосредственно в оперативной памяти. Например, при разработке компилятора может возникнуть необходимость неоднократного сканирования одной и той же строки, либо возврата обратно в поток целой строки символов. Для обеспечения такой возможности C++ предоставляет библиотеку strstream для работы со строками символов как с обычными потоками ввода-вывода. Классы библиотеки strstream определены в заголовочном файле <strstream.h> и имеют следующую структуру:

```
class strstreambuf: public streambuf {...};
class strstreambase: virtual public ios {...};
class istrstream: public strstreambase, public istream {...};
class ostrstream: public strstreambase, public ostream {...};
class strstream: public strstreambase, public iostream {...};
```

Связь потока с конкретным символьным массивом осуществляется с помощью конструкторов соответствующих классов. Класс istrstream определяет входной строковый поток и имеет два конструктора:

```
istrstream(char *buf);
istrstream(char *buf, int n);
```

Первый конструктор позволяет рассматривать всю строку buf как входной поток, ограниченный нулевым символом '\0' конца строки, а второй делает то же самое, но разрешает использовать только n символов строки buf.

Класс ostrstream определяет выходной строковый поток и

также имеет два конструктора:

```
ostrstream();
ostrstream(char *buf, int n, int mode=ios::out);
```

Первый конструктор создает динамический выходной строковый поток, память которому выделяется по мере необходимости. Второй конструктор определяет строку buf как выходной поток в режиме mode, ограниченный n символами. Если mode имеет значение ios::app или ios::ate, то указатель put устанавливается на символ конца строки '\0'.

Класс strstream имеет два конструктора:

```
strstream();
strstream(char *buf, int n, int mode);
```

которые аналогичны конструкторам класса ostrstream.

Приведем примеры определения строковых потоков:

```
char buf[81];
istrstream ins(buf,81);          // входной строковый поток
ostrstream out(buf,81,ios::app); // выходной строковый поток
strstream  ios(buf,81,ios::in | ios::out); // как входной, так
                                 // и выходной поток
ostrstream outsd;                // динамический выходной поток
```

Для всех строковых потоков справедливо применение рассмотренных выше операторов форматного ввода-вывода << и >>, а также использование манипуляторов. Строковые потоки также наследуют все функции неформатного ввода-вывода. Кроме этого чтение и запись в массив строк единственного символа осуществляется с помощью функций sgetc() и sputc(), имеющих прототипы:

```
int  sgetc();
int  sputc(int ch);
```

например:

```
char ch;
ch = is.sgetc();
os.sput(ch);
```

Получение из массива и помещение в массив сразу n символов выполняется с помощью функций sgetn() и sputn(); имеющих вид:

```
int sgetn(char *srt,int n);
int sputtn(char *srt,int n);
```

Возврат в массив последнего прочитанного символа выполняет функция sputbackc() вида:

```
int sputbackc(char);
```

Заполнение выходного буфера можно определить с помощью функции pcount():

```
char * pcount();
```

которая возвращает количество байт, запомненных в буфере. Можно также "заморозить" буфер, запретив запись в него любых символов с помощью функции str():

```
char * str();
```

причем, если буфер был динамический, то его следует освободить.

В качестве примера рассмотрим программу, которая позволяет

вводить в произвольном формате строку, содержащую целое, плавающее и строку символов, и выводить ее в отформатированном виде :

```
#include <strstrea.h>
#include <string.h>
void main(){
  int i;
  float f;
  char str[81], buf[81];
  cout << "Введите строку, содержащую целое, плавающее "
       << "и строку символов\n";
  cin.getline(buf,81);
  istrstream  ins(buf, strlen(buf));
  ins >> i >> f >> ws;
  ins.getline(str,81);
  cout << i << "\t" << f << "\t" << str << endl;
}
```

Консольный вывод

Для потокового вывода на экран в текстовом режиме служит библиотека constream. Ее классы определены в заголовочном файле <constream.h> и имеют вид:

```
class conbuf: public streambuf{...};
class constream: public ostream{...};
class omanip_int_int{...};
```

Класс constream предоставляет функциональные возможности библиотечных функций языка C, объявленных в <conio.h>. Конструктор класса

```
constream();
```

позволяет создавать консольные выходные потоки, не связанные с экраном. Для установления соответствия консольного потока с прямоугольной областью на экране служит функция window():

```
void window(int left, int top, int right, int bottom);
```

где left и top определяют координаты левого верхнего угла окна на экране, а right и bottom - координаты правого нижнего угла.

После установления связи консольного потока с окном на экране, можно производить вывод на экран, пользуясь всеми возможностями форматного и неформатного вывода языка C++. Кроме этого, библиотека constream предоставляет рад функций, специально предназначенных для вывода на экран. Эти функции фактически повторяют библиотечные функции языка C, поэтому ограничимся их кратким описанием:

void clrscr() - очищает окно;
void clreol() - очищает до конца строки;
void setcursortype(int) - устанавливает вид курсора;
void textcolor(int) - устанавливает цвет символов;
void textbackground(int) - устанавливает цвет фона;
void textattr(int) - устанавливает байт атрибута;
void highvideo() - устанавливает повышенную яркость символов;

void lowvideo() - устанавливает пониженную яркость символов;
void normvideo() - устанавливает нормальную яркость символов;
void gotoxy(int, int) - перемещает курсор в заданную позицию;
int wherex() - возвращает горизонтальную позицию курсора;
int wherey() - возвращает вертикальную позицию курсора;
void delline() - удаляет строку;
void insline() - вставляет строку;
static void textmode(int) - устанавливает новый текстовый режим экрана. Например, вывод строки в центре экрана можно осуществить следующим образом:

```
#include <constrea.h>
void main(){
  constream win;
  win.window(30,12,50,13);
  win.clrscr();
  win << "Минск - город-герой!";
}
```

Библиотека constream позволяет также управлять консольными потоками с помощью следующих специальных манипуляторов:

Манипулятор	Действие
clreol	очищает до конца строки;
setcrstype(int)	устанавливает вид курсора;
setclr(int)	устанавливает цвет символов;
setbk(int)	устанавливает цвет фона;
setattr(int)	устанавливает байт атрибута;
highvideo	устанавливает повышенную яркость символов;
lowvideo	устанавливает пониженную яркость символов;
normvideo	устанавливает нормальную яркость символов;
setxy(int, int)	перемещает курсор в заданную позицию;
delline	удаляет строку;
insline	вставляет строку;

Следующий пример демонстрирует использование консольных функций и манипуляторов :

```
#include <constrea.h>
void main(){
  constream win;
  int left=2, top=2, right=79, bottom=24;
  win.window(left,top,right,bottom);
  win.clrscr();
  textattr((BLUE << 4) | YELLOW);
  for (int i=1; i<=(right-left+1)*(bottom-top+1); i++)
     win << " ";
  win << setxy(30,3) << "СТОЛИЦЫ ГОСУДАРСТВ:";
  constream win1;
  int left1=5, top1=7, right1=28, bottom1=13;
  win1.window(left1,top1,right1,bottom1);
  win1.clrscr();
  textattr((GREEN << 4) | WHITE);
```

```
  for (i=1; i<=(right1-left1+1)*(bottom1-top1+1); i++)
     win1 << " ";
  win1 << setxy(8,2) << "ГОРОДА" << setclr(MAGENTA)
     <<  setxy(10,4)  <<  "Минск"
     << setxy(10,5)  <<  "Москва"
     << setxy(10,6) << "Киев";
  constream win2;
  int left2=45, top2=7, right2=68, bottom2=13;
  win2.window(left2,top2,right2,bottom2);
  win2.clrscr();
  textattr((GREEN << 4) | WHITE);
  for (i=1; i<=(right2-left2+1)*(bottom2-top2+1); i++)
      win2 << " ";
  win2 << setxy(8,2) << "ГОСУДАРСТВА" << setclr(MAGENTA)
    << setxy(5,4) << "Республика Беларусь"
    << setxy(5,5) << "Россия"
    << setxy(5,6) << "Украина";
}
```

Лекция 11. Технология программирования. Понятие программного обеспечения. Жизненный цикл программы. Модели жизненного цикла ПО.

Понятие программного обеспечения. Понятие "программное обеспечение" вошло в жизнь с развитием компьютерной индустрии. Без него компьютер - это всего лишь электронное устройство, которое не управляется и поэтому не может приносить пользы.

Основной целью программного обеспечения (ПО) является функционирование вычислительной системы. Для понимания сути ПО нужно определить это понятие и его составляющие. ПО - это совокупность взаимосвязанных компьютерных программ, баз данных и документации.
Жизненный цикл программы. Одним из базовых понятий методологии проектирования программ является понятие жизненного цикла программного обеспечения (ЖЦ ПО). ЖЦ ПО - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (ISO - International Organization of Standardization - Международная организация по стандартизации, IEC - International Electrotechnical Commission - Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.
Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:

- основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование баз данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО. Верификация - это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разра-

ботки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании сложных проектов, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ISO 12207-2.
Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Модели жизненного цикла ПО

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ:

- каскадная модель (70-85 г.г.);
- спиральная модель (86-90 г.г.).

В изначально существовавших однородных программах каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 10.1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.
Положительные стороны применения каскадного подхода заключаются в следующем:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.
- сроки завершения всех работ и соответствующие затраты.

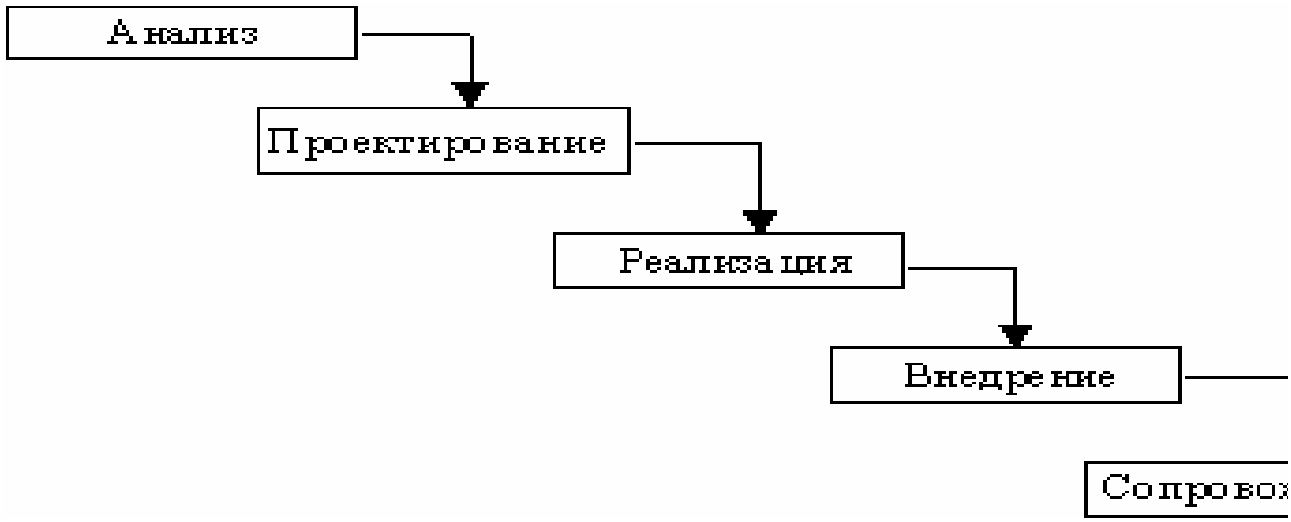

Рис. 10.1. Каскадная схема разработки ПО

Каскадный подход хорошо зарекомендовал себя при построении программ, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рис. 10.2):

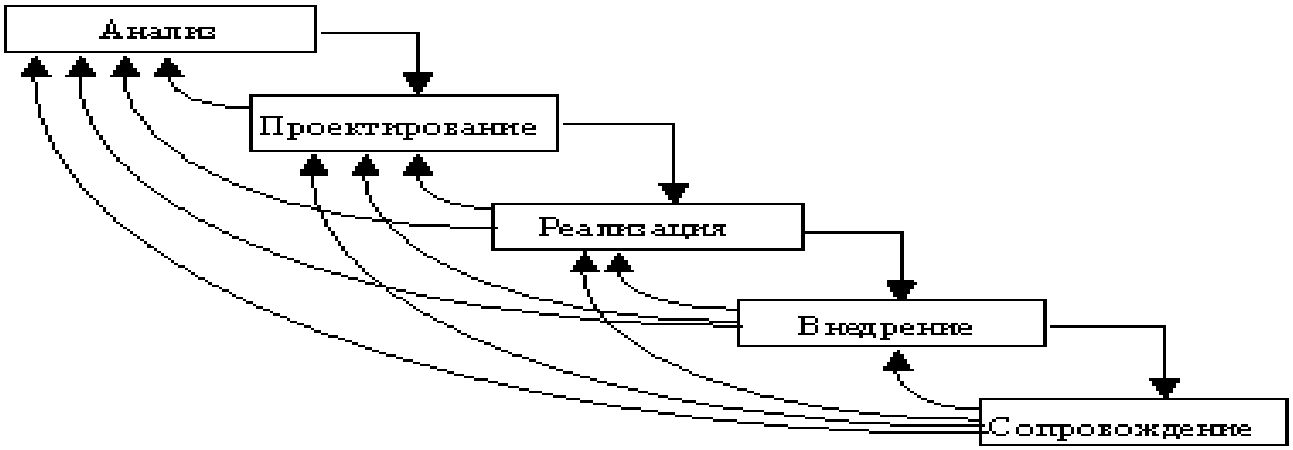

Рис. 10.2. Модификация каскадной схемы разработки ПО

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к программе "заморожены" в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ (рис. 10.3), делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания систе-

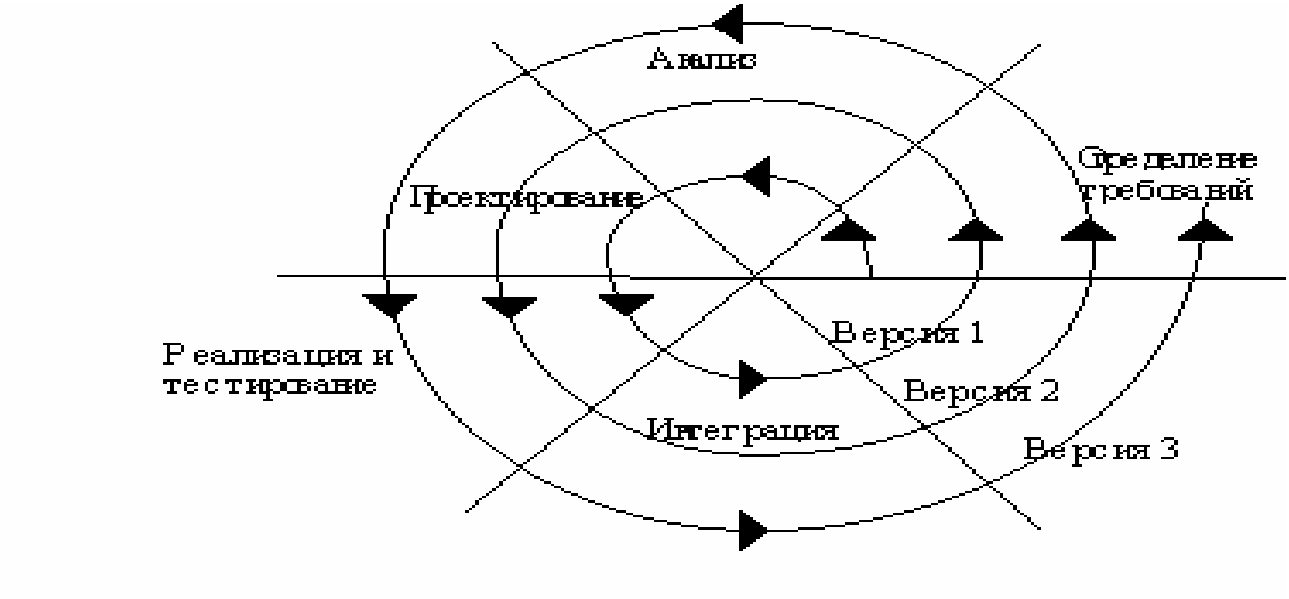

Рис 10.3. Спиральная модель ЖЦ

мы. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Литература.

1.Бьерн Страуструп. Язык программирование Си++. - М.: Бином, 2005.

2. Айра Пол. Объектно-ориентированное программирование на С++. - СПб.:, М.: Невский диалект - Бином, 1999.

3. Герберт Шилдт. Самоучитель С++: Пер. с англ. - 3-е изд. - СПб.: БХВ-Петербург, 2003.

4. Буч Г. Объектно-ориентированное проектирование с примерами применения. Пер. с англ.- М.: Конкорд, 1992.

5. Соловьёв В.В. Объектно-ориентированное программирование на языке С++: Методические указания по курсу “Программирование” для студентов специальности 22.01.- Мн.: БГУИР, 1994.

6. Элиас М., Страуструп Б. Справочное руководство по языку С++ с комментариями. - М.: Мир, 1992.

7.Скляров В.А. Язык С++ и объектно-ориентированное программирование: Справ. пособие.- Мн.: Вышэйшая школа, 1997.