Лекция-1.

Введение.
Предпосылки развития ООП. Ключевые понятия ООП. Язык программирования С++.

Свое развитие С++ получил на базе популярного и мощного языка С, который в свою очередь появился на основе 2-ух языков: BCPL и B. Язык BCPL был создан в 1967 году и предназначался для написания компиляторов и операционных систем. Создателем языка BCPL считается Мартин Ричард. Язык программирования B был разработан Кеном Томсоном из Bell Laboratories в 1970 году и представлял собой улучшенную копию языка BCPL. Язык B использовался для написания ранних версий ОС UNIX для компьютеров DEC PDP-7. Как логическое продолжение двух предшествующих языков на свет появился язык С, который представлял собой старый язык B с добавленными многими новыми типами данных и других усовершенствований. Разработка языка С принадлежит Деннису Ритчи, и впервые был реализован для компьютеров DEC PDP-11 в 1972 году. После этого С стал все шире применяться для написания операционной системы UNIX.

Собственно язык С++ был разработан Бьярном Стаустропом в начале 80-х в той же Bell Laboratories. В отличие от предшественника С С++ обладает более гибким синтаксисом и, что самое важное, обеспечивает возможность объектно-ориентированного программирования. Сама идея ООП впервые была заложена в языке SmallTalk, в котором практически все являлось объектом. В отличие от SmallTalk С++ предусматривает программирования как в привычном структурном стиле С языка так и комбинировать программирование с ООП.

Рассмотрим более подробно ключевые понятия ООП.

Все структурные языки программирования предусматривают простую схему взаимодействия исполняемого кода и данных. Данные группируются с использованием доступных возможностей языка программирования, как в С массивы, структуры, объединения, отдельные переменные различных типов.

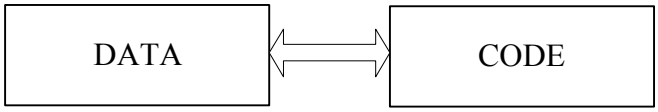


Рис.1.

Для обработки и взаимодействия кода с данными применяются функции (в С). Для взаимодействия с различными типами и структурами данных необходимо использовать различные функции, что в свою очередь затрудняет написание больших проектов, в которых используются множество структур данных. В объектно-ориентированной технологии программирования ключевым понятием взаимодействия кода и данных является ***объект***. Под ***объектом*** будем понимать некую структуру, которая включает в себя данные и код, который взаимодействует с этими данными. Структурно это можно представить следующим образом:

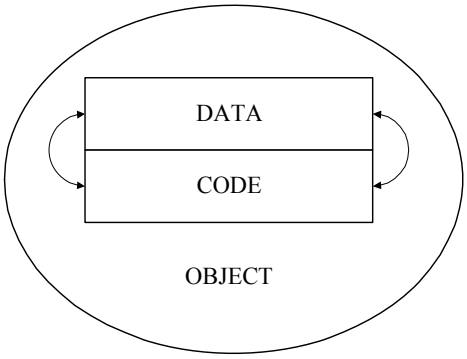


Рис.2.

В языке С++ для создания объектов используют следующие структуры данных, называемые классами (***class***). Данная структура описывается почти так же как и знакомые из С стуктуры и смеси.

Рассмотрим пример описания классов:

```
class <имя класса>
{
  [описание данных]
  [описание кода]
};
```

В отличие от структур и смесей классы содержат не только данные, но и код управляющий этими данными. После определения подобной структуры можно определять конкретные экземпляры классов, которые и представляют собой ***объекты***:

```
<имя класса> <объект 1>,<объект 2>,...,<объект n>;
```

Как мы увидим дальше описывать подобные шаблоны объектов в C++ можно не только с помощью классов, но также с помощью структур и смесей.

Теперь, когда мы познакомились с понятие объекта и методами его описания в C++ перейдем к рассмотрению основных понятий ООП.

Первым понятием ООП является ***пакетирование данных***, иначе называемое ***ENCAPSULATION*** - это механизм языка программирования, который позволяет объединять данные и код, взаимодействующий с этими данными. Описанный класс представляет собой новый тип данных. Экземпляр (элемент) нового типа представляет собой объект. Часть данных или кода может быть защищена от воздействия и использования вне описания объекта (класса).

Наследование (***INHERITANCE***) - описав один класс можно на его базе создать другой, который может приобретать свойства первого. В этом случае первый класс называется ***базовым*** классом, второй - ***производным***. При таком механизме производный класс может наследовать как некоторые свойства базового класса, так и обладать своими уникальными свойствами.

Рассмотрим пример иерархической структуры наследования объектов. Базовым классом является класс описывающий точку. Данный класс может содержать в себе такие данные как координаты текущей точки и цвет, а так же и код, который выводит точку на экране.

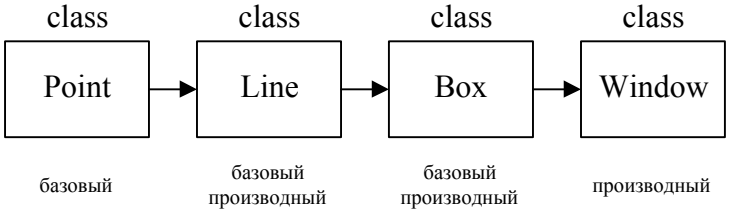


Рис.3.

Затем можно определить производный класс, компонентами которого, к примеру могут быть два объекта типа точка и некоторый код, отвечающий за перемещение и копирование точек. На основе имеющихся двух классов можно строить некоторые графические примитивы типа прямоугольников и т.д.

Полиморфизм (***POLIMORPHISM***) - это свойство, позволяющее использовать одно и тоже имя для обработки различных типов данных. Так например, в языке C, для реализации функций осуществляющих суммирование элементов разнородных массивов необходимо определить несколько функций с различными именами:

```
void Summ1(*char, int);
void Summ2(*int);
void Summ3(*long);
```

В C++ каждая из этих функций может иметь одно и то же имя Summ.

```
char str[10]; int array1[10]; long array2[10];
Summ(str,10);
Summ(array1);
Summ(array2);
```

Как мы увидим в дальнейшем свойство полиморфизма в C++ применимо не только к функциям но и к операторам. Действия, обеспечивающие свойства полиморфизма, называют соответственно перегрузкой функций и перегрузкой операторов.

Перейдем непосредственно к рассмотрению самого языка программирования C++. Рассмотрим характерные особенности и отличия от обыкновенного C.

```
/*Пример программы на C*/
#include <stdio.h>
void main(void)
{ fprintf(stdout,"C language. Example\n"; }
```

Самым первым свойством, которым обладает C++ по сравнению с обычным C является консольный ввод/вывод. В языке C это реализовано с помощью стандартных функций ***printf()*** и ***scanf()***, а так же соответствующих им макросов ***fprintf()*** и ***fscanf()***. В C++ консольный ввод вывод реализован также как и в C на основе стандартных потоков ввода/вывода (stdin/stdout в C) ***cout*** и ***cin*** в C++ и при помощи перегруженных операторов поразрядного сдвига << и >>.

```
//Пример программы на C++
#include <iostream.h>
main()
{ cout<<"C++ language. Example\n"; }
```

Рассмотрим простой пример программы на C++, которая осуществляет консольный ввод и вывод.

```
//Перевод дюймов в сантиметры
#include <iostream.h>
int main()
{
  int inches=0;
  cout<<"Inches=";
```

```
  cin>>inches;
  cout<<inches<<" inches="<<inches*2.54<<" cm\n";
  return 0;
}
```

```
Inches=10
10 inches=25.4 cm
```

В заключение ознакомления с языком C++ отметить некоторые отличия его от языка C.

1. В C++ в отличие от C слово void не является обязательным.
2. В программах C++ все функции должны иметь прототипы.

```
#include <iostream.h>
main()
{ print(); }
print()
{ cout<<"Wrong!\n"; return;]
```

3. Если в C++ функция имеет тип возвращаемого значения отличным от void, то оператор return внутри этой функции должен содержать значение данного типа. В данном случае в C функция возвращает неопределенное значение.
4. В C++ локальные переменные могут быть объявлены в любом месте программы. В C они могут объявляться только в начале блока.

```
#include <iostream.h>
int main()
{
   for(int i=0;i<10;i++) cout<<"i="<<i<<"\n";
   return 0;
}
```

И последний пример использования всех вышеперечисленных свойств программирования на C++:

```
//Вывод массива в обратном порядке
#include <iostream.h>
in();
out();
int array[10];
int main()
{
   in();
   out();
   return 0;
}

in()
{
   for(int i=0;i<10;cin>>array[i],i++);
   return 0;
}
out()
{
   for(int i=9;i>=0;cout<<array[i]<<" ",i--);
   return 0;
}
```

Лекция-2.

Особенности языка C++. Перегрузка функций. Значение формальных параметров по умолчанию.

Продолжаем рассмотрение особенностей программирования на языке C++ начатое на прошлой лекции и сегодня более подробно рассмотрим перегрузку функций.

1.Перегрузка функций.

Перегрузка функций является одной из особенностей ООП, отражающее свойство полиморфизма. ***Перегрузка функций*** (***function overloading***) позволяет определять несколько функций с одним и тем же именем, если эти функции имеют различное количество и(или) типы аргументов. Перегрузка функций обычно используется для создания функций, предназначенных для выполнения однотипных задач, оперирующих с различными структурами и типа

данных. Не являются перегруженными функции, которые имеют одинаковый список аргументов и различные типы возвращаемых значений. При вызове перегруженной функции компилятор определяет адрес вызова нужной функции путем анализа количества, типа и порядка следования аргументов функции. Каждая перегруженная функция имеет свой уникальный идентификатор, называемый ***сигнатурой***.

Рассмотрим пример использования перегруженных функций.

```
#include <iostream.h>
print(){cout<<"Nothig\n";}
print(int n){cout<<"Number="<<n<<"\n";}
print(int n1,int n2){cout<<n1<<"+"<<n2<<"="<<n1+n2<<"\n";}

main()
{
   print();
   print(5);
   print(4,6);
}
```

```
Nothig
Number=5
4+6=10
```

Если в рассмотренном примере определить следующую функцию

```
int print(int n1, int n2){cout<<n1+n2<<"\n";return(n1+n2);}
```

то такая функция не является перегруженной и компилятор выдаст сообщение о синтаксической ошибке, что тело функции **print(int,int)** уже определено.

1.2. Использование аргументов по умолчанию.

Данная особенность перегрузки функций позволяет задавать аргументам значения по умолчанию, в случае если аргумент не задан при вызове функции. Для обеспечения работы такого механизма передачи значений аргументам функциям обеспечивается следующим синтаксисом:

```
возвращаемый_тип имя_функции (тип_аргумента имя_аргумента=значение, ....)
int function(int a=0, int b=1, int c=2);
```

Данный синтаксис напоминает инициализацию переменных. Рассмотрим пример с использованием подобных функций и способы вызова таких функций.

```
#include <iostream.h>
add(int a=0,int b=0,int c=0)
{ cout<<"Summ="<<a+b+c<<": "<<a<<" "<<b<<" "<<c<<"\n";}
main()
{
   add(1,2,3);
   add(1,2);
   add(1);
   add();

}
```

```
Summ=6: 1 2 3
Summ=3: 1 2 0
Summ=1: 1 0 0
Summ=0: 0 0 0
```

Как видно из примера нельзя задать значение по умолчанию аргументу **a** и присвоить какое-либо значение аргументу **b**. При описании функции аргументы по умолчанию задаются лишь один раз - или в описании прототипа функции, либо в определении функции. Обязательно определение подобных функций должно следовать до первого вызова функции. В противном случае компилятор выдаст сообщение об ошибке, что в вызове функции не хватает аргументов.

```
add(int a,int b,int c)
main()
{
   add(1,2,3);
   add(1,2);     //ERROR!!! To few parameters in call
   add(1);       //ERROR!!! To few parameters in call
   add();        //ERROR!!! To few parameters in call
}
add(int a=0,int b-0,int c=0)
{....}
```

Если в определении функции хотя бы один аргумент описан как аргумент по умолчанию, то

все аргументы, стоящие правее него, так же должны быть описаны по умолчанию. Это правило распространяется и на вызовы подобных функций. Если при вызове один аргумент был задан по умолчанию, то все остальные аргументы нельзя задавать явно.

При использовании перегрузки функций и задание аргументов по умолчанию могут возникать ситуации ***неоднозначности***, при которых компилятору не удается выбрать правильную функцию. Рассмотрим пример иллюстрирующий ситуацию неоднозначности.

```
#include <iostream.h>
add(int n){cout<<n<<"\n";}
add(int n1,int n2=0){cout<<n1+n2<<"\n";}
main()
{
   add(2,3);
   add(5);  //Error!!!
}
```

Описание подобных перегруженных функций не вызывает никаких неоднозначностей. Однако на стадии компиляции возникнет проблема, какую версию перегруженной функции ***add()*** необходимо подставить при вызове ***add(5)*** - первую или вторую, при использовании второго аргумента по умолчанию?

Заканчивая рассмотрение перегрузки функций приведем пример, который определяет адрес перегруженной функции.

```
#include <iostream.h>
print(char *str){cout<<"String="<<str<<"\n";}
print(char chr){cout<<"Character="<<chr<<"\n";}
print(int n){cout<<"Number="<<n<<"\n";}

//Указатели на соответствующие типы функций
(*f1)(char*);
(*f2)(char);
(*f3)(int);
main()
{
   f1=print; f2=print; f3=print;
   print("Hello");
   print('A');
   print(10);
   cout<<f1<<"\n"<<f2<<"\n"<<f3<<"\n";
}
```

```
String=Hello
Character=A
Number=10
0x02c2
0x02e7
0x030d
```

2. Встраиваемые функции.

Встраиваемые функции в языке C++ задаются с помощью спецификатора ***inline***. Механизм использования таких функций похож на использование макроопределений с параметрами в языке C. Если функция имеет спецификатор ***inline***, то тело такой функции встраивается в блок программы после места ее вызова. Преимуществом таких функций является быстродействие, заключающееся в экономии времени процессора на вызов функции и обеспечения механизма возврата из нее. Рассмотри пример использования функции со спецификатором ***inline***.

```
#include <iostream.h>
inline print(){cout<<"Inline function\n";}
print(char *str){cout<<str<<"\n";}
main()
{
  print();
  print("Ordinary function\n");
}
```

```
Inline function
Ordinary function
```

Обязательное правило использование inline функций заключается в том, что ее тело должно быть описано до первого вызова этой функции. В противном случае при компиляции возникнет затруднение с тем, какой код встраивать в программу. Отметим ситуации, при которых inline функции не генерируются компилятором:

1. Если в функции присутствуют циклы, операторы switch, goto, return.

2. Для рекурсивных функций и функций содержащих статические переменные (static).
3. Для функций, которые не возвращают значения, но имеют оператор return.
И последнее, встраиваемые функции можно перегружать.

3. Локальные и глобальные переменные.

Как было отмечено на предыдущей лекции переменные можно объявлять в любом месте программы. Более того, при объявлении переменных им можно задавать конкретные инициализирующие значения. Такое свойство языка программирования называется ***динамической инициализацией***. Не редко в программах возникают ситуации, когда имя глобальной переменной совпадает с именем локальной переменно, объявленной внутри функции или блока. Для того, чтобы получить доступ из блока, где объявлена локальная переменная, к глобальной с таким же именем, в C++ применяется оператор "***области видимости***" (***scope resolution operator***) - "::".

Рассмотрим пример использования данного оператора.

```
#include <iostream.h>
int i=0; //глобальная переменная
function()
{
   int i=1; //локальная переменная 1
   {
      int i=2; //локальная переменная 2
      ::i++;
      i++;
      cout<<"Global = "<<::i<<"\n";
      cout<<"Local 2 = "<<i<<"\n";
   }
   i++;
   cout<<"Local 1 = "<<i<<"\n";
}

main()
{
   function();
}
```

```
Global = 1
Local 2 = 3
Local 1 = 2
```

Все локальные переменные являются переменными с локальным временем жизни. Это означает, что доступ к локальным переменным возможен лишь внутри функции или блока, где они объявлены. Вне пределов блока или функции локальные переменные недоступны. Все локальные переменные имеют по умолчанию атрибут ***auto***. При входе в блок или функцию выделяется память под такие переменные, а после выхода из блока - они исчезают. Любую переменную или функцию можно задать с таким атрибутом, который обеспечивает глобальное времени жизни. Такие переменные называются статическими и имеют атрибут ***static***. Такие переменные начинают свое существование с самого начала работы программы. Если программист явно не проинициализировал статическую переменную, то компилятор по умолчанию присваивает ей значение 0. Локальные статические переменные имеют область видимости лишь в той функции или блоке, где они объявлены, но в отличие от обычных локальных переменных, они сохраняют свои значения на протяжении всего жизненного цикла программы. Рассмотрим пример, иллюстрирующий все вышесказанное.

```
#include <iostream.h>
function()
{
   static int n=0;
   n++;
   cout<<n<<" start of function\n";

}
main()
{
   function();
   function();
   function();
}
```

```
1 start of function
2 start of function
3 start of function
```

4. Выделение динамической памяти.

В стандарте языка С для выделения и освобождения динамической памяти использовались функции ***malloc()*** и ***free()*** соответственно из стандартной библиотеки <stdlib.h>. В языке С++ для выделения и освобождения динамической памяти используются следующие два оператора ***new*** и ***delete***. Данные операторы являются стандартом языка С++ и не требуют подключение какой-либо библиотеки. Синтаксис операций выделения и освобождения памяти для одной переменной определенного типа выглядит следующим образом:

```
Указатель_на_тип = new тип;
delete указатель_на_тип;
```

Если по каким-либо причинам оператор new не может выделить память, к примеру недостаточно свободной памяти, то оператор возвратить указатель на ноль. В противном случае возвращается указатель на выделенный блок памяти. Оператор new автоматически выделяет требуемое количество памяти для хранения объекта заданного типа, и автоматически возвращает указатель на заданный тип. При этом нет необходимости осуществлять операцию по привидению типов, как это было в С. Еще одним достоинством данных операторов является тот факт, что при выделении памяти под переменные можно автоматически производить их инициализацию, а так же выделять память под массивы и структуры данных - в том числе и объектов. Синтаксис операций выделения и освобождения памяти под одномерные массивы выглядит следующим образом:

```
указатель_на_тип = new тип [размерность];
delete [] указатель_на_тип;
```

При выделении памяти под многомерные массивы в версии языка С++ фирмы Borland Int., можно воспользоваться следующим синтаксисом:

```
#include <iostream.h>
main()
{
   int *var=new int (10);
   cout<<*var<<"\n";
   int *ptr=new int [5,5];
   for(int i=0;i<5;i++)
   {
      for(int j=0;j<5;j++)
      {
         if(i==j) ptr[i,j]=i;
         else ptr[i,j]=0;
         cout<<ptr[i,j];
      }
      cout<<"\n";
   }
   delete var;
   delete [] ptr;
}
```

```
10
00000
01000
00200
00030
00004
```

И последнее замечание, в программах не рекомендуется использовать смешанные функции по управлению динамической памятью. Так, если память была выделена при помощи оператора ***new***, то освобождать такую память необходимо только при помощи оператора ***delete***, а не функцией ***free()***.

5. Ссылки.

В большинстве языков программирования аргументы в функциях передаются либо при помощи ссылки (by reference) или по значению (by value). В первом случае функция работает непосредственно с переменной, во втором только с ее значением. Различия здесь очевидны - переменную, переданную по ссылке функция может модифицировать, а переданную по значению - нет. В стандартном языке С параметры в функции передаются только по значению, и для того чтобы модифицировать ее внутри функции, необходимо в качестве параметра передать указатель на переменную:

```
inc(int *var)
{
   (*var)++;
}
main()
{
   int n=0;
   inc(&n);
}
```

В приведенном примере, для изменения переменной n в функции inc необходимо передать в нее указатель на переменную, а при вызове функции указать, что в качестве аргумента в функцию передается адрес переменной n. При использовании ссылок в языке C++ нет необходимости в передаче адреса переменной в функции и в использовании оператора * для модификации переменной внутри тела функции. Рассмотрим этот же пример, но только с использованием ссылки.

```
inc(int &var)
{
   var++;
}
main()
{
   int n=9;
   inc(n);
}
```

Следует отметить, что функции могут в качестве возвращаемого значения использовать ссылку. Рассмотрим пример, при котором функция возвращает ссылку на переменную целого типа:

```
int x;
 int &f(){return x;}
 int main()
 {
    f()=0xff;
    return 0;
 }
```

Необходимо заметить, что оператор return в функции f() возвращает не значение глобальной переменной x, а ссылку (адрес) этой переменной. Дальнейшее действие компилятор воспринимает как занесение по адресу переменной x значение 0xff. Это легко увидеть в тексте данной программы на языке ассемблера:

```
int main();
  mov bp,sp
  call f
  mov bx,ax
  mov word ptr [bx],0x00ff
  xor ax,ax
  pop bp
  ret
```

```
int &f();
  push bp
  mov bp,sp
  mov ax,0x028c
  pop bp
  ret
```

Следует отметить что многие компиляторы запрещают возвращать функциям ссылки на локальные переменные. Теоретически это возможно осуществить, однако использовать такую ссылку невозможно за пределами видимости локальной переменной.

Рассмотрим еще один пример, в котором функция возвращает ссылку на переменную, объявленную внутри функции.

Так же ссылка может быть применена как дубликат какой-либо переменной следующим образом:

```
int n=10;
int &ref=n;
ref++;
```

Лекция-3.

(продолжение)

4. Выделение динамической памяти.

В стандарте языка C для выделения и освобождения динамической памяти использовались функции ***malloc()*** и ***free()*** соответственно из стандартной библиотеки <stdlib.h>. В языке C++ для выделения и освобождения динамической памяти используются следующие два оператора ***new*** и ***delete***. Данные операторы являются стандартом языка C++ и не требуют подключение какой-либо библиотеки. Синтаксис операций выделения и освобождения памяти для одной переменной определенного типа выглядит следующим образом:

```
указатель_на_тип = new тип;
delete указатель_на_тип;
```

Если по каким-либо причинам оператор new не может выделить память, к примеру недостаточно свободной памяти, то оператор возвратить указатель на ноль. В противном случае возвращается указатель на выделенный блок памяти. Оператор new автоматически выделяет требуемое количество памяти для хранения объекта заданного типа, и автоматически возвращает указатель на заданный тип. При этом нет необходимости осуществлять операцию по привидению типов, как это было в C. Еще одним достоинством данных операторов является тот факт, что при выделении памяти под переменные можно автоматически производить их инициализацию, а так же выделять память

под массивы и структуры данных - в том числе и объектов. Синтаксис операций выделения и освобождения памяти под одномерные массивы выглядит следующим образом:

```
указатель_на_тип = new тип [размерность];
delete [] указатель_на_тип;
```

При выделении памяти под многомерные массивы в версии языка C++ фирмы Borland Int., можно воспользоваться следующим синтаксисом:

```
#include <iostream.h>
main()
{
   int *var=new int (10);
   cout<<*var<<"\n";
   int *ptr=new int [5,5];
   for(int i=0;i<5;i++)
   {
      for(int j=0;j<5;j++)
      {
         if(i==j) ptr[i,j]=i;
         else ptr[i,j]=0;
         cout<<ptr[i,j];
      }
      cout<<"\n";
   }
   delete var;
   delete [] ptr;
}
```

```
10
00000
01000
00200
00030
00004
```

И последнее замечание, в программах не рекомендуется использовать смешанные функции по управлению динамической памятью. Так, если память была выделена при помощи оператора ***new***, то освобождать такую память необходимо только при помощи оператора ***delete***, а не функцией ***free()***.

5. Ссылки.

В большинстве языков программирования аргументы в функциях передаются либо при помощи ссылки (by reference) или по значению (by value). В первом случае функция работает непосредственно с переменной, во втором только с ее значением. Различия здесь очевидны - переменную, переданную по ссылке функция может модифицировать, а переданную по значению - нет. В стандартном языке C параметры в функции передаются только по значению, и для того чтобы модифицировать ее внутри функции, необходимо в качестве параметра передать указатель на переменную:

```
inc(int *var)
{
   (*var)++;
}
main()
{
   int n=0;
   inc(&n);
}
```

В приведенном примере, для изменения переменной n в функции inc необходимо передать в нее указатель на переменную, а при вызове функции указать, что в качестве аргумента в функцию передается адрес переменной n. При использовании ссылок в языке C++ нет необходимости в передаче адреса переменной в функции и в использовании оператора * для модификации переменной внутри тела функции. Рассмотрим этот же пример, но только с использованием ссылки.

```
Inc(int &var)
{
   var++;
}
main()
{
```

```
   int n=9;
   inc(n);
}
```

Следует отметить, что функции могут в качестве возвращаемого значения использовать ссылку. Рассмотрим пример, при котором функция возвращает ссылку на переменную целого типа:

```
int x;
 int &f(){return x;}
 int main()
 {
    f()=0xff;
    return 0;
 }
```

Необходимо заметить, что оператор return в функции f() возвращает не значение глобальной переменной x, а ссылку (адрес) этой переменной. Дальнейшее действие компилятор воспринимает как занесение по адресу переменной x значение 0xff. Это легко увидеть в тексте данной программы на языке ассемблера:

```
int main();
  mov bp,sp
  call f
  mov bx,ax
  mov word ptr [bx],0x00ff
  xor ax,ax
  pop bp
  ret
```

```
int &f();
  push bp
  mov bp,sp
  mov ax,0x028c
  pop bp
  ret
```

Следует отметить что многие компиляторы запрещают возвращать функциям ссылки на локальные переменные. Теоретически это возможно осуществить, однако использовать такую ссылку невозможно за пределами видимости локальной переменной.

Рассмотрим еще один пример, в котором функция возвращает ссылку на переменную, объявленную внутри функции.

```
#include <iostream.h>
int& function()
{
   static int n=5;
   cout<<"n="<<n<<"\n";
   return n;
}
main()
{
   int &ref=function();
   cout<<"ref="<<ref<<"\n";
   ref++;
   ref=function();
   cout<<"ref="<<ref<<"\n";
   return 0;
}
```

```
n=5
ref=5
n=6
ref=6
```

Так же ссылка может быть применена как дубликат какой-либо переменной следующим образом:

```
int n=10;
int &ref=n;
ref++;
```

Однако использовать данный механизм не рекомендуется в виду того, что может возникать путаница с именами переменных, и т.д.

6.Указатель на VOID

В языке C++ указатель на любой тип можно присваивать другому указателю такого же типа. В противном случае необходимо использовать процедуру приведения типов.

Лекция-4:

2. Объекты.

С этой лекции мы начинаем углубленное изучение объектов. Начнем это изучение с рассмотрения одного из основных понятий ООП - ***инкапсуляции*** (***пакетирование данных***). Мы с вами отмечали, что объект представляет собой структуру, которая содержит некоторые данные и код, который взаимодействует непосредственно с этими данными. Такие подобные структуры в C++ можно описывать с помощью структур (struct), смесей (union) и классов (class). Подобные описания кода и данных в C++ представляют собой новый тип данных. Конкретные экземпляры таких типов - и есть ***объекты***. В ходе рассмотрения объектов мы с вами изучим следующие положения: способы описания объектов, возможности осуществления доступа к данным и коду внутри объектов, способы наследования одних объектов свойств других, и т.д.

Начнем наше знакомство с рассмотрения способа описания объектов с помощью классов.

2.1. Описание объектов при помощи классов (class).

Описание объектов при помощи классов очень напоминают запись структур и смесей в языке C. Определение класса начинается с ключевого слова class. Данные и функции одного класса заключаются в фигурные скобки ({}). Заканчивается определение символом ;

```
class имя_класса
{
  //функции
  //данные
};
```

Каждая функция и элемент данных, описанные внутри класса имеет атрибут доступа. В языке C++ существуют три атрибута доступа:

1. ***public*** - открытый (общий). Доступ к элементам класса может быть осуществлен из самого класса так и извне класса.
2. ***private*** - закрытый (частный). Доступ к элементам класса может быть осуществлен только из функций самого класса. Однако как мы увидим позднее доступ к элементам с атрибутом private можно осуществлять и не только внутри класса.
3. ***protected*** - защищенный. Также действие данного атрибута доступа мы рассмотрим позднее.

По умолчанию все элементы класса имеют атрибут доступа ***private***. Атрибут задается ключевым словом и символом (:). Действие атрибута сохраняется до следующего атрибута или до закрывающей фигурной скобки в объявлении класса.

```
class C
{
  int X;
  public:
  int function();
  private:
  int Y;
};
```

Для доступа к элементам класса используется символ (.)

```
C obj;
obj.X; obj.Y; //неправильно!!!
int n=obj.function;
```

или символы (->) если используется не имя объекта а указатель.

```
C *obj;
obj->X; obj->Y;
int n=obj->function;
```

Описание функций (***методов***) класса можно проводить как внутри описания класса, так и за его пределами. В первом случае компилятор воспримет функцию как inline функцию, во втором - как обычную.

```
class C
{
  public:
  void f1(){cout<<"Inline function\n";}
  void f2();
};
void C::f2()
{ cout<<"Ordinary function\n";}
main()
{
  C obj;
  obj.f1(); obj.f2();
}
```

Для описания функции за пределами класса, внутри тела класса описывается только прототип функции. Для описания тела такой функции за пределами класса необходимо использовать механизм привязки функции к конкретному классу при помощи оператора области видимости (::). Такой метод описания функций класса весьма удобен, т.к. в противном случае код всех функций будет расположен внутри создаваемых объектов, что является не очень хорошим стилем программирования, хотя и повышает скорость выполняемого кода.

Использование методов внутри класса осуществляется просто при помощи имени метода, за пределами класса - через имя объекта, которому принадлежит метод. Методы класса могут быть перегружены, но только в пределах описания класса.

```
#include <iostream.h>
void f(int n){cout<<"Global function\n";}
class C
{
public:
      void f(){::f(0);cout<<"Inline \n";}
      void f(int);
};
void C::f(int n)
{     cout<<"Number="<<n<<"\n";}
main()
{
      C obj;
      obj.f(); obj.f(5);
}
```

Для вызова глобальной перегруженной функции необходимо использовать оператор области видимости (::). В остальных случаях вызываются перегруженные функции, являющиеся элементами данного класса.

2.2. Конструкторы и деструкторы.

Замечание по описанию тела класса: внутри класса нельзя явно инициализировать переменные класса. Инициализацию можно проводит вне пределов класса, если переменная является общедоступной, или при помощи функций класса. Однако для инициализации переменных предусмотрен специальный механизм использования функций-конструкторов (или просто - ***конструкторов***). Данная функция является элементом класса и вызывается всякий раз при создании объекта данного класса. Это означает, что если внутри конструктора расположены операции по инициализации каких-либо переменных, то такая инициализация будет происходить автоматически каждый раз при создании новых объектов. Конструктор может быть описан как обычная функция класса, т.е. внутри и вне тела класса. Имя конструктора всегда совпадает с именем класса. Конструктор не может возвращать какое-либо значение. Конструктор может быть перегружен.

```
#include <iostream.h>
class C
{
      char *str;
      int num;
public:
      C(){cout<<"Constructor\n";str=0;num=0;}
      C(char*);
      C(int);
      void print();
};
C::C(char *s) {str=s; num=0; cout<<"String init.\n";}
C::C(int n) {num=n; str="Nothing";cout<<"Number init.\n";}
void C::print()
{ cout<<"String="<<str<<" Number="<<num<<"\n";}

main()
{
      C obj1, obj2("Object 2"), obj3(3);
      obj1.print();
      obj2.print();
      obj3.print();
}
```

```
Constructor
String init.
Number init.
```

```
String= Number=0
String=Object 2 Number=0
String=Nothing Number=3
```

Также конструктор может быть задан с аргументами по умолчанию. Использование таких конструкторов аналогично использованию функций с аргументами по умолчанию.

```
#include <iostream.h>
class C
{
	char *str;
	int num;
public:
	C(char *s="Nothing",int n=0);
	void print();
};
C::C(char *s,int n)
{
	str=s; num=n;
}
void C::print()
{ cout<<"String="<<str<<" Number="<<num<<"\n";}

main()
{
	C obj1, obj2("Object 2"), obj3("Object 3",3);
	obj1.print();	obj2.print();	obj3.print();
}
```

```
String=Nothing Number=0
String=Object 2 Number=0
String=Object 3 Number=3
```

Обратной смысл конструктору принадлежит специальной функции класса - ***деструктору***, который если объявлен внутри класса будет вызываться всякий раз при уничтожении объекта (окончание программы, выход из области видимости объекта, освобождение памяти, выделенной под объект). Имя деструктора, как и конструктора, совпадает с именем класса, но перед именем деструктора ставиться специальный символ - тильда (~). Как и конструктор деструктор не может возвращать значения, а также в деструктор нельзя передавать аргументы (передача аргументов удаляемому объекту запрещена). Перегрузка деструкторов - невозможна. В общем случае деструктор не выполняет действия, связанные с освобождением памяти под имеющийся объект, а предшествует этому освобождению. В большинстве случаев в деструкторе осуществляются действия связанные с освобождением динамической памяти, которая была выделена некоторым структурам данных, описанных внутри класса. Конструкторы и деструктор должны иметь атрибут доступа public, в противном случае объект не будет создан или уничтожен.

```
#include <iostream.h>
class C
{
public:
	C(){cout<<"Constructor\n";}
	~C(){cout<<"Destructor\n";}
};
main()
{ C obj;}
```

Рассмотрим случаи, при которых вызываются конструкторы и деструкторы объектов.

```
#include <iostream.h>
class C
{
	int id;
public:
	C(int n){id=n; cout<<"Constructor "<<id<<"\n";}
	~C(){cout<<"Destructor "<<id<<"\n";}
} obj1(1);

function(){ static C obj5(5);}

main()
```

```
{
  cout<<"BEGIN\n";
  C obj2(2);
      {
            C obj3(3);
      }
  C obj4(4);
  function();
  function();
  function();
  cout<<"END\n";
}
```

```
Constructor 1
BEGIN
Constructor 2
Constructor 3
Destructor 3
Constructor 4
Constructor 5
END
Destructor 4
Destructor 2
Destructor 5
Destructor 1
```

Еще одна особенности деструкторов заключается в том, что они могут быть явно вызваны. Однако это не стоит применять на практике, особенно если деструктор отвечает за принудительную процедуру освобождения динамической памяти. Если в предыдущем примере перед выводом сообщения "END" явно вызвать деструктор объекта obj4 (obj4.~C();), то программа выдаст в стандартный выводной поток дополнительное сообщение перед "END" - "Destructor 4".

Конструкторы глобальных объектов вызываются в первую очередь, до выполнения первого действия в функции main(). Деструкторы глобальных объектов вызываются самыми последними, после того как отработала функция main(). Жизнь локальных объектов, как и локальных переменных ограничивается областью видимости. Конструкторы таких объектов вызываются в месте создания объекта, деструкторы - после выхода из блока (области видимости). Конструкторы статических объектов вызываются один раз при первом достижении команды создания объекта, а деструкторы - после завершения функции main(). В общем случае, если несколько объектов создаются в одном блоке, то конструкторы таких объектов вызываются в порядке создания объектов, а деструкторы - в обратном порядке.

2.3. Конструкторы копирования.

Рассмотрим пример, в котором в качестве аргумента функции передается объект.

```
#include <iostream.h>
class C
{
      public:
      int n;

      C(int number){n=number;cout<<"Constructor\n";}
      ~C(){cout<<"Destructor\n";}
      print(){cout<<"n="<<n<<"\n";}
};

inc(C obj)
{
      obj.n++;
      obj.print();
}

main()
{
      C obj(5);   inc(obj);
      obj.print();
}
```

Результат работы будет неожиданным. Рассмотрим это более детально: при вызове функции inc происходит создание нового временного объекта obj, причем все его свойства копируются от глобального объекта obj, созданного в функции main(). Достигается это за счет того, что не происходит вызов конструктора объекта obj (в функции inc). Однако, после выхода из области видимости в функции inc временный объект obj уничтожается, что приводит к автоматическому вызову деструктора временного объекта. После отработки функции inc функция print() выводит значение переменной n глобального объекта, которое не было изменено. Для того, чтобы осуществлять доступ к внутренним переменным объектов внутри функций, в качестве аргумента необходимо передавать указатель на объект или ссылку на объект. При таком механизме не происходит создание временных объектов, а соответственно и вызов деструкторов. Передача объекта в качестве аргумента функции опасна еще с той точки зрения, что деструктор может освободить туже память, что была выделена для копии данного объекта, что в свою очередь приведет к потери данных. Рассмотрим пример:

```
#include <iostream.h>
class C
{
public:
	int *ptr;
	C(int number) {ptr=new int(number); cout<<"Constructor\n";}
	~C(){delete ptr; cout<<"Destructor\n";}
	print(){cout<<"*ptr="<<(*ptr)<<"\n";}
};
main()
{
	C obj1(5);
	{
		C obj2=obj1;
		obj2.print();
	}
	C obj3(10);
	obj1.print();
}
```

```
Constructor
*ptr=5
Destructor
Constructor
*ptr=10
Destructor
Destructor
```

Первый раз конструктор вызывается для объекта obj1. При создании объекта obj2 конструктор не вызывается, т.к. происходит побитное копирование объекта obj1 в объект obj2. Это означает, что указатели на целое ptr в объектах obj1 и obj 2 равны. При выходе из области видимости объекта obj2 вызывается деструктор, который освобождает память под переменную целого типа, но в этом случае как для временного объекта obj2, так и для объекта obj1. Так получилось, что в данной программе сразу же после уничтожения объекта obj1 происходит автоматическое выделение памяти под объект obj3, который размещается на месте объекта obj1. Для предотвращения таких нежелательных действий, в языке C++ применяется механизм, называемый конструктором копирования. В общем случае конструктор копирования имеет следующий синтаксис:

```
имя_класса (const имя_класса & имя_объекта){....}
```

Теперь, если в предыдущем примере мы опишем следующий конструктор копирования:

```
C(const C &obj){ptr=new int(*obj.ptr);cout<<"Copy constructor \n";}
```

то при явной инициализации локального объекта obj2 произойдет вызов конструктора копирования, при котором выделиться память под целое, а значение в выделенную память скопируется из объекта obj, который присваивается текущему объекту.

Результат работы такой программы будет выглядеть следующим образом:

```
Constructor
Copy constructor
*ptr=5
Destructor
Constructor
*ptr=5
Destructor
Destructor
```

Следует отметить, что конструктор копирования вызывается только при явной инициализации объектов, когда одному объекту присваивается другой. В противном случае компилятор воспринимает оператор (=) как обыкновенную операцию присваивания и производит побитное копирование содержимого одного объекта в другой. Существует еще одна ситуация, при которой вызывается конструктор копирования. Если описана функция, которая возвращает объект, т.е. возвращает временный объект, созданный внутри функции. Рассмотри такой пример:

```
#include <iostream.h>
class C
{     int n;
      public:
      C(int number){n=number; cout<<"Constructor\n";}
      C(const C &obj){n=obj.n; cout<<"Copy constructor\n";}
      ~C(){cout<<"Destructor\n";}
      print(){cout<<"n="<<n<<"\n";}
};

C fnul()
{     cout<<"BEGIN:fnul\n";
      C tmp(0);
      cout<<"END:fnul\n";
      return tmp;
}

main()
{     cout<<"BEGIN:main\n";
      C obj1(5);   obj1.print();  obj1=fnul(); obj1.print();
      cout<<"END:main\n";
}
```

```
BEGIN:main
Constructor
n=5
BEGIN:fnul
Constructor
END:fnul
Copy constructor
Destructor
Destructor
n=0
END:main
Destructor
```

При вызове функции fnul создается временный объект, предназначенный для хранения возвращаемого объекта. Далее, когда происходит инициализация временного объекта локальным объектом tmp внутри функции fnul(), происходит вызов конструктора копирования. После этого происходит последовательный вызов деструкторов временного объекта и объекта tmp. Значение временного объекта побитно копируется в объект obj1 без вызовов конструктора копирования и деструктора.

Лекция-5:

(продолжение)

2.4. Статические переменные внутри класса.

В ANSI стандарте языка C++ нет никаких ограничений на использование статических переменных внутри классов. Однако в реализации компилятора C++ фирмы Borland Int. объявление статических переменных внутри классов приводит к возникновению ошибки на стадии линковки объектного кода. Для предотвращения данного эффекта следует пользоваться следующей записью:

```
#include <iostream.h>
class C
{
  static int n;
  public:

  C(){n++; cout<<"Object #"<<n<<"\n";}
};
int C::n=0;
```

```
main()
{
  C obj1,obj2,oj3;
  return 0;
}
```

Данная запись заключается в определении статической переменной внутри класса и повторного определения этой же переменной за пределами класса с явной инициализацией или без нее. В последнем случае компилятор произведет сам инициализацию этой переменной значением "0".

2.5. Указатели, ссылки и массивы объектов. Инициализация объектов.

Как было отмечено ранее, при описании объектов при помощи классов программистом создается новый тип данных, а переменная этого типа (экземпляр) и есть объект. В связи с этим, если рассматривать класс как новый тип данных, то в C++ можно создавать массивы объектов. Синтаксис создания массивов объектов аналогичен синтаксису создания обыкновенных объектов в C++. Если задан произвольный класс C, то для того чтобы создать одномерный массив объектов, можно воспользоваться следующей записью:

```
class C {...};
main()
{
  C objs[4];
}
```

Данная запись относиться к тому случаю, если в классе описан конструктор с аргументами по умолчанию или конструктор с одним аргументом void. Иначе дело обстоит при создании массивов объектов если описан только конструктор с аргументами. В этом случае необходимо при описании массива явно указывать передаваемые параметры. Данная запись выглядит следующим образом:

```
#include <iostream.h>
class C
{
      int n;
      public:
      C(int nn){n=nn; cout<<"Constructor "<<n<<" \n";}
      ~C(){cout<<"Destructor\n";}
      print(){cout<<"n="<<n<<"\n";}
};
main()
{
      C objs[4]={C(1),C(2),C(3),C(4)};
      C[4].print();
      return 0;
}
```

Если конструктор имеет всего один аргумент, то данная запись может быть сокращена без использования явного вызова конструктора - обычным перечислением аргументов через запятую. Доступ к открытым членам класса при описании массивов объектов производиться через явное указание элемента массива (объекта) и оператора точка (.).

Можно создавать указатели и ссылки на объекты. Если в программе объявлен указатель на объект, то такой указатель можно применять для доступа к компонентам реальных объектов, только в том случае если имена классов, используемые для описания указателя и объекта, совпадают. Так, если в предыдущем примере описать указатель на класс C, то можно использовать следующую запись для вызова метода print() третьего объекта в массиве:

```
C *ptr;
 ptr=&objs[2];
 ptr->print();
```

Для доступа к компонентам по указателю на класс используется оператор (->). Если был объявлен указатель на класс, то при помощи данного указателя можно выделить динамическую память под объект.

```
C *ptr;
 ptr=new C;
 ptr->print();
 delete ptr;
```

Если конструктор имеет аргументы, то можно при выделении памяти под объект производить динамическую инициализацию путем передачи аргументов конструктору.

```
C *ptr;
 ptr=new C (1,2);
 ptr->print();
 delete ptr;
```

Можно выделят при помощи операторов и массивы объектов, однако, следует отметить, что невозможно осуществлять динамическую инициализацию массивов объектов. При удалении объекта при помощи оператора delete происходит автоматический вызов деструктора.

В C++ можно объявлять и ссылки на объекты. Мы с вами сталкивались на прошлой лекции с малоприятными случаями, когда в функцию в качестве параметра передается объект, и когда деструктор содержит оператор освобождения динамической памяти. В этих случаях приходилось применять конструкторы копирования. Однако более простой способ обходить такие неприятные ситуации заключается в использовании ссылок на объекты. Напоминаем, что при использовании ссылок не происходит создания копии объектов, а соответственно не происходит вызов деструкторов копии. При использовании ссылок на объекты в качестве аргументов функций встречаются ситуации, при которых нежелательны какие-либо изменения открытых компонентов классов. Для этого при описании ссылки ставиться спецификатор const, говорящий о том, что компоненты данного объекта изменить нельзя.

```
#include <iostream.h>
class C
{     public:
       int n;
       C(int nn){n=nn}
       print(){cout<<"n="<<n<<"\n";return 0;}
};
function(const C &obj)
{       obj.n++; //ERROR!!!
       obj.print();
}
main()
{ C obj(5); function(obj); }
```

Мы рассмотрели указатели и ссылки на объекты. Однако для каждого экземпляра класса существует специальный уникальный указатель. Например, при создании двух объектов одного и того же класса происходит вызов метода этих объектов. Компилятор выбирает определенную версию метода для конкретного объекта путем использования специального указателя на объект. Такой указатель называется указателем this. Очевидно, что запись *this (разыменование указателя) и есть сам объект. This - сугубо внутренний указатель и применяется лишь методами конкретного класса для доступа к остальным компонентам. Рассмотрим следующий пример:

```
#include <iostream.h>
class C
{
       int n;
       public:
       C(int n) {this->n=n; cout<<"Address="<<this<<"\n";}
       print(){cout<<"n="<<n<<"\n";}
       ~C(){(*this).print();}

};

main()
{
       C obj(15);

       return 0;
}
```

Какое важное значение использования указателя this мы рассмотрим позже - при изучении вопросов касающихся перегрузки операторов.

2.6. Другие способы описания объектов.

Мы ознакомились с одним способом описания объектов в языке C++ - с классами. Однако в C++ можно описывать объекты также при помощи структур и смесей. Применение структур для описания объекта скорее используется для поддержания совместимости языка C и C++ для тех разработчиков, которые переносят свои старые разработки в стиле языка C на C++. Так для любой структуры в C++ можно описать любой метод, в том числе конструктор и деструктор. Одно отличие классов и структур выражается в том, что по умолчанию все компоненты структур имеют атрибут открытый доступа ***public***. Для того чтобы объявить некоторые компоненты структуры закрытыми по отношению к внешним воздействиям необходимо явно задать перед их описанием атрибут доступа ***private***. Описывать объекты в C++ можно также и при помощи смесей. Понятие смеси в C++ близко к определению структур (как и в обычном C), за исключением того, что все компоненты разделяют одну и туже область памяти. Рассмотрим пример использование смесей для определения объектов:

```
#include <iostream.h>
union word
{
  unsigned N;
  unsigned char B[2];
  word(int N){this->N=N;}
  convert(){cout<<(int)B[1]<<" - "<<(int)B[0]<<"\n";}
};
main()
{   word W(356);
  W.convert();
}
```

Результат работы этой программы очевиден. В одной и той же области памяти располагается переменная N (1 слово=2 байтам) и два байта в виде массива char. Следует отметить, что компилятор разделяет общую память внутри union только под переменные, как было показано в предыдущем примере. Если же внутри union описать указатель на внешнюю функцию, то он будет восприниматься как обычная переменная, которая будет разделять с другими общую память. Рассмотрим следующий пример:

```
#include <iostream.h>
union U{
      int i;
      void (*function)(U&);
} u;

void print(U &obj)
{ cout<<obj.i<<"\n"; }

void main()
{
 u.i=10;
 u.function=&print;
 u.function(u);
}
```

Если в предыдущем примере осуществить последовательно следующие две операции

```
u.i=5;
u.function();
```

то это приведет к некорректному вызову функции по адресу 5.
Существуют некоторые ограничения на использование смесей для описания объектов:
1. Смеси не могут наследовать какие-либо объекты.
2. Смеси не могут использоваться в качестве определения базовых объектов для других.
3. Компоненты смесей не могут иметь атрибут static.
4. Смеси не могут содержать операции создания объектов, которые используют конструкторы и деструкторы. Сами смеси могут иметь конструкторы и деструктор.

3. Дружественные функции.

При написании программ часто встречаются ситуации. При которых необходимо иметь по тем или иным причинам доступ к закрытым компонентам классов. При использовании обычных внешних функций это не возможно. На стадии компиляции компилятор выдаст сообщение об ошибке, говорящее что доступ к тому или иному компоненту невозможен. Для решения данной задачи в С++ применяются так называемые дружественные функции. (***friend functions***). Отличительная особенность дружественных функций от обычных заключается в том, что такие функции не являются компонентами классов, однако могут обращаться к закрытым компонентам. В отличие от обычных функций дружественные функции задаются при помощи спецификатора ***friend***, и их прототипы задаются в теле описания класса. Тело дружественной функции может быть описано за пределами описания класса и внутри класса. Но лучше производить описание тела дружественной функции за пределами класса, тем самым подчеркивая тот факт. Что она не является компонентом класса.

```
#include <iostream.h>
class C
{
  int n;
  public:
  C(int n) {this->n=n;}
  print(){cout<<"n="<<n<<"\n";}
  friend add(C&);
};

add(C &obj)
{
  obj.n++;
}
main()
{
  C obj(4);
  obj.print();
  add(obj);
  obj.print();
  return 0;
}
```

Для того, чтобы дружественная функция осуществляла доступ к компонентам класса необходимо в качестве аргумента передавать указатель или ссылку на объект. Объявлять прототип дружественной функции можно в любом месте описания класса. Атрибуты доступа для дружественных функций не играют роли. Так как дружественная функций не является компонентом класса ее нельзя вызвать с использованием имени объекта или указателя на объект. Вызов осуществляется обычным способом. Дружественные функции обычно применяются для доступа к компонентам сразу же нескольких классов. Рассмотрим пример, в котором описана дружественная функция, которая производит сравнение закрытых переменных двух классов.

```
#include <iostream.h>
class B; // ссылка вперед
class A
{
      int i;
      public:
      A(int i){this->i=i;}
      friend char* equal(const A&,const B&);
};

class B
{
      int i;
      public:
      B(int i){this->i=i*i;}
      friend char* equal(const A&,const B&);
};

char* equal(const A &obj1,const B &obj2)
{
      if(obj1.i==obj2.i) return ("A.i=B.i\n");
      return ("A.i<>B.i\n");
}
```

```
main()
{
      A obj1(100); B obj2(10);
      cout<<equal(obj1,obj2);
      return 0;
}
```

Дружественные функции могут быть перегружены.
Любой метод одного класса может быть дружественным по отношению к другому классу. В этом случае для доступа к компонентам классов в качестве аргументов необходимо лишь передать ссылки или указатели на объекты, для которых функция является дружественной. Предположим, что в предыдущем примере функция equal является компонентом класса A и дружественной по отношению к классу B. Тогда описания этой функции будут выглядеть следующим образом:

```
В классе A
...
char* equal(const B&);
...
```

```
В классе B
...
friend char* A::equal (consr B&);
...
```

последняя запись сообщает компилятору, что функция equal() является дружественной к классу B и является компонентом класса A.

Подводя предварительные итоги по дружественным функциям можно сделать напрашивающийся вывод - дружественные функции не имеют доступа к внутреннему указателю объектов this.

Лекция-6:

1. Перегрузка операторов.

До этого момента мы с Вами сталкивались лишь с одном примером использования полиморфизма в C++. Этот вопрос касался перегрузки функций. Сегодня мы преступаем к изучению такого вопроса, как ***перегрузка операторов***. По сути дела каждый оператор в C++ и представляет собой функцию. Операторы могут иметь один или два аргумента. Если оператор имеет один аргумент, то такой оператор называется унарным. Примером унарных операторов в C++ могут служить операторы инкрементации и декрементации (++ и --). Если оператор имеет два аргумента, то такой оператор называется бинарным. Примером бинарных операторов могут служить операторы +,-, *, /, и т.д. Отметим, что операторы встроенных типов C++ не могут быть перегружены. Это означает, например, что невозможно перегрузить оператор ++ таким образом, чтобы он инкрементировал элементы одномерного массива типа int. Все операторы для встроенных типов C++ и так являются перегруженными и поэтому не стоит их перегружать. Любые операции по перегрузке операторов относятся только к пользовательским типам данных, а точнее к типам, которые описывают объекты. Итак, следует подчеркнуть, что перегружать операторы можно только для пользовательских типов данных, т.к. для встроенных типов операторы уже перегружены.

Рассмотрим как же определяется перегрузка операторов на примере классов C++. Перегрузка операторов очень похожа на механизм перегрузки функций. Перегруженный оператор может являться членом класса или быть дружественным к нему. В C++ нельзя перегрузить оператор отдельно от класса. Если вы хотите, что бы перегруженный оператор можно было бы использовать за пределами класса, то такой перегруженный оператор должен иметь общедоступный атрибут доступа (private).

Рассмотрим общие синтаксисы, применяемые для перегрузки операторов. Если перегруженный оператор является членом какого-либо класса, то его прототип задается следующим образом:

```
возвращаемый_тип <имя_класса> ::operator XX (аргументы);
```

Вместо записи XX непосредственно записывается оператор, который подлежит перегрузке. В C++ есть ряд операторов, которые не могут быть подвергнуты перегрузки. В основном это специфичные операторы, используемые для работы методами и данными классов. Отметим, какие из операторов C++ не могут быть перегружены: ., .*, ::, sizeof и ?:. Если вы не знакомы с последним оператором ?: - рассмотрим его подробнее. Данный оператор называется условным

оператором и является в C++ единственным оператором, который использует три аргумента. Для того чтобы понять его работу рассмотрим простейший пример:

```
#include <iostream.h>
main()
{
	int n=0;
	cout<<(n==3 ? "Zero\n" : "Non zero\n");
	return 0;
}
```

Первым аргументом является проверяемое условие. В случае если условие верно, то данный оператор возвращает условное выражение, которое является вторым аргументом, и возвращает третий, если условие не верно.

Возвращаясь к вопросу о перегрузке операторов отметим, что нельзя перегрузить приоритет какого-либо оператора. Скажем, если при выполнения следующей операции:

```
i=(--j+1)+10;
```

выполниться сначала инкремент переменной j, затем сложение этого результата с "1" и только после этого сложение полученного результата с "10", то нельзя перегрузить оператор + так, чтобы он выполнялся в этом случае первым. Список всех операторов, которые могут быть перегружены, а также их приоритеты можно найти в книгах по C++. Напоминаем, что могут быть перегружены такие операторы C++ как ***new*** и ***delete***. Еще одним из негласных правил, которым пользуются при перегрузке операторов, означает, что нельзя перегрузить унарный оператор таким образом, чтобы он работал как бинарный и наоборот. И последнее ограничение на перегрузку операторов - оператор не может иметь аргументов по умолчанию.

Теперь непосредственно перейдем к рассмотрению вопросов. Касающихся перегрузки операторов, и начнем с вопроса о перегрузки бинарных операторов.

2. Перегрузка бинарных операторов.

Сначала рассмотрим механизмы перегрузки операторов, которые являются членами классов. Если для какого-либо класса осуществляется перегрузка бинарного оператора, то данный оператор будет иметь только один аргумент. Объясняется это тем, что второй аргумент по умолчанию является экземпляр этого класса. При этом нет необходимости передавать его отдельным аргументом. Доступ к нему осуществляется с помощью указателя this. Давайте сейчас определим один класс, на примере которого будем рассматривать перегрузку всех операторов. Пусть этот класс будет копией встроенного типа int, однако он реализован как пользовательский. Ниже приведено тело данного класса, которое по мере изучения нами материала будет пополняться все новыми перегруженными операторами.

```
Class Number
{
	int n;
	public:
	Number(int n=0){this->n=n;}
	print();
};
Number::print(){cout<<"n="<<n<<"\n";}
```

Для начала давайте перегрузим бинарные операторы "+" и "-", таким образом чтобы можно было их использовать так:

```
Number N1(5);N2;
N2=N1+5;
N1=N2-4;
```

В этом случае у бинарных операторов будет только один аргумент типа int (целое число), а второй - сам объект, по отношению к которому применяется перегруженный оператор. В обоих случаях эти операторы должны возвращать объекты, т.к. после того как отработают перегруженные операторы - результатом должен быть объект, который присваивается другому. Для того, чтобы не возникало никаких побочных эффектов при возврате объекта из перегруженного оператора, необходимо наличие конструктора копирования (если это необходимо). Запись обоих операторов будет выглядеть следующим образом:

```
Number Number::operator + (const int arg)
{
	Number tmp;
```

```
	tmp.n=this->n+arg;
	return tmp;
}

Number Number::operator - (const int arg)
{
	Number tmp;
	tmp.n=this->n-arg;
	return tmp;
}
```

Следует отметить, что данные перегруженные операторы работают только при записях, описанных в примере их использования. Если бы перегруженные операторы работали без временных объектов и возвращали бы ссылки на текущие объекты, то использование таких операторов было бы некорректным. Так, в случае операции N2=N1-5 сначала бы сработал оператор - , который изменил бы объект N1, а затем измененный объект присвоил бы объекту N2, т.е. произошло бы изменение сразу же двух объектов. Также необходимо отметить, что использовать перегруженный оператор - можно только в данной записи, если будет попытка использовать его следующим образом N2=10-N1, то компилятор такой записи не пропустит. Дело в том, что при такой записи для компилятора появляются два явных аргумента - это целое число и экземпляр класса. В этом случае при описании данного оператора внутри класса компилятор не пропустит данную запись и выдаст предупреждение о том, что у перегруженного бинарного оператора - (+) должен быть один аргумент, а не два. Избежать этого можно определив перегружаемый оператор как дружественный к данному классу. В этом случае передача второго аргумента (ссылка на объект) является весьма оправданной.

```
...
friend Number operator - (const int, const Number&);
...

Number operator - (const int arg, const Number &obj)
{
	Number tmp;
	tmp.n=arg-obj.n;
	return tmp;
}
```

Перегрузка оператора + является идентичной, за исключением одной операции сложения внутри тела оператора.

Как известно в языке C++ доступна следующая запись N2+=5, упрощающая более длинную N2=N2+5. Используется это благодаря бинарному перегруженному оператору +=, у которого имеется два аргумента - целое число и объект. При этом оператор должен возвращать ссылку на тот объект, который используется в качестве аргумента, т.е. ссылку сам на себя. Реализация такого оператора представлена ниже:

```
Number& Number::operator += (const int arg)
{
	this->n+=arg;
	return (*this);
}
```

Мы рассмотрели те случаи, когда вторым аргументом бинарного оператора является другой тип, в данном случае встроенный int. Однако можно перегружать бинарные операторы таким образом, чтобы оба аргумента были объектами данного класса. Рассмотрим пример перегрузки бинарного оператора сложения двух объектов:

```
Number Number::operator + (const Number &obj)
{
	Number tmp;
	tmp.n=this->n+obj.n;
	return tmp;
}
```

Теперь, после определения такого оператора возможна следующая запись:

```
Number N1(2),N2(8),N3;
N3=N1+N2;
```

Особое внимание следует уделить оператору присваивания, который тоже является бинарным. Одна особенность перегрузки такого оператора заключается в том, что он не может быть дружественным классу, а обязательно должен быть членом класса. Применяется этот оператор в качестве механизма, предотвращающего созданий копий и побитного копирования экземпляров класса. Рассмотрим перегрузку оператора присваивания на примере нашего класса. Оператор будет иметь один явный аргумент - константную ссылку на объект, и возвращать - также ссылку на модифицированный объект.

```
Number& Number::operator = (const Number &obj)
{
      this->n=obj.n;
      return (*this);
}
```

Все рассмотренные нами примеры перегрузки бинарных операторов возвращают объект или ссылку на объект. На самом деле в C++ нет ограничений на возвращаемый тип при перегрузке операторов. Однако если оператор возвращает объект или ссылку на него, то можно это свойство использовать для записи более сложных выражений таких как:

```
Number N1(5),N2,N3;
N3=N1=N1+N2+=5+6;
```

Однако существуют бинарные операторы, в которых желательно возвращать вовсе не объект или ссылку на него, а, например, встроенный тип. Это требование относиться к логическим операторам и операторам отношений.

3. Перегрузка логических операторов и операторов отношения.

Все операторы данных типов являются бинарными, и если они перегружаются как члены класса, то должны иметь один аргумент - объект или ссылку на него , а в качестве возвращаемого типа должен быть один из встроенных типов, который говорил бы о верности или нет выражения. Это позволит при проверке с помощью операторов отношения использовать длинные записи, такие как:

```
if((obj1>obj2)||(obj2<obj3)) ...
```

Рассмотрим перегрузку оператора отношения "= =" для нашего класса:

```
int Number::operator == (const Number &obj)
{
      if((this->n)==obj.n) return 1;
      return 0;
}
```

Аналогичным образом можно перегрузить и все остальные логические операторы и операторы отношения.

Лекция-7:

Перегрузка операторов (продолжение).

1. Перегрузка унарных операторов.

Все унарные операторы имеют только один аргумент. При перегрузке унарных операторов этим аргументом является экземпляр класса. В случае если унарный оператор перегружается как член класса, то такой оператор не будет иметь аргументов. Рассмотрим пример перегрузки унарного знака "-" для нашего класса Number:

```
Number& Number::operator - ()
{
        n=-n;
        return (*this);
}
```

Унарный оператор может в качестве возвращаемого значения передавать void, однако возвращение ссылки на измененный объект позволяет использовать унарные операторы в сложных записях, особенно где происходит присваивание объектов.

Особое внимание следует уделить перегрузке таких унарных операторов как "++" и "--". Многие компиляторы C++ при перегрузке данных операторов эквивалентно воспринимают как постфиксную, так и префиксную их запись. В ANSI стандарте языка C++ можно различать постфиксную и префиксную записи. Осуществляется это путем использования второго фиктивного аргумента типа int. Если необходимо подчеркнуть, что оператор будет использоваться в постфиксной записи, то необходимо наличие этого аргумента, который по умолчанию компилятором устанавливается в "0".

```
Number& Number::operator ++ (int)
{
        cout<<"Postfix\n";
        n++;
        return (*this);
}
Number& Number::operator ++ ()
{
        cout<<"Prefix\n";
        ++n;
        return (*this);
}
```

В случае перегрузки данных операторов как дружественных по отношению к классу, их прототипы будут выглядеть следующим образом:

```
Number& operator ++ (Number &obj)
{
        cout<<"Prefix\n";
        ++(obj.n);
        return obj;
}

Number& operator ++ (Number &obj,int)
{
        cout<<"Postfix\n";
        (obj.n)++;
        return obj;
}
```

При использовании постфиксной записи оператора ++ в качестве второго неявного аргумента передается значение "0".

Перегруженный унарный оператор также может возвращать и встроенный тип. Это целесообразно использовать, например, при перегрузке унарной логической операции "НЕ".

2. Перегрузка операторов вставки и извлечения из потока данных.

Как уже известно, данные операторы "<<" и ">>" уже являются перегруженными в стандарте языка C++. Пока мы не перешли к более подробному изучению механизмов ввода/вывода в языке C++, отметим следующее, что для реализации ввода/вывода в C++ также используются стандартные классы. Одними из таких классов являются ***ostream*** и ***istream***. Данные классы созданы для поддержки стандартного вывода и ввода данных соответственно. Экземпляры данных классов вам хорошо уже известны - это ***cout*** и ***cin***. Одной из особенностью языка C++, связанной с его расширяемостью, является возможность перегрузки операторов для работы со стандартными потоками данных. Сегодня мы рассмотрим как можно перегрузить операторы "<<" и ">>" для стандартных потоков. Следует отметить, что данные операторы работы со стандартными потоками не могут быть членами какого либо класса, однако они могут быть дружественными по отношению к какому-либо классу.

Общий синтаксис перегрузки оператора вставки в поток для классов имеет

```
ostream& operator << (ostream &ссылка_на_поток, имя_класса объект)
```

Рассмотрим пример перегрузки оператора вставки в поток для нашего класса Number. Оператор будет дружественным по отношению к классу, т.к. необходимо будет осуществлять доступ к закрытой компоненте n.

```
...
friend ostream& operator << (ostream&,Number&);
...
ostream& operator << (ostream &os,Number &obj)
{
```

```
        os<<"n="<<obj.n<<"\n";
        return os;
}
```

Форма перегрузки оператора извлечения из потока очень похожа на перегрузку оператора "<<":

```
istream& operator >> (istream&,имя_класса объект);
```

Данный оператора также может быть дружественным по отношению к какому-либо классу, или определяться как независимый оператор. Рассмотрим пример перегрузки оператора извлечения из стандартного потока для нашего класса:

```
...
friend istream& operator >> (istream&,Number&);
...
istream& operator >> (istream &is,Number &obj)
{
        cout<<"Input n:";
        is>>obj.n;
        return is;
}
```

3. Перегрузка манипуляторов.

Для стандартных потоков данных в C++ могут быть применены специальные операторы, называемые ***манипуляторами***. Манипуляторы предоставляют средства форматированного ввода-вывода. Часть стандартных манипуляторов определена в заголовочном файле iostream.h а часть в iomanip.h. Вы наверное уже сталкивались со многими стандартными манипуляторами. Одним из самых часто используемых является манипулятор ***endl,*** который отвечает за очистку буфера и перевод на начала следующей строки. Приведем пример еще нескольких манипуляторов: ***oct, dec, hex*** - которые отвечают за установку системы счисления для выводимых данных. Все вышеперечисленные манипуляторы работают с внутренними компонентами, в данном случае класса ostream, и действуют до их явной отмены. Средства работы с манипуляторами в C++ являются расширяемыми. Пользователь может определить свой собственный манипулятор. Начнем рассмотрение с простейшего манипулятора, который записывает в выводной поток 10 символов "*".

```
ostream& stars(ostream &os)
{
        for(int i=0;i<10; os<<'*', i++);
        os<<endl;
        return os;
}
...
cout<<stars;
...
```

Как видно из этого примера данный манипулятор является простейшим - без параметров. Определение манипуляторов такого типа весьма простое - один единственный аргумент - ссылка на поток (в данном случае на выводной поток) и возвращаемое значение - ссылка на видоизмененный поток. Данное описание является описанием простой функции. Компилятор воспринимает запись вида cout<<stars следующим образом. Происходит разыменование перегруженного оператора <<, результат такого разыменования есть адрес функции stars. Затем выполняется эта функция и после этого происходит возврат ссылки на поток и дальнейшие действия связанные с ним.

Рассмотрим более сложные примеры использования пользовательских манипуляторов, у которых есть один и более аргументов. В заголовочном файле iomanip.h описан ряд специальных макросов, предназначенных для создания манипуляторов с параметрами (аргументами). Простейшие манипуляторы могут принимать только один аргумент типа int или long. Рассмотрим пример использования пользовательского манипулятора с одним аргументом:

```
ostream& stars(ostream &os, int n)
{
        for(int i=0;i<n; os<<'*', i++);
        os<<endl;
        return os;
}

OMANIP(int) Stars(int how_many)
```

```
{
        return OMANIP(int) (stars,how_many);
}
...
cout<<Stars(12);
...
```

В начале определяется функция, которая отвечает за выполняемые над потоком данных действия. В данном случае это функция stars, которая имеет два аргумента - ссылку на поток и количество выводимых символов. Возвращать эта функция должна ссылку на этот поток. После того как определена функция - необходимо определить манипулятор. Для этого используется специальный макрос OMANIP(int), расширяемый в класс _ _omanip_int.

```
#define OMANIP(typ) __omanip_##typ
```

Рассмотрим подробнее этот класс:

```
class _ _omanip_int
{
   ostream& (*fun)(ostream&,int);
   int i;
   public:
   _ _omanip_int(ostream& (*tfun)(ostream&,int),int ti){fun=tfun; i=ti;}
  friend ostream& operator << (ostream &os, const _ _omanip_int &m)
  {
      return (*m.fun)(os, m.i);
  }
};
```

Данный класс содержит два элемента - указатель на функцию с 2-мя параметрами ostream& и int и возвращающая указатель на поток. Второй элемент - int. Внутри класса описан конструктор с двумя параметрами, который производит инициализацию внутренних переменных. Дружественный оператор вставки в поток, который имеет тоже два аргумента - ссылку на поток и константную ссылку на объект класса _ _omanip_int. Внутри перегруженного оператора происходит вызов функции, указатель на которую содержит объект класса _ _omanip_int, и возвращает возвращаемый тип этой функции - ссылку на поток. Рассмотрим работу такого манипулятора при его использовании. При включении функции Stars(12) в поток она вызывает конструктор класса _ _omanip_int, который производит инициализацию внутренних компонентов объекта, и после этого используется перегруженный оператор вставки в поток, вызывающий функцию stars(), которая возвращает ссылку на поток.

При создании манипуляторов, использующие аргументы отличные от int и long, необходимо использовать макрос IOMANIPdeclare(typ), где typ - тип, который необходимо передавать. Для передачи нескольких аргументов в данном случае используется структура. Манипулятор получает значения аргументов через структуру, инициализирует их, вызывает конструктор, который инициализирует внутренние компоненты объекта класса _ _omanip. Описание данного класса напоминает описание класса _ _omanip_int, за исключением используемого стандартного типа int.

```
struct st
{
        int length;
        char symbol;
};

IOMANIPdeclare (st);

ostream& STARS(ostream &os, st _st)
{
        for(int i=0;i<_st.length; os<<_st.symbol, i++);
        os<<endl;
        return os;
}

OMANIP(st) S(int length, char symbol)
{
        st _st;
        _st.length=length;
        _st.symbol=symbol;
```

```
        return OMANIP(st) (STARS,_st);
}
```

Действия по использованию манипулятора с несколькими параметрами аналогичны вышеописанным для манипуляторов с один параметром стандартного типа.

Лекция-8:

1. Наследование.

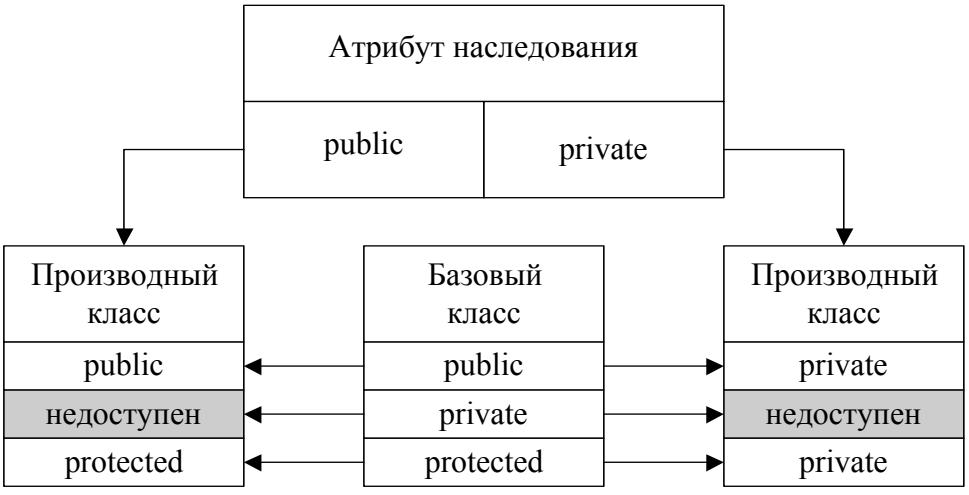
Наследование является одним из ключевых понятий ООП. Наследование подразумевает использование данных и методов одного класса другим. При разговоре о наследовании всегда выделяют класс, от которого наследуются его свойства - базовый класс, и класс, который наследуется - ***производный класс***. Начнем рассматривать наследование на простых примерах, касающихся одиночного наследования, когда имеется только один базовый класс, а производных классов - сколь угодно. Продолжим изучение вопроса о наследовании на примерах множественного наследования, и очередного важного вопроса, связанного с наследованием - полиморфизмом.

2. Одиночное наследование.

Наследование подразумевает использование принципа иерархии и абстрагирования объектов. Приведем пример, пусть имеется один базовый класс, который описывает, или содержит в себе общие свойства разных объектов. Самым распространенным примером является пример с геометрическими фигурами. Можно определить некий базовый класс, который описывает многоугольники. Пусть у данного класса определены некоторые свойства, такие как количество множество координат вершин. На базе данного класса можно создать несколько производных классов, например четырех угольник. Сам производный класс четырехугольник может быть в свою очередь быть базовым для таких классов, как ромб или прямоугольник, и т.д. Производный класс получается некой надстройкой над базовым классом, определяя в себе все, или почти все, свойства базового класса и имея свои собственные свойства. Рассмотрим как можно определить наследование одного класса другим в C++:

```
class имя_производного_класса : атрибут_наследования имя_базового класса {...};
```

Атрибут наследования определяется тремя известными ключевыми словами : private, public and protected., которые использовались для доступа к внутренним компонентам классов. Данный атрибут при наследовании определяет видимость компонент базового класса внутри производного. Рассмотрим как атрибут наследования влияет на видимость компонент базового класса.



Атрибут наследования protected эквивалентен private, только за одним исключением - все компоненты базового класса с атрибутом доступа protected, доступны в производных классах. Как базовый, так и производные ему классы могут иметь конструкторы и деструкторы, ничем не отличающиеся от тех, которые были уже рассмотрены. Существует правило вызова конструкторов и деструкторов при одиночном наследовании. При создании экземпляра производного класса сначала выполняется конструктор базового а затем производного класса. Деструкторы выполняются в обратном

порядке. При создании объектов базового класса выполняются конструктор и деструктор только данного базового класса. Эта идея очень проста, если учесть тот факт, что любой объект производного класса должен быть построен на основе объекта базового класса. Конструкторы производных и базового классов могут иметь аргументы. Если при создании экземпляра производного класса нет необходимости в передаче каких-либо аргументов конструктору базового класса, то данный конструктор должен быть описан либо как конструктор без параметров либо иметь все аргументы по умолчанию. Производный класс может производить фиктивную передачу аргументов конструктору базового класса без их использования. Для того чтобы осуществить передачу аргументов для конструктора базового класса необходимо воспользоваться следующей записью:

```
конструктор_производного_класса                    (аргументы)                    :
конструктор_базового_класса(аргументы)
```

Количество и тип аргументов для конструкторов производных и базового класса могут не совпадать. Рассмотрим пример использования всех вышеперечисленных свойств:

```
#include <iostream.h>
class A
{
        protected:
        int n;
        public:
        A(int n=0){this->n=n;cout<<"Constructor A\n";}
        ~A(){cout<<"Destructor A\n";}
};
class B:protected A
{
        public:
        print(){cout<<"n="<<n<<"\n";}
        B():A(5) {cout<<"Constructor B\n";}
        ~B(){cout<<"Destructor B\n";}
};

main()
{
        B obj;        obj.print();
}
```

Если конструктор базового класса имеет аргументы, то для передачи этих аргументов их должен содержать конструктор производного класса. Рассмотрим пример:

```
#include <iostream.h>
class A
{
        int n;
        public:
        A(int n)
        {this->n=n; cout<<"Constructor A\n n="<<n<<"\n";}
        ~A(){cout<<"Destructor A\n";}
};
class B:public A
{
        int m;
        public:
        B(int m,int n):A(n)
        {this->m=m;cout<<"Constructor B\n m="<<m<<"\n";}
        ~B(){cout<<"Destructor B\n";}
};
main()
{
        B obj(1,2); return 0;
}
```

```
Constructor A
 n=2
Constructor B
```

```
 m=1
Destructor B
Destructor A
```

В случае, если конструктор базового класса имеет все аргументы по умолчанию, то такой конструктор можно явно не задавать при описании конструктора производного класса.

Отметим еще одну особенность, связанную с использованием указателей на базовый и производный классы. Если в программе определен указатель на базовый класс, то его можно использовать для доступа к компонентам объектов базового класса, а также для доступа к компонентам базового класса, которые наследуются в производном классе. Если указатель указывает на экземпляр производного класса, то с помощью такого указателя нельзя получить доступ к тем компонентам, которые описаны только внутри производного класса. Если же описан указатель на производный класс, то при помощи такого указателя можно осуществлять доступ к открытым компонентам данного класса, в том числе и к компонентам базового класса, которые открыто наследуются в производном классе. Нельзя использовать указатель производного класса для доступа к компонентам объектов базового класса. Рассмотрим пример, в котором используются указатели на производный и базовый классы:

```
#include <iostream.h>
class A
{
        int n;
        public:
        A(int n=0)
        {this->n=n; cout<<"Constructor A\n n="<<n<<"\n";}
        ~A(){cout<<"Destructor A\n";}
        print_A(){cout<<"n="<<n<<"\n";}
};
class B:public A
{
        int m;
        public:
        B(int m,int n):A(n)
        {this->m=m;cout<<"Constructor B\n m="<<m<<"\n";}
        ~B(){cout<<"Destructor B\n";}
        print_B(){cout<<"m="<<m<<"\n";}
};
main()
{
    A obj1(5);
    B obj2(1,2);

    A *ptr=&obj1;
    ptr->print_A();
    ptr=&obj2;
    ptr->print_A();
    B *ptr2=&obj2;
    ptr2->print_A();
    ptr2->print_B();
    return 0;

}
```

```
Constructor A
 n=5
Constructor A
 n=2
Constructor B
 m=1
n=5
n=2
n=2
m=1
Destructor B
Destructor A
Destructor A
```

Лекция-9:

Продолжение.

1. Виртуальные функции.

Рассмотрев указатели на производные и базовые классы, перейдем к рассмотрению одного из самых интересных моментов ООП - вопрос о виртуальных функциях. Рассмотрим следующий пример:

```
#include <iostream.h>
class A
{
        int n;
        public:
        A(int n){this->n=n;}
        print(){cout<<"Base n="<<n<<"\n"; return 0;}
};
class B: public A
{
        char c;
        public:
        B(char c):A(1) {this->c=c;}
        print(){cout<<"Derived c="<<c<<"\n"; return 0;}
};

main()
{
        A obj1(1); B obj2('W');
        obj1.print();
        obj2.print();
}
```

```
Base n=1
Derived c=W
```

В данном примере выглядеть все логично. В производном и базовом классах определены функции, отвечающие за вывод внутренних компонент. Вызов данных функций происходит у явным указанием экземпляров этих классов. А теперь рассмотрим этот же пример, но вызов функций print() будем осуществлять через указатель на базовый класс.

```
main()
{
        A obj1(1); B obj2('W');
        A *ptr;
        ptr=&obj1;
        ptr->print();
        ptr=&obj2;
        ptr->print();
        return 0;
}
```

```
Base n=1
Base n=1
```

Результат неожиданный, но вполне объяснимый. Мы с вами знаем, что указатель, объявленный на базовый класс, может быт использован в качестве указателя на любой производный класс. В нашем примере, когда указатель на базовый класс указывает на экземпляр базового класса - вполне логично происходит вызов функции print() базового класса. В случае же, когда указатель на базовый класс указывает на экземпляр производного класса, по идее должна вызваться функция из производного класса. Но этого не происходит. Объясняется это следующим образом. Указатель на базовый класс может быть использован только для доступа к компонентам, которые описаны в базовом классе. Для доступа к компонентам из производных классов необходимо использовать указатель на производный класс. В случае использования указателя на производный класс эта задача решается путем использования виртуальных функций (методов).

Виртуальная функция задается точно также как и обычная, только в начале определения такой функции ставиться ключевое слово virtual. Виртуальная функция объявляется внутри базового класса. Если виртуальная функция переопределяется в производных классах, то она автоматически в них становиться виртуальной, и в этом случает нет необходимости использовать ключевое слово virtual. Виртуальная функция может быть вызвана как

обычная, но что происходит, когда используется указатель на базовый класс и виртуальная функция. Определим в нашем примере функцию print() как виртуальную, и проследим, что произойдет с ее вызовами.

```
Base n=1
Derived c=W
```

С понятием виртуальных функций тесно связано такое понятие как полиморфизм. На рассмотренном примере это будет выглядеть следующим образом: если используется указатель на базовый класс, в котором определена виртуальная функция, и эта функция переопределена в производных классах, то при адресации указателя базового класса на экземпляры производных, будет вызываться функция, соответствующая каждому производному классу. Виртуальные функции немного отличаются от перегруженных функций. Виртуальная функция, в отличие от перегруженных, должна полностью повторяться в производных классах, т.е. имя функции, список аргументов и возвращаемое значение обязательно должны совпадать, иначе такая функция будет считаться просто перегруженной функцией и не будет являться виртуальной. Еще один нюанс, касающийся виртуальных функций - они обязательно должны быть компонентами класса.

Рассмотрим один небольшой пример:

```
#include <iostream.h>
class A
{
      public:
      virtual print(){cout<<"Base A"<<"\n"; return 0;}
};
class B: public A
{
      public:
      print(){cout<<"Derived B"<<"\n"; return 0;}
};
class C: public A
{
      public:
      func() {cout<<"Nothing\n"; return 0;}
};
main()
{
      A obj1; B obj2; C obj3;     A *ptr;
      ptr=&obj1; ptr->print();
      ptr=&obj2; ptr->print();
      ptr=&obj3; ptr->print();   ptr->func();
      return 0;
}
```

Попробуйте найти ошибку и устранить ее. Как будет работать программа при ликвидации этой ошибки?

Запишем следующее: если виртуальная функция не определена в производном классе, однако осуществляется ее вызов через указатель на базовый класс, который указывает на данный производный, то вызовется виртуальная функция из базового класса.

2. Чисто виртуальные функции.

Часто возникают ситуации при которых виртуальные функции, определенные в базовых не используются, а иногда и не содержат никаких действий, а являются лишь прототипами (шаблонами) для конкретных реализаций виртуальных функций в производных классах. Для того, чтобы подчеркнуть, что в программе не предусматривается вызов виртуальной функции для базового класса, используют чисто виртуальные функции. Для их определения используют следующую запись:

```
virtual возвращаемый_тип имя (аргументы) =0;
```

Данная запись при компиляции воспринимается как отсутствие определения виртуальной функции для базового класса, и вызов такой функции будет восприниматься как ошибка. Если в базовом классе определен прототип чисто виртуальной функции, то она должна быть обязательна определена для всех производных классов. Если в базовом классе определена хотя бы одна чисто виртуальная функция то такой класс называется абстрактным базовым классом.

Подробнее про абстрактные классы мы будем говорить при рассмотрении вопроса о множественном наследовании. Отметим лишь, что нельзя создать экземпляр абстрактного класса.

Особо интересное использование виртуальных функций является их использование при случайных событиях, которые обрабатывает программа. Яркий пример - ожидание ввода с клавиатуры. В зависимости от того каков будет ввод данных, к примеру, должны будут вызываться виртуальные функции из разных производных классов с использованием указателя на базовый. Этот пример является классическим в рассмотрении такого вопроса как ранее и позднее связывание.

2. Ранее и позднее связывание.

Отметим, что под процессами раннего связывания понимают те процессы, которые могут быть предопределены на стадии компиляции. Пример - при использовании вызовов обычных функций. Компилятор заранее знает адрес вызываемой функции, и этот адрес помещает в место вызовов этой функции.

Другое дело обстоит при вызове виртуальных функций - позднее связывание. Процессы, относящиеся к позднему связыванию, определяются на стадии выполнения программы. Так, например, компилятор заранее может не предугадать вызов виртуальной функции для конкретных экземпляров производных классов. Адрес виртуальной функции вычисляется только при работе программы.

Лекция-10:

1.Виртуальные функции (продолжение).

Как уже было отмечено на прошлой лекции, нужная версия виртуальной функции выбирается на стадии выполнения программы. Такой процесс называется "поздним связыванием". Как же это происходит на самом деле мы сегодня с вами рассмотрим на примере объектного кода следующей программы:

```
#include <iostream.h>
class A
{
        int n;
        public:
        A(int n){this->n=n;}
        virtual void print(){cout<<"n="<<n<<endl;}
};
class B:public A
{
        int m;
        public:
        B(int m=0):A(0){this->m=m;}
        void print(){cout<<"m="<<m<<endl;}
};

main()
{
        A obj1(1);
        B obj2;
        A *ptr;
        ptr=&obj1;
        ptr->print();   // n=1
        ptr=&obj2;
        ptr->print();   // m=0
        return 0;
}
```

Рассмотрим объектный код этой программы. Будем считать, что она была скомпилирована с использованием модели памяти Small, без оптимизации, без использования регистровых переменных и с отключенной опцией "Far virtual tables".

```
; bp - используется для адресации к локальным переменным
;[bp-04] - obj1
;[bp-0A] - obj2
;[bp-0C] - ptr
```

```
;A obj1(1);
mov    ax,0001    ; передаем в конструктор число 1
push   ax         ;
lea    ax,[bp-04]; ax=obj1
push   ax         ; передаем в конструктор указатель this
call   A::A       ; вызываем конструктор базового класса
add    sp,0004    ; восстанавливаем состояние стека

;B obj2;
xor    ax,ax      ; передаем в конструктор по умолчанию число 0
push   ax
lea    ax,[bp-0A]; ax=obj2
push   ax         ; предаем в конструктор указатель this
call   B::B       ; вызываем конструктор производного класса
add    sp,0004    ; восстанавливаем состояние стека

;ptr=&obj1;
lea    ax,[bp-04]; ax=obj1
mov    [bp-0C],ax; ptr=ax=obj1

;ptr->print();
push   ax         ; передаем указатель this для obj1
mov    bx,[bp-0C]; bx=ptr - он же указатель на таблицу виртуальных
                  ; функций
mov    bx,[bx]    ; сейчас bx содержит указатель на первую функцию
                  ; из списка виртуальных функций
call   [bx]       ; вызываем виртуальную функцию
add    sp,0002    ; восстанавливаем стек

;ptr=&obj2;
lea    ax,[bp-0A]; ax=obj2
mov    [bp-0C],ax; ptr=ax=obj2

;ptr->print();
push   ax         ; данный фрагмент кода полностью аналогичен
                  ; предыдущему. Если была бы включена оптимизация
                  ; по размеру, то этот фрагмент был бы помещен в
                  ; отдельную функцию.
mov    bx,[bp-0C]
mov    bx,[bx]
call   [bx]
add    sp,0002

;return 0;
xor    ax,ax      ; возвращаем число 0
;}
mov    sp,bp
pop    bp
ret
```

```
;A::A:  A(int n); конструктор базового класса

push   bp
mov    bp,sp
cmp    word ptr [bp+04],0000 ; сравниваем указатель this с "0"
jne    m1                    ; объект уже существует - на m1
mov    ax,0004               ; объект надо создать, размер - 4
                             ; байта
push   ax
call   operator new          ; выделяем память под 4 байта
add    sp,0002               ; восстанавливаем стек
mov    [bp+04],ax            ; в ax возвращается указатель на
                             ; выделенную память
or     ax,ax                 ; проверяем указатель на NULL
je     m2

m1:
mov    bx,[bp+04]
```

```
mov     word ptr [bx],00B2      ; указатель на виртуальную функцию из
                                ; базового класса (offset A::print)
mov     ax,[bp+06]              ; размещение переменной n для
                                ; базового класса
mov     [bx+02],ax

m2:
mov     ax,[bp+04]              ; в ax возвращаем указатель this
pop     bp
ret
```

```
;B::B:  B(int m=0):A(0); конструктор производного класса

push    bp                      ;
mov     bp,sp                   ;
cmp     word ptr [bp+04],0000 ; сравниваем this с "0"
jne     m3                      ; объект уже создан - на m3

mov     ax,0006                 ; иначе - объект надо создать
push    ax                      ; размер объекта - 6 байт
call    operator new            ; выделяем память
add     sp,0002                 ; восстанавливаем стек
mov     [bp+04],ax              ; в ax - указатель на выделенную
                                ; память
or      ax,ax                   ; сравниваем указатель с "0"
je      036A                    ; если все в порядке то на m4

m3:
xor     ax,ax                   ; передаем конструктору базового
                                ; класса число "0"
push    ax                      ;
push    word ptr [bp+04]        ; и указатель this производного
                                ; класса
call    A::A                    ; вызываем конструктор базового
                                ; класса
add     sp,0004                 ; восстанавливаем стек
mov     bx,[bp+04]              ;
mov     word ptr [bx],00B0      ; размещаем указатель на функцию
                                ; производного класса в таблицу
                                ; виртуальных функций
mov     ax,[bp+06]              ; инициализируем переменную m
mov     [bx+04],ax

m4:
mov     ax,[bp+04]              ; в ax - указатель this
pop     bp
ret
```

В рассмотренном примере объекты базового и производного классов имеют размеры 4 и 6 байт соответственно.

Объект базового класса :
первые 2-а байта - указатель на таблицу виртуальных функций;
последние 2-а байта - переменная int n.
Объект производного класса :
первые 2-а байта - указатель на таблицу виртуальных функций;
следующие 2-а байта - переменная int n;
последние 2-а байта - переменная int m.

2.Виртуальные деструкторы

Виртуальные деструкторы необходимы в случаях использования указателей на базовые классы при выделении динамической памяти под объекты производных классов. Рассмотрим на примере следующей программы, как бы вызывались деструкторы производных и базового классов в случае виртуальных и не виртуальных деструкторов.

```
#include <iostream.h>
class Something
{
        public:
        Something(char *what){cout<<"This is "<<what<<endl;}
        void virtual print()=0;
        virtual ~Something() {cout<<"Base destructor\n";}
```

```
};

class Number:public Something
{
        int n;
        public:
        Number(int n, char *id="number"):Something(id)
        {this->n=n;}
        void print() {cout<<"n="<<n<<endl;}
        ~Number(){cout<<"Number destructor\n";}
};
class String:public Something
{
        char *str;
        public:
        String(char *str, char *id="string"):Something(id)
        {this->str=str;}
        void print() {cout<<"str="<<str<<endl;}
        ~String(){cout<<"String destructor\n";}
};
main()
{
        Something *ptr1,*ptr2;
        ptr1=new Number (5);
        ptr2=new String ("Hello!");
        ptr1->print();
        ptr2->print();
        delete ptr1;
        delete ptr2;
        return 0;
}
```

```
This is number
This is string
n=5
str=Hello!
Number destructor
Base destructor
String destructor
Base destructor
```

В случае не виртуальных деструкторов при удалении объектов производных классов вызывался бы деструктор только базового класса. В случае, если деструктор базового класса объявлен как виртуальный, все деструкторы производных классов автоматически становятся виртуальными. Как и в случае виртуальных функций, если в производном классе не описан деструктор, то при удалении объекта будет вызван деструктор базового класса. Если деструктор - описан, то сперва произойдет вызов деструктора производного, а затем базового классов.

Лекция-11:

1. Множественное наследование.

До сих пор мы с вами рассматривали случаи, когда один класс мог наследоваться только от одного базового класса. При этом любой класс мог быть базовым для другого класса. Такой механизм наследования называется одиночным наследованием. Так же мы рассмотрели вопросы, связанные с вызовами конструкторов и деструкторов производных классов и виртуальные функции. Напомним, что при создании экземпляра производного класса сперва вызывается конструктор базового класса, а затем конструктор производного класса. Порядок вызова деструкторов является обратным порядку вызова конструкторов.

В C++ существует возможность наследовать более одного класса в качестве базового. Такой механизм наследования называется множественным наследованием. При множественном наследовании объекты производных классов могут использовать данные и методы нескольких классов, а также в свою очередь быть базовыми классами для других. Рассмотрим как задаются производные классы от более чем одного базового класса:

```
class derived: атрибут_наследования base1, ..., атрибут_наследования baseN
{
....
};
```

Атрибуты наследования и их смысл тот же, что и при одиночном наследовании. Данная последовательность базовых классов определяет порядок вызова конструкторов базовых классов. При создании экземпляра производного класса в первую очередь вызовется конструктор базового класса, имя которого определено первым в списке наследуемых классов. Далее - по порядку. Последним вызовется конструктор производного класса. При уничтожении объекта производного класса деструкторы вызываются в обратном порядке вызовам конструкторов. Как и в случае одиночного наследования, если конструктор или конструкторы базового класса содержат как минимум один аргумент, то производный класс также должен содержать конструктор. Если конструктор базового класса имеет аргументы без умолчания, то их надо передать через конструктор производного класса. При описании конструктора производного класса задаются те конструкторы базовых классов, которые имеют аргументы. Если базовый класс не имеет аргументов или все аргументы такого конструктора используются по умолчанию, то имя такого конструктора можно не задавать в описании производного класса.

```
Конструктор_производного_класса     (аргументы)     :     base1(аргументы),     ...
,baseN(аргументы)
{
...
}
```

Список конструкторов базовых классов не влияет на порядок их вызова. Он определяется только списком имен базовых классов в начале определения производного класса.

2.Виртуальные базовые классы.

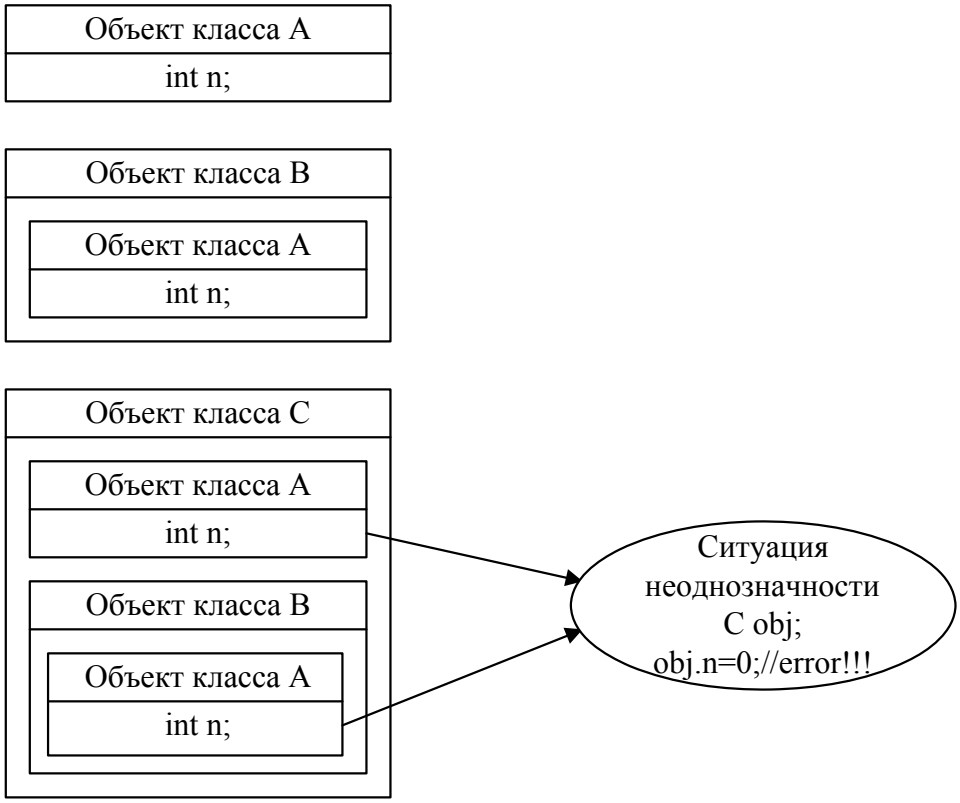
В языке C++ запрещено непосредственно предавать базовый класс в производный более одного раза, т.е. имя класса в списке базовых классов не может повторяться:

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

Однако могут возникать ситуации, когда один производный класс косвенно может наследовать один и тот же базовый класс через другие базовые классы. Так, например:

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

При данной схеме наследования классов получается следующая структура объектов.



Такое наследование вполне возможно и компилятор не выдаст сообщение об ошибке, а только предупредит, что наследуемый класс A также находиться и в наследуемом классе B. Однако в данном примере ошибка возникнет на стадии компиляции. Компилятор не различит, какую именно переменную int n базового класса необходимо изменить: или переменную, которая непосредственно является компонентой класса A или переменную, которая доступна из наследуемого класса B? Решить такого рода проблему в C++ можно путем использования виртуальных базовых классов. Если один базовый класс косвенно наследуется в одном производном классе и наследуется с атрибутом virtual, то в экземпляр производного класса будет помещена только одна копия базового класса. Базовый класс объявляется как виртуальный только при наследовании - в списке базовых классов с указанием спецификатора virtual:

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

Следует отметить, что если базовый класс наследуется производным как виртуальный, то его компоненты все равно доступны в производном классе. Отличие между обычным и виртуальным наследованием заключается в том, что когда класс наследует базовый класс более одного раза, то он будет содержать только одно вхождение базового класса. Рассмотрим следующий пример использования виртуальных базовых классов, а также механизмы вызовов конструкторов и деструкторов.

```
#include <iostream.h>
class A
{
        public:
        int a;
        A(int a){this->a=a;cout<<"Constructor A\n";}
        ~A(){cout<<"Destructor A\n";}
        void print(){cout<<"a="<<a<<endl;}
};
class B
{
        public:
        int b;
        B(int b){this->b=b;cout<<"Constructor B\n";}
        ~B(){cout<<"Destructor B\n";}
        void print(){cout<<"b="<<b<<endl;}
};
class C:virtual public A
{
        public:
```

```
        int c;
        C(int c):A(c){this->c=c;cout<<"Constructor C\n";}
        ~C(){cout<<"Destructor C\n";}
        void print(){cout<<"c="<<c<<endl;}
};
class D:public B,virtual public C,virtual public A
{
        public:
        int d;
        D(int d):A(d+1),B(d+2),C(d+3)
        {this->d=d;cout<<"Constructor D\n";}
        ~D(){cout<<"Destructor D\n";}
        void print()
        {
                cout<<"d="<<d<<endl;
                A::print();
                B::print();
                C::print();
        }
};
main()
{
        D obj(1);
        obj.print();
        return 0;
}
```

```
Constructor A
Constructor C
Constructor B
Constructor D
d=1
a=2
b=3
c=4
Destructor D
Destructor B
Destructor C
Destructor A
```

При виртуальном наследовании базовых классов в первую очередь вызываются конструкторы виртуальных базовых классов в порядке наследования. Так, в нашем примере первым будет вызван конструктор виртуального класса A, затем конструктор виртуального класса C, затем конструктор базового класса B и последним будет вызван конструктор производного класса. Деструкторы вызываются в обратном порядке. В случае, если базовый класс C наследовался бы как не виртуальный, то порядок вызова конструкторов был бы следующий: конструктор класса A, конструктор класса B, конструктор класса C, конструктор класса D. И последнее, если базовый класс A наследовался бы как не виртуальный, то это привело бы к ошибке на стадии компиляции.

На последок, для закрепления материала, рассмотрим пример программы, который использует виртуальные функции. Предположим, что необходимо создать список элементов разных типов, например, int и char*. Любой элемент вне зависимости от его типа может принадлежать какому-то множеству. Необходимо также создать функции, которые выводили бы значения элементов, принадлежащих заданному множеству, а также удаляли бы из списка элементы заданного множества. Принадлежность элемента тому или иному множеству задается при помощи идентификатора множества.

```
#include <iostream.h>
class Base              //базовый класс
{
protected:
        Base *next;
        Base *prev;
        int num;        //идентификатор подмножества
        static Base *Begin;
        static int NumObj;
public:
        Base(int);
        virtual void print()=0;
        static void printall(int);
        static void del(int);
};
```

```
Base *Base::Begin=NULL;
int Base::NumObj=0;
Base::Base(int k)       //при создании объекта класса Base
                        //он сам себя помещает в стек
{
num=k;
        NumObj++;
        next=Begin;
        if (Begin!=NULL) Begin->prev=this;
        Begin=this;
        Begin->prev=NULL;
}

void Base::printall(int k)      //печатает значения всех элементов
                                //с заданным идентификатором подмножества
{
        Base *p=Begin;
        for(int i=0;i<NumObj;p=p->next,i++)
                if (p->num==k) p->print();
}

void Base::del(int k)           //удаляет все элементы с заданным
                                //идентификатором подмножества
{
        int i=NumObj;
        for(Base *tmp=Begin;i>0;tmp=tmp->next,i--)
if (tmp->num==k)
                {
                        if (tmp==Begin) Begin=tmp->next;
                                else tmp->prev->next=tmp->next;
                        delete tmp;
                }
}

class Set                       //класс для работы с подмножествами
{
        static int Lastident;
        int ident;
public:
        Set() {ident=++Lastident;}
        ~Set(){Base::del(ident);}
        int getident() {return ident;}
        friend ostream &operator<<(ostream &,Set&);
};

int Set::Lastident=0;
ostream &operator<<(ostream &os, Set &s)
{
        Base::printall(s.ident);
        return os<<endl;
}

class Integer:public Base       //производный класс с типом элементов int
{
        int i;
public:
        Integer(int i, int num):Base(num) {this->i=i;}
        void print() {cout<<i<<' ';}
        int& get() {return i;}
};

class String:public Base        //производный класс с типом элементов *char
{
        char *str;
public:
        String(char *str,int num):Base(num) {this->str=str;}
        void print() {cout<<str<<' ';}
};
main()
```

```
{
        Set s1;
        int id1=s1.getident(),id2;
        Integer t1(1,id1),t2(5,id1);
        String t3("string1",id1),t4("string2",id1);
        {
                Set s2;
                id2=s2.getident();
                String t5("Hello",id2);
                Integer t6(12,id2);
                cout<<s1<<s2;
        }
        Base::printall(id1);
        return 0;
}
```

```
string2 string1 5 1
12 Hello
string2 string1 5 1
```

Лекция-12:

1.Шаблоны.

На этой лекции мы с вами познакомимся с одним из последних нововведений стандарта языка C++ - шаблонами. Данное понятие распространено на функции и классы. Начнем наше рассмотрение этого материала с шаблонов функций.

2. Шаблоны функций.

Шаблоны функций являются логическим продолжением механизма перегрузки функций. Обычно перегрузку функций осуществляют для различных типов данных, причем все перегруженные функции осуществляют идентичные действия. Так, например, для сортировки и вывода на экран одномерных массивов данных различных типов, приходиться реализовывать различные перегруженные функции, содержимое которых порой отличается только используемыми типами данных. Так, для возможности сортировки массивов int и char* приходиться реализовывать две перегруженные функции sort(int,int) и sort(char*,int). Механизм шаблонов функций в C++ позволяет более легких и гибким для программиста путем решить эту проблему. Достаточно описать один шаблон, к примеру, функции сортировки и вывода на экран, а компилятор сгенерирует код такой функции в соответствии используемому типу данных при вызове такой функции. Рассмотрим пример:

```
#include <iostream.h>
void print(int *array,int d)
{
        for(int i=0;i<d;cout<<array[i],i++);
        cout<<endl;
}
void print(char *array,int d)
{
        for(int i=0;i<d;cout<<array[i],i++);
        cout<<endl;
}

main()
{
        char *A1="HELLO!";
        int A2[5]={1,2,3,4,5};
        print(A1,6);
        print(A2,5);
        return 0;
}
```

В данном примере пришлось реализовывать две идентичные функции, которые выполняют одни и те же операции и различаются только типом одного аргумента. Рассмотрим как это можно реализовать с помощью шаблонов. В общем случае шаблон функции задается следующим образом:

```
template <class тип_параметра> возвращаемый_тип имя_функции(аргументы)
{...}
```

Определение каждого типа параметра начинается с ключевого слова class. Если функция использует более одного типа, то все они разделяются запятой. При компиляции компилятор вместо типа параметра подставляет необходимый тип, который был задан при вызове функции. Определим шаблон функции print(). В данном случае в описании шаблона будет присутствовать только один тип параметра - тип одномерного массива.

```
#include <iostream.h>
template <class Array_Type> void print(Array_Type *array,int d)
{
        for(int i=0;i<d;cout<<array[i]<<'-',i++);
        cout<<endl;
}

main()
{
        char *A1="HELLO!";
        int A2[5]={1,2,3,4,5};
        long A3[2]={0xfeffff,0xefff00};
        print(A1,6);
        print(A2,5);
        print(A3,2);
        return 0;
}
```

В качестве типа параметра могут быть заданы не только встроенные типы, а также типы, определенные пользователем, в том числе и классы. Рассмотрим пример шаблона функции, аргументом которого является класс:

```
#include <iostream.h>
class Integer
{
        int n;
        public:
        Integer(int n=0){this->n=n;}
        int get(){return n;}
};
class String
{
        char *str;
        public:
        String(char *str="Nothing"){this->str=str;}
        char* get(){return str;}
};

template <class Class_Type> void print_it(Class_Type Obj)
{
        cout<<Obj.get()<<endl;
}
main()
{
        Integer num(12);
        String buf;
        print_it(num);
        print_it(buf);
        return 0;
}
```

Шаблоны функций могут быть перегружены. Если в шаблоне описаны несколько типов параметров, то они должны быть все обязательно описаны в списке аргументов функции.

```
template <class Class_Type1,class Class_Type2>
void print_it(Class_Type1 Obj1, Class_Type2 Obj2)
{
```

```
        cout<<"Obj1="<<Obj1.get()<<endl;
        cout<<"Obj2="<<Obj2.get()<<endl;
}
```

Последний пример шаблона функции примечателен тем, что при его использовании порядок аргументов не имеет значения, т.е. можно использовать этот шаблон двумя способами:

```
  print_it(num,buf);
  print_it(buf,num);
```

3.Шаблоны классов.

Помимо шаблонов функций в C++ можно использовать и шаблоны классов. Идея использования аналогична - описывается шаблон класса, который осуществляет идентичные действия с различными типами данных. Используемый тип данных определяется на стадии компиляции. Общая форма описания шаблона класса представлена ниже:

```
template <class Class_Type1,...,class Class_type2> class имя_класса
{
...
};
```

Экземпляры шаблонов классов описываются так:

```
имя_класса <тип> объект;
```

Все функции компоненты шаблонов классов автоматически становятся шаблонами и при их описании необязательно задавать ключевое слово template. Отметим один нюанс, связанный с объявлением статических переменных внутри шаблонов классов. Статические переменные шаблонов классов необходимо инициализировать для каждого используемого типа данных. Рассмотрим пример:

```
#include <iostream.h>

template <class Type> class Class_type
{
        static int Nobj;
        Type element;
        public:
        Class_type(Type element){this->element=element;Nobj++;}
        Type& get(){return element;}
        friend ostream& operator<<(ostream &os,Class_type &Obj)
        {
                return os<<"Element"<<Nobj<<"="<<Obj.element<<endl;

        }
};

int Class_type<int>::Nobj=0;
int Class_type<char*>::Nobj=0;

main()
{
        Class_type<int> Num(5);
        Class_type<char*> Str("Message");
        cout<<Num<<Str;
        Num.get()++;
        Str.get()="Hello";
        cout<<Num<<Str;

        Class_type<int> Obj(12);
```

```
        cout<<Obj;
        return 0;
}
```

Лекция-13:

1. Обработка исключительных ситуаций.

Одним из последних нововведений в стандарт языка C++ является механизм обработки ошибок или обработки исключительных ситуаций. К исключительным ситуациям относят такие события, происходящие во время выполнения программы, которые могут привести к ее сбою, например: деление на ноль, выделение динамической памяти размером более допустимого, выход индексов массивов за дозволенные пределы, и т.д. К сожалению обработка исключительных ситуаций введена только в последние версии стандарта языка C++ (GCC, Watcom, и т.д.). Механизм обработки исключительных ситуаций реализован с помощью трех ключевых слов в C++ : **try**, **catch**, **throw**. **Try** и **catch** являются блоками. В блоке **try** располагаются те операторы, которые необходимо проверить на исключительные ситуации. Если исключительная ситуация имеет место быть в блоке **try**, то сообщение о ней генерируется оператором **throw**. Перехватывается и обрабатывается сообщение об исключительной ситуации блоком **catch**, в котором располагаются соответствующие операторы - чаще всего сообщения об возникновении исключительных ситуациях.

Блок **try** должен содержать ту часть программы, в которой необходимо отслеживать ошибки. Это могут быть как несколько операторов внутри одной функции, так и все операторы функции main(). Оператор **throw** должен выполняться либо внутри блока **try**, либо в любой функции, вызов которой происходит внутри блока **try**. Любая исключительная ситуация должна перехватываться блоком **catch**, который располагается непосредственно за блоком **try**. Для одного блока try может существовать несколько блоков **catch**. В этом случае выполняется тот блок **catch**, аргумент которого соответствует типу сгенерированного сообщения об исключительной ситуации с помощью оператора **throw**. Общая форма записи блоков try и catch выглядит следующим образом:

```
try
{
...
throw ...;
f(); // throw
}
catch (type1 arg) {...}
catch (typeN arg) {...}
```

Тип сообщения об ошибке может быть любым, в том числе и пользовательским. Записывается оператор throw следующим образом:

```
throw сообщение;
throw 1;
throw "Error!";
throw 'E';
```

Если в программе нет блока catch, аргумент которого соответствует сообщению об ошибке, то это приводит к ненормальному завершению программы (*Abnormal program termination*).
Рассмотрим несколько примеров:

```
#include <iostream.h>
void main()
{
   cout<<"Начало\n";
try
  {
      cout<<"Блок try\n";
      throw 10;
      cout<<"Не выполняется \n";
  }
catch (char* msg)
   {
      cout<<msg<<endl;
   }
catch(int n)
   {
      cout<<"Ошибка номер"<<n<<endl;
   }
```

```
cout<<"Конец\n";
}
```

```
Начало
Блок try
Ошибка номер 10
Конец
```

Как видно из примера, было перехвачено сообщение типа int, а блок catch с типом char* был проигнорирован.

Исключительная ситуация может быть вызвана из не входящего в блок **try** оператора, если сам этот оператор входит в функцию, которая вызывается из блока **try**:

```
#include <iostream.h>
void TEST(int val)
{
      cout<<"Внутри TEST "<<val<<"\n";
      if(val) trow val;
}
void main()
{
      try
      {
            TEST(0);
            TEST(1);
            TEST(2);
      }
      catch(int i)
      {
            cout<<"Ошибка !!\n";
            cout<<i<<"\n";
      }
}
```

```
Внутри TEST 0
Внутри TEST 1
Ошибка !!
1
```

Блок **try** можно располагать внутри функции. В этом случае при каждом входе в функцию обработчик исключительной ситуации устанавливается снова.

```
void TEST(int val)
{
      try
      {
            if(val) throw val;
      }
      catch(int i)
      {
            cout<<"Ошибка №"<<i<<"\n";
      }
}
void main()
{
      TEST(1);
      TEST(0);
      TEST(2);
}
```

```
Ошибка №1
Ошибка №2
```

В некоторых случаях необходимо перехватывать все исключительные ситуации, независимо от их типа. Для этого используют следующую форму блока **cath**:

```
cath(...)
{
      //обработка всех исключительных ситуаций
}
```

Многоточие соответствует любому типу данных.

Для функции, вызываемой из блока **try**, можно ограничить число типов исключительных ситуаций, которая способна сгенерировать функция. Таким образом можно даже запрещать генерировать некоторые типы исключительных ситуаций.

Возвращаемый_тип имя_функции (аргументы) **throw**(список_типов)
{

}

Генерация любого другого типа исключительной ситуации приведет к аварийному завершению программы. Если необходимо, чтобы функция не генерировала никаких исключительных ситуаций, то можно список типов сделать пустым.

Очень удобно оператор **cath(...)** использовать в качестве последнего оператора в группе операторов **catch**. Т.е. он будет перехватывать все остальное. Кроме этого, путем перехвата всех исключительных ситуаций, предотвращается аварийное завершение программы из-за необработанной исключительной ситуации.

Если генерируется исключительная ситуация, для которой нет соответствующего оператора **catch**, то произойдет вызов функции **terminate()**, которая в свою очередь вызовет функцию **abort()**.

Последний момент, если необходимо повторно сгенерировать исключительную ситуацию из процедуры обработки исключительной ситуации (**catch**), можно просто использовать оператор **throw** без параметров. Это приведет к тому, что текущая исключительная ситуация передастся внешней последовательности **try/catch.**

```
void TEST()
{
	try
	{
		thorw "Ошибка!\n";
	}
	catch(char*)
	{
		cout<<"Перехват ошибки внутри TEST\n";
		throw;
	}
}
void main()
{
	try
	{TEST();}
	catch(char*)
	{cout<<"Перехват ошибки внутри main\n";}
}
```

```
Перехват ошибки внутри TEST
Перехват ошибки внутри main
```

На последок рассмотрим возможность обработки исключительной ситуации в BC.3.1 касающейся выделения памяти с помощью оператора new.
Это функция set_new_handler() (NEW.H).
Прототип этой функции выглядит следующим образом:

void (* set_new_handler(void (* my_handler)()))();

Эта функция запускает обработчик пользователя my_handler() в том случае, если оператор new не может выделить память. По умолчанию оператор new возвращает значение "0" в случае когда не удается выделить память требуемого размера.
Для установки обработчика, который используется по умолчанию необходимо указать следующее: set_new_handler(0);

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>
void TEST() {cerr <<"Невозможно выделить память! \n"; exit(1);}
void main()
{
 set_new_handler(TEST);
 char *ptr = new char [100];
 cout << "Указатель на блок памяти: ptr = "<<hex<<long(ptr)<<"\n";
 ptr = new char[1024*1024*200];
 cout << "Указатель на блок памяти: ptr = "<<hex<<long(ptr)<<"\n";
```

```
 set_new_handler(0);
}
_____________________________________
}
```

Лекция-14:

1. БИБЛИОТЕКА СТАНДАРТНЫХ ШАБЛОНОВ (STANDARD TEMPLATES LIBRARY (STL)).

При рассмотрении вопроса, связанного с STL необходимо выделить некоторые ключевые понятия: контейнеры, итераторы и алгоритмы. Под контейнером понимают объекты, предназначенные для хранения объектов других (произвольных) типов. Тип контейнера может быть любым. Так, в STL описаны следующие типы контейнеров:

bitset	*множество битов*
deque	*двунаправленная очередь*
list	*линейный список*
map	*ассоциативный список (ключ -> одно значение)*
multimap	*ассоциативный список (ключ -> несколько значений)*
multiset	*множество*
priority_queue	*очередь с приоритетом*
queue	*очередь*
set	*множество уникальных элементов*
stack	*стек*
vector	*вектор (динамически изменяемый стек)*

Ассоциативный контейнер отличается от остальных типов тем, что значение или значения, хранящиеся в контейнере, могут быть доступны только по определенному ключу. (Функционирование аналогично ассоциативному запоминающему устройству). Каждому контейнеру принадлежит набор уникальных функций, отвечающих за манипулирование с содержимым контейнера. Так, в каждом контейнере определены функции, отвечающие за добавление элемента в контейнер и удаление элемента из контейнера. Итераторы - указатели на контейнер. Используется для доступа к элементам контейнера. Выделяют итераторы 5 типов:

random access (RandIter)	*считывание и запись элементов с произвольным доступом*
bidirectional (BiIter)	*считывание и запись элементов, два направления прохода контейнера*
forward (ForIter)	*считывание и запись элементов, однонаправленный проход контейнера*
input (InIter)	*считывание элементов, однонаправленный проход контейнера*
output (OutIter)	*запись элементов, однонаправленный проход контейнера*

Использование итераторов аналогично использованию обычных указателей на типы, т.е. итераторы как и указатели можно инкрементировать и т.д. Алгоритмы осуществляют операции над элементами контейнеров (сортировка, поиск, замена и т.д.). Для каждого контейнера определен так называемый allocator (распределитель памяти), который управляет процессом выделения памяти под конкретный контейнер. По умолчанию распределитель памяти для каждого контейнера является экземпляром класса allocator. Однако можно определить и свой собственный распределитель памяти. Рассмотрим более подробно механизмы использования контейнеров на конкретном примере - контейнер vector.

2. КОНТЕЙНЕР VECTOR.

Контейнер vector определяет динамически изменяемый стек. Прототип шаблона класса vector выглядит следующим образом:

```
template <class Type, class Allocator = allocator <Type>> class vector
```

Type - тип хранящихся данных в контейнере, Allocator - распределитель памяти, по умолчанию - стандартного типа allocator. Класс vector имеет несколько перегруженных конструкторов:

- конструктор, создающий пустой вектор:

vector (const Allocator &alloc = Allocator())

- конструктор, создающий вектор с инициализацией каждого элемента:

vector (size_type number, const Type &what = Type(), const Allocator &alloc = Allocator ())

- конструктор, создающий вектор из объектов одинакового типа:

vector (const vector <Type, Allocator> &object)

- конструктор, создающий вектор по диапазону

template <class InIter> vector (InInter Begin, InInter End, const Allocator &alloc = Allocator ())

Если элементом вектора является экземпляр класса, то в описании такого класса обязательно должен присутствовать конструктор по умолчанию и перегружены необходимые операторы. Для встроенный типов все операторы перегружены по умолчанию. В самом шаблоне vector перегружены операторы сравнения <,>,<=,>=, !=, = = и оператор индексации [ ]. Рассмотрим подробнее некоторые функции-члены шаблона класса vector.

template <class InInter> void assign (InInter Begin, InInter End);	*присваивает вектору последовательность, определенную итераторами Begin и End*
reference at(size_type i);	*возвращает ссылку на i-ый элемент контейнера*
reference back();	*возвращает ссылку на последний элемент*
reference front();	*возвращает ссылку на первый элемент*
iterator begin();	*возвращает итератор первого элемента*
iterator end();	*возвращает итератор последнего элемента*
void clear();	*удаляет все элементы вектора*
iterator erase(iterator i);	*удаляет элемент, на который указывает итератор i*
reference operator [] (size_type i) const;	*возвращает ссылку на i-ый элемент*
void pop_back();	*удаляет последний элемент вектора*
void push_back(const Type &value);	*добавляет в конец вектора элемент*
size_type size() const;	*возвращает текущее количество элементов*

и т.д.

Как можно было заметить в описании шаблона vector присутствуют переопределенные типы. Ниже приведены расшифровки некоторых из них:

reference	*ссылка на элемент вектора*
const_reference	*константная ссылка на элемент вектора*
iterator	*итератор*
const_iterator	*константный итератор*
value_type	*тип элемента вектора*
allocator_type	*тип распределителя памяти (по умолчанию allocator)*
key_type	*тип ключа для ассоциативных контейнеров*

и т.д.

Для закрепления материала рассмотрим пример использования стандартного шаблона vector.

```
#include <iostream.h>
#include <vector.h>                    //подключение стандартной библиотек
                                       //vector
main()
{
 vector<int> m;                        //пустой вектор элементов типа int
 vector<int>::iterator ptr;            //итератор
 int buf=0;
 while(cout<<"Input:",cin>>buf,buf!=0) m.push_back(buf); //добавление элементов
 cout<<"Vector size:"<<m.size()<<endl; //вывод количества элементов
 cout<<"Vector value:";
 for(int i=0;i<m.size();cout<<m[i++]); cout<<endl; //вывод самих элементов
 cout<<"Vector value, part#2:";
 ptr=m.begin();                        //итератор указывает на начало вектора
 while(ptr!=m.end()) cout<<*ptr++; cout<<endl; //вывод при помощи итератора
 cout<<"Delete all elements\n";
 m.erase(m.begin(),m.end());           //удаление всех элементов
```

```
 cout<<"Vector size:"<<m.size()<<endl; //вывод количества элементов
}
```

Пример использования контейнера vector для хранения экземпляров классов.

```
#include <iostream.h>
#include <vector.h>
class Number
{
 int n;
 public:
 Number() {n=0;}
 Number(int n){this->n=n;}
 friend ostream& operator << (ostream&,Number&);
};
ostream& operator << (ostream &os, Number &obj)
{
 return (os<<" "<<hex<<obj.n);
}

main()
{
 vector<Number> N(5,0xff);
 vector<Number>::iterator ptr;
 cout<<"Vector size:"<<N.size()<<endl;
 ptr=N.begin();
 for(int i=0;i<5;cout<<N[i++]);
 cout<<"\nVector value:";
 while(ptr!=N.end()) cout<<*ptr++;
 cout<<endl;
}
```

3. АЛГОРИТМЫ.

Как уже было отмечено, каждый контейнер содержит алгоритмы, необходимые для работы с элементами. Все алгоритмы представляют собой шаблоны функций. Для использования стандартных алгоритмов STL необходимо подключить библиотеку <algorithm.h>. Описанные в ней алгоритмы применимы к любому типу контейнеров. Рассмотрим пример использования стандартных алгоритмов для нашего первого примера.

```
#include <iostream.h>
#include <vector.h>
#include <algorithm>//<algo.h>
main()
{
vector<int> m;
vector<int>::iterator ptr;
int buf=0;
while(cout<<"Input:",cin>>buf,buf!=0) m.push_back(buf);
cout<<"Vector size:"<<m.size()<<endl;
cout<<"Vector value:";
for(int i=0;i<m.size();cout<<m[i++]); cout<<endl;
random_shuffle(m.begin(),m.end()); //случайное перемешивание
                                   //элементов определенного диапазона
cout<<"Vector value, part#2:";
ptr=m.begin();
while(ptr!=m.end()) cout<<*ptr++; cout<<endl;
ptr=max_element(m.begin(),m.end());//возвращает итератор максимального
                                   //элемента в диапазоне
cout<<"Maximal element:"<<*ptr<<endl;
cout<<"Delete all elements\n";
m.erase(m.begin(),m.end());
cout<<"Vector size:"<<m.size()<<endl;
}
```