ООП. Наследование

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.

При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Класс, лежащий в основе иерархии, называется базовый (base class) (предок, суперкласс), а класс, наследующий свойства базового класса, -- производным (derived class) (наследник, подкласс). Производные классы, в свою очередь, могут быть базовыми по отношению к другим классам.

Объявление наследования класса:-

class <имя_производного_класса> :
<уровень_доступа имя_базового_класса>
{
<тело_класса>
} ;

уровень_доступа – это public, private или protected. По умолчанию для производного класса private, а для производной структуры – public. Член класса со спецификатором protected недоступен вне класса, но может наследоваться производным классом.

ООП. Наследование

Пример

```
#include <iostream.h>
class building
{   int rooms;        int floors;       int area;
 public:
    void set_rooms (int num);
    int get_rooms();
    void set_floors (int num);
    int get_floors();
    void set_area (int num);
    int get_area();   };
// Класс house – производный от building
class house : public building
{  int bedrooms;  int baths;
 public:
    void set_bedrooms (int num);
    int get_bedrooms();
    void set_baths (int num);
    int get_baths();  };
// Класс school – также производный от building
class school : public building
{  int classrooms;  int offices;
 public:
    void set_classrooms (int num);
    int get_classrooms();
    void set_offices (int num);
    int get_offices();  };
void building :: set_rooms (int num)
    { rooms = num;  }
void building :: set_floors (int num)
    { floors = num;  }
```

// см. продолжение

// продолжение

```
void building :: set_area (int num)
    { area = num;  }
int building :: get_rooms ()
    { return rooms;  }
int building :: get_rooms ()
    { return rooms;  }
int building :: get_floors ()
    { return floors;  }
int building :: get_area ()
   { return area;  }
void house :: set_bedrooms (int num)
   { bedrooms = num;   }
void house :: set_baths (int num)
   { baths= num;   }
int house :: get_bedrooms ()
   { return bedrooms;  }
int house :: get_baths ()
   { return baths;  }
void school :: set_classrooms (int num)
   { classrooms= num;   }
void school :: set_offices (int num)
   {offices= num;   }
int school :: get_classrooms ()
   { return classrooms;  }
int school :: get_offices  ()
   { return offices ;  }
```

// см. продолжение

ООП. Наследование

Пример

```
// продолжение
int main ()
{
  house h;
  school s;
h.set_rooms (12);
h.set_floors (3);
h.set_area (4500);
h.set_bedrooms (5);
h.set_baths (3);
  cout << "В доме " << h.get_bedrooms ();
  cout << "  спален\n";
s.set_rooms (200);
s.set_classrooms (180);
s.set_offices (5);
s.set_area (25000);
  cout << "В школе" << s.get_classrooms ();
  cout << "  кабинетов\n";
  cout <<"Её площадь равна  " << s.get.area ();
   return 0;
}
```

Вывод на экран:

В доме 5 спален
В школе 180 кабинетов
Её площадь равна 25000

Замечания:

1. **Открытые члены класса building становятся открытыми членами производных классов house и school.**
2. **НО, методы (функции-члены) классов house и school *не имеют доступа* к закрытым членам класса building.**

То есть, наследование не нарушает инкапсуляцию.

ООП. Наследование

Простое (или одиночное) наследование – это наследование, при котором производный класс имеет только одного родителя.

В Примере на предыдущих слайдах как раз реализовано простое наследование. Формально наследование одного класса от другого можно задать следующей конструкций:

class имя_класса_потомка: [модификатор_доступа]
имя_базового_класса
{ тело__класса }

Класс-потомок наследует структуру (все элементы данных) и поведение (все методы) базового класса. Модификатор доступа определяет доступность элементов базового класса в классе-наследнике. Этот модификатор мы будем называть модификатором наследования.

Если в качестве модификатора наследования записано слово public, то такое наследование открытое. При использовании модификатора protected – защищенное наследование, а private означает закрытое наследование.

ООП. Наследование

Множественное наследование отличается от простого (одиночного) наличием нескольких базовых классов:

```
class A {}; class B {};
class D: public A, public B
```

Базовые классы перечисляются через запятую; количество их стандартом не ограничивается. Модификатор наследования для каждого базового класса может быть разный: можно от одного класса наследовать открыто, а от другого – закрыто.

При множественном наследовании выполняется все то же самое, что и при одиночном, то есть класс-потомок наследует структуру (все элементы данных) и поведение (все методы) всех базовых классов.

ООП. Наследование

Конструкторы при наследовании

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

Порядок вызова конструкторов определяется приведенными ниже правилами:

- **Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров).**
- **Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.**
- **В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.**

ВНИМАНИЕ!

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации.

ООП. Наследование

Деструкторы при наследовании

Правила наследования деструкторов

- Деструкторы **не наследуются**, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.
- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.
- Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

ООП. Наследование

Домашние задания

1. По 1-у индивидуальному домашнему заданию каждому студенту.

ООП. Полиморфизм.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы:

один интерфейс, несколько методов.

Простым примером полиморфизма может служить перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам.

С перегрузкой функций тесно связан механизм перегрузки операторов (операций), применяемый для настройки их на конкретный класс. Перегруженный оператор сохраняет своё первоначальное предназначение, Просто набор типов данных, к которым его можно применять, расширяется.

Например, в классе stack можно перегрузить оператор "+" для заталкивания элемента в стек, а оператор "-" для выталкивания элемента из стека.

Ещё пример – использование шаблонов функций (и шаблонов классов), когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра.

ООП. Полиморфизм.
Перегрузка операций

Перегрузка функций (function overloading) – это использование одного имени для нескольких функций, отличающихся либо другими типами данных параметров, либо другим количеством параметров.
Перегрузка функций является особенностью языка C++, которой нет в языке C, это одна из разновидностей полиморфизма.

Основные концепции перегрузки функций:

- Перегрузка функций предоставляет несколько "взглядов" на одну и ту же функцию внутри вашей программы.
- Для перегрузки функций просто надо определить несколько функций с одним и тем же именем, которые отличаются только количеством и/или типом параметров.
- Компилятор C++ определит, какую функцию следует вызвать, основываясь на количестве и типе передаваемых (фактических) параметров.
- Перегрузка функций упрощает программирование, позволяя программистам работать только с одним именем функции.

Правила описания перегруженных функций:

- Перегруженные функции должны находиться в одной области видимости,
- Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать,
- Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или использованием ссылки (например, int и const int или int и int&).

ООП. Полиморфизм. Перегрузка функций

ПРИМЕР: надо сделать функции поиска в базе данных, каждая функции имеет осмысленное имя:

```
int Search_int(int Num);
int Search_long(long Num);
int Search_float(float Num);
int Search_double(double Num);
```

Но гораздо проще дать всем этим функциям одно имя для поиска всех типов данных. Например:

```
int Search(int Num);
int Search(long Num);
int Search(float Num);
int Search(double Num);
```

Заметим - имена функций одинаковы, отличие только в **типах аргументов**. **Цель перегрузки функций состоит в том, чтобы функции с одним именем ВЫПОЛНЯЛИСЬ ПО-РАЗНОМУ (и возможно возвращали разные значения) при обращении к ним с разными по типам и количеству параметрами.** В языке C++ функции могут иметь одинаковые имена до тех пор, пока они значимо отличаются хотя бы одним параметром. Если значимого различия нет – компилятор предупредит о возникшей неопределенности.

Необходимо отличать перегрузку и переопределение функции.

Если функции возвращают одинаковый тип и список параметров у них абсолютно одинаковый, то второе объявление функции будет обработано как повторное определение. Если списки параметров двух функций абсолютно одинаковы, но отличаются только типы возврата, то второе объявление - ошибка:

```
int  Search(int Num);
long Search(int Num);  //ошибка!
```

ООП. Полиморфизм. Перегрузка функций

Как компилятор решает, какую именно функцию надо использовать для данного вызова (запроса).

Компилятор находит соответствие автоматически, сравнивая фактические параметры, используемые в запросе, с параметрами, предлагаемыми каждой функцией из набора перегруженных функций. Рассмотрим набор функций и один вызов функции:

```
void calc();
void calc(int);
void calc(int, int);
void calc(double, double=1.2345);
```

calc(5.4321); //вызвана будет функция void calc(double, double)

Как в этом случае компилятор ведет поиск соответствия функции?

Сначала определяются функции-кандидаты на проверку соответствия. Функция-кандидат должна иметь то же имя, что и вызываемая функция, и ее объявление должно быть видимым в точке запроса.

Затем определяются "жизнеспособные" функции. Они должны или иметь то же самое количество параметров что и в запросе, или тип каждого параметра должен соответствовать параметру в запросе (или быть конвертируемым типом).

Для нашего запроса calc(5.4321), мы можем сразу выкинуть функции-кандидаты calc() и calc(int, int). Запрос имеет только один параметр, а эти функции имеют нуль и два параметра, соответственно. calc(int) - вполне "жизнеспособная" функция, потому что можно конвертировать тип параметра double к типу int. Функция calc(double, double) тоже "жизнеспособная", потому что заданный по умолчанию параметр обеспечивает "якобы недостающий" второй параметр функции, а первый параметр имеет тип double, который точно соответствует типу параметра в вызове.

Потом компилятор определяет, какая из найденных "жизнеспособных" функций имеет "явно лучшее" соответствие фактическим параметрам в запросе. Что понимать под "явно лучшим"? Идея состоит в том, что чем ближе типы параметров друг к другу, тем лучше соответствие. То есть, точное соответствие типа лучше чем соответствие, которое требует преобразования типа.

В нашем запросе calc(5.4321) - только один явный параметр, и он имеет тип double. Чтобы вызвать функцию calc(int), параметр должен быть преобразован в int. Функция calc(double, double) является более точным соответствием для этого параметра. И именно эту функцию использует компилятор.

ООП. Полиморфизм. Перегрузка функций

Поиск соответствия функции усложняется, если в вызове имеется не один, а два параметра или более. Будем использовать тот же набор перегруженных функций. Давайте проанализируем следующий запрос:

calc(123, 5.4321);

С функциями-кандидатами все ясно – набор не изменился. Проверим "жизнеспособные" функций. Компилятор выбирает те функции, которые имеют требуемое количество параметров и для которых типы параметров соответствуют параметрам вызова. "Жизнеспособных" функций опять две. Это calc(int, int) и calc(double, double).

Далее компилятор проверяет параметр за параметром. Соответствие считается найденным, если есть ОДНА И ТОЛЬКО ОДНА функция для которой:

- соответствие для каждого параметра - не хуже чем соответствие, требуемое любой другой "жизнеспособной" функцией;
- есть не менее одного параметра, для которого соответствие лучше, чем соответствие, обеспеченное любой другой "жизнеспособной" функцией.

Если не найдется полностью соответствующей функции – запрос неоднозначен.

Смотрим первый параметр в вызове – функция calc(int, int) точно соответствует по типу первому параметру. Чтобы соответствовать второй функции calc(double, double), int параметр "123" в вызове должен быть преобразован в double. Соответствие через такое преобразование "менее хорошо". Делаем вывод - calc(int, int) имеет лучшее соответствие.

Посмотрим на второй параметр. Тогда получится, что функция calc(double, double) имеет точное соответствие параметру "5.4321". Теперь для функции calc(int, int) потребуется преобразование из double в int. А мы уже знаем, что соответствие через такое преобразование "менее хорошо". Делаем вывод - calc(double, double) имеет лучшее соответствие.

НО, такой результат получен и для первого параметра типа int.

Вот поэтому запрос и неоднозначен - обе функции имеют соответствие запросу по одному из параметров. Компилятор сгенерирует ошибку.

Мы могли бы сами создать соответствие, используя явное приведение типов одного из параметров в вызове функции:

```
calc(static_cast<double>(123), 5.4321);  // будет вызвана функция calc(double, double)
```

или так:

```
calc(123, static_cast<int>(5.4321));     // будет вызвана функция calc(int, int)
```

Вообще говоря, параметры не должны нуждаться в явном приведении типов при вызове перегруженных функций. Потребность в приведении типов означает, что наборы параметров были плохо разработаны.

ООП. Полиморфизм. Перегрузка функций

Примеры

```
// Различные типы данных параметров:
//   выводится на экран значение параметра

#include <iostream.h>
// прототипы функций
  int refunc(int i);
  double refunc(double i);

int main()
{
  cout  <<  refunc(10)  <<  " ";
  cout  << refunc(5.4);
return 0;
}

// Функция для целых типов данных
int  refunc(int i)
{  return i; }
// Функция для данных двойной точности
double  refunc(double  i)
{  return i; }
```

```
// Различное количество параметров:
// выводится при одном параметре  значение
// параметра, а при 2-х – произведение их

#include <iostream.h>
// прототипы функций
  int refunc(int i);
  int refunc(int i, int j);

int main()
{
  cout  <<  refunc(10)  <<  " ";
  cout  << refunc(5, 4);
return 0;
}

// Функция для одного параметра
int  refunc(int i)
{  return i; }
// Функция для двух паарметров
int  refunc(int i, int j)
{  return i*j; }
```

ООП. Полиморфизм.
Перегрузка операторов

Перегрузка операторов (operator overloading) – это изменение смысла оператора, при котором возможно одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Например, оператора плюс (+), который обычно используется для сложения, при использовании его с определенным классом меняет смысл. Для объектов класса string оператор плюс (+) будет добавлять указанные символы к текущему содержимому строки, а оператор минус (-) будет удалять каждое вхождение указанного символа из строки.

Перегрузка операторов осуществляется с помощью операторных функций (operator function), которые определяют действия перегруженных операторов применительно к соответствующему классу. Операторные функции создаются с помощью ключевого слова **operator**.

Операторные функции могут быть как членами класса, так и обычными функциями. Однако обычные операторные функции, как правило, объявляют дружественными по отношению к классу, для которого они перегружают оператор.

ООП. Полиморфизм.
Перегрузка операторов

Операторная функция-член имеет следующий вид:

<тип_возвращаемого_значения> <имя_класса> :: operator# (список-аргументов)
{
... // Операции
}

Обычно операторная функция возвращает объект класса, с которым она работает, однако тип возвращаемого значения может быть любым. Символ # заменяется перегружаемым оператором.

Например, если в классе перегружается оператор деления "/", операторная функция-член называется operator/.

При перегрузке унарного оператора список аргументов остается пустым.

При перегрузке бинарного оператора список аргументов содержит один параметр.

ООП. Полиморфизм.
Перегрузка операторов

ПРИМЕР

Программа создает класс loc, в котором хранятся географические координаты: широта и долгота. В ней перегружается оператор "+".

```
#include <iostream>
using namespace std;
  class loc
   {
    int longitude, latitude;
      public:
        loc() {}
        loc(int lg,  int lt)
          {
           longitude = lg;
           latitude = lt;
          }
         void show()
          {
           cout << longitude << " ";
           cout << latitude << "\n";
          }
        loc operator+(loc op2);
  };
```

см. продолжение

Продолжение

```
// Overload + for loc.
loc loc :: operator+(loc op2)
  {
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude,
   return temp;
  }

int main()
 {
  loc obi(10,  20),  ob2( 5, 30);
  ob1.show(); // Выводит на экран числа 10 20
  ob2.show();  // Выводит на экран числа 5  30
   ob1 = ob1 + ob2;
   ob1.showO;  // Выводит на экран числа 15 50
   return 0;
  }
```

ООП. Полиморфизм. Перегрузка операторов

ПРИМЕР

Функция operator+() имеет только один параметр, несмотря на то, что она перегружает бинарный оператор. Причина в том, что операнд, стоящий в левой части оператора, передается операторной функции неявно с помощью указателя this. Операнд, стоящий в правой части оператора, передается операторной функции через параметр op2.

Отсюда следует важный вывод: при перегрузке бинарного оператора вызов операторной функции генерируется объектом, стоящим в левой части оператора.

Как правило, перегруженные операторные функции возвращают объект класса, с которым они работают. Следовательно, перегруженный оператор можно использовать внутри выражений.

Например, если бы операторная функция operator+() возвращала объект другого типа, следующее выражение было бы неверным:

```
|      ob1 = ob1 + ob2;
```

Для того чтобы присвоить сумму объектов ob1 и ob2 объекту ob1, необходимо, чтобы результат операции+ имел тип loc.

Кроме того, поскольку операторная функция operator+() возвращает объект типа loc, допускается следующее выражение:

```
(obl+ob2).show();  // Выводит на экран сумму obl+ob2
```

В этой ситуации операторная функция создает временный объект, который уничтожается после возвращения из функции show().

Как правило, операторные функции возвращают объекты классов, с которыми они работает.

И еще одно замечание: операторная функция operator+() не модифицирует свои операнды. Поскольку традиционный оператор "+" не изменяет свои операнды, не имеет смысла предусматривать для функции operator+() иное поведение. (Например, 5+7 равно 12, но при этом слагаемые по-прежнему равны 5 и 7.)

ООП. Полиморфизм. Перегрузка операторов

Ограничения на перегруженные операторы

Во-первых, нельзя изменить приоритет оператора.
Во-вторых, невозможно изменить количество операндов оператора. (Однако операнд можно игнорировать.)
В-третьих, операторную функцию нельзя вызывать с аргументами, значения которых заданы по умолчанию.

И, в заключение, нельзя перегружать следующие операторы:

- оператор выбора члена класса **.**
- оператор селектора члена класса **.***
- оператор разрешения области видимости **::**
- тернарный оператор – условная операция **?:**
- статический оператор вычисления длины оператора в байтах **sizeof**

Кроме перечисленных операторов, относящихся к синтаксису языка C++, не разрешается переопределять следующие операторы препроцессора:

- оператор превращения в строку **#**
- оператор конкатенации **##**

Примечание

Операторы # и ## используются в директиве препроцессора #def ine, которую современный стандарт C++ применять не рекомендует. Однако следует помнить о невозможности их перегрузки для классов.

За исключением оператора "=", операторные функции наследуются производными классами. Однако в производном классе каждый из этих операторов снова можно перегрузить.