Вычислительная техника и компьютерное моделирование в физике

Лекция 7

Зинчик Александр Адольфович

zinchik_alex@mail.ru

Основными свойствами ООП являются

- инкапсуляция
- наследование
- полиморфизм.

- Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется *инкапсуляцией* (encapsulation).
- Инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним

- ***Наследование*** — это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы.
- Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях.
- В C++ каждый класс может иметь сколько угодно потомков и предков. Иногда предки называются надклассами или суперклассами, а потомки — подклассами или субклассами.

- ***Полиморфизм*** — возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и выбирать требуемое действие во время выполнения программы.
- Простым примером полиморфизма может служить перегрузка функций.
- Другой пример — использование шаблонов функций (шаблонов классов)
- Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов

Класс является типом данных, определяемым пользователем.

В классе задаются свойства и поведение какого-либо предмета или процесса в виде полей данных (аналогично структуре) и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы.

Свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом.

Интерфейсом класса являются заголовки его методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

- Конкретные величины типа данных «класс» называются ***экземплярами класса,*** или ***объектами***. Объекты взаимодействуют между собой, посылая и получая сообщения.
- ***Сообщение*** — это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций. Таким образом, с помощью ООП легко реализуется так называемая «событийно-управляемая модель», когда данные активны и управляют вызовом того или иного фрагмента программного кода

Класс является абстрактным типом данных, определяемым пользователем.

Данные класса называются ***полями*** (по аналогии с полями структуры), а функции класса — ***методами***.

Поля и методы называются *элементами класса*.

Простейшее описание класса:

```
class <имя>{
   [ private: ]
    <описание скрытых элементов>
   public:
    <описание доступных элементов>
}; // Описание заканчивается точкой с запятой
```

- Спецификаторы доступа **private** и **public** управляют видимостью элементов класса. Элементы, описанные после служебного слова **private**, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию.
- Интерфейс класса описывается после спецификатора **public**.
- Действие любого спецификатора распространяется до следующего спецификатора или до конца класса.
- Можно задавать несколько секций **private** и **public**, порядок их следования значения не имеет.

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором **const**, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны с модификатором **static** но не как **auto**, **extern** и **register**.

Инициализация полей при описании не допускается.

```
class Point{
    int x, y;
  public:
    Point(int dx = 100, int dy  = 10){
    x = dx; y = dy;}
    void draw();
    int get_x(){return x;}
    int get_y(){return y;}
};
```

- Если тело метода определено внутри класса, он является ***встроенным*** (inline). Как правило, встроенными делают короткие методы.
- Если внутри класса записано только объявление (заголовок) метода, сам метод должен быть определен в другом месте программы с помощью операции доступа к области видимости (**::**):

- Все методы класса имеют непосредственный доступ к его скрытым полям, иными словами, тела функций класса входят в область видимости private элементов класса.

- В приведенном классе содержится три определения методов и одно объявление (метод draw).

```
void Point::draw(){
   /* тело метода */
}
```

Метод можно определить как встроенный и вне класса с помощью директивы inline (как и для обычных функций, она носит рекомендательный характер):

```
inline int Point::get_x(){
   return x;}
```

- В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется ***конструктором*** и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта.
- Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

Описание объектов

```
Point p1;                    // Объект класса Point с
                             параметрами по умолчанию
Point p2(200, 300);          // Объект с явной
                             инициализацией
Point plane[100];            // Массив объектов с
                             параметрами по умолчанию
Point *pPoint = new Point (10);
                             //Динамический объект
                             //(второй параметр
                             //задается по умолчанию)
Point& p3 = p1;              // Ссылка на объект
```

- При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор

Доступ к элементам объекта аналогичен доступу к полям структуры.

Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель, например:

```
int n = p1.get_x();
plane[5].draw();
cout << pPoint->get_x();
```

- Обратиться таким образом можно только к элементам со спецификатором public.
- Получить или изменить значения элементов со спецификатором private можно только через обращение к соответствующим методам.

Статические элементы класса

- С помощью модификатора **static** можно описать статические поля и методы класса.
- Их можно рассматривать как глобальные переменные или функции, доступные только в пределах области класса.

Статические поля

- Статические поля применяются для хранения данных, общих для всех объектов класса, например, количества объектов или ссылки на разделяемый всеми объектами ресурс.
- Эти поля существуют для всех объектов класса в единственном экземпляре, то есть не дублируются.

Особенности статических полей

- Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии) и инициализируется с помощью операции доступа к области действия, а не операции выбора (определение должно быть записано вне функций):

```
class A{
   public:
    static int count;
};
int A::count;
```

- Статические поля доступны как через имя класса, так и через имя объекта:

```
A *a, b;
cout << A::count << a->count << b.count;
// Будет выведено одно и то же
```

- На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как private, нельзя изменить с помощью операции доступа к области действия, как описано выше. Это можно сделать только с помощью статических методов (см. ниже).
- Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции sizeof.

Статические методы

- Статические методы предназначены для обращения к статическим полям класса.
- Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель this. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

```
class A{
    static int count;   // Поле count — скрытое
  public:
    static void inc_count(){ count++; }

};
```

```
A::int count;      // Определение в глобальной области

void f(){
   A a;
   // a.count++ — нельзя, поле count скрытое
   // Изменение поля с помощью статического //метода:
   a.inc_count();      // или A::inc_count();
}
```

Статические методы не могут быть константными (const) и виртуальными (virtual).

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании.

Основные свойства конструкторов.

- Конструктор ***не возвращает*** *значение*, даже типа void. Нельзя получить указатель на конструктор.
- Класс может иметь *несколько* ***конструкторов*** с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
- Конструктор, вызываемый без параметров, называется ***конструктором по умолчанию***.

- ***Параметры конструктора*** могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
- Если в классе не указан ни один конструктора, компилятор создает его *автоматически.* Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов.

- В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.

- *Конструкторы не наследуются*.

- Конструкторы нельзя описывать с модификаторами const, virtual и static.

- Конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия.
- Конструктор запускается при создании временного объекта (например, при передаче объекта из функции (или в функцию)).

Конструктор вызывается, если в программе встретилась какая-либо из следующих синтаксических конструкций:

- **имя_класса имя_объекта [(список параметров)];**

// Список параметров не должен быть пустым

- **имя_класса (список параметров);**

// Создается объект без имени (список может быть пустым)

- **имя_класса имя_объекта = выражение;**

// Создается объект без имени и копируется

Деструктор

- Деструктор — это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. *Деструктор вызывается автоматически*, когда объект выходит из области видимости:
- для *локальных* объектов — при выходе из блока, в котором они объявлены;
- для *глобальных* — как часть процедуры выхода из main;
- для *объектов, заданных через указатели*, деструктор вызывается неявно при использовании операции delete.

- Автоматический вызов деструктора объекта при выходе из области действия указателя на него не производится.

- Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

- Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически — иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса.

Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как **const** или **static**;
- не наследуется;
- может быть виртуальным

- **Присваивание объектов**
- Один объект можно присвоить другому, если оба объекта одинакового типа. Когда объект **A** присваивается объекту **B**, то осуществляется побитовое копирование всех переменных-членов **A** в соответствующие переменные-члены **B**.

пример:

```
#include <iostream.h>
class MyClass
{
   int a, b;
public:
   void Set(int i, int j) { a = i; b = j; }
   void Show() { cout << a << “   “ << b << endl; }
};
```

```
int main()
{
   MyClass ob1, ob2;
   ob1.Set(10, 77);
   ob2 = ob1;
   ob1.Show();
   ob2.Show();
   return 0;
}
```

Эта программа в процессе выполнения выведет:

10 77
10 77

При присваивании одного объекта другому необходимо быть очень внимательным!!!

Конструктор копирования

Одной из важнейших форм перегружаемого конструктора является *конструктор копирования (copy constructor)*.

Конструктор копирования вызывается в трех ситуациях:

- а) когда в операторе объявления один объект используется для инициализации другого,
- б) когда объект передается (по значению) в функцию в качестве параметра,
- в) когда создается временный объект для возврата значения из функции.

- Если при объявлении класса конструктор копирования не задан, то компилятор C++ создает *конструктор копирования по умолчанию*, который просто дублирует объект, осуществляя побитовое копирование.
- Однако при этом возможно появление серьезных проблем, связанных с копированием указателей.

Любой конструктор копирования имеет следующую общую форму:

```
имя_класса (const имя_класса & obj )
{
    // тело конструктора
}
```

Здесь *obj* – это ссылка на объект, используемый для инициализации другого объекта. Например, пусть имеется класс ***MyClass***, а ***y*** – это объект типа ***MyClass*.**

Тогда следующие операторы вызовут конструктор копирования класса ***MyClass***:

```
MyClass  x = y;      // y явно инициализирует x
func1(y);            // y передается в качестве параметра (по значению)
y = func2();         // y получает возвращаемый объект.
```

- В двух первых случаях конструктору копирования передается ссылка на ***y***. В последнем случае конструктору копирования передается ссылка на объект, возвращаемый функцией ***func2()***.

Указатель this

- Каждый объект содержит свой экземпляр полей класса.
- Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны.
- Это обеспечивается передачей в функцию скрытого параметра **this**, в котором хранится константный указатель на вызвавший функцию объект.

- Указатель **this** неявно используется внутри метода для ссылок на элементы объекта.
- В явном виде этот указатель применяется в основном для возвращения из метода указателя (return this;) или ссылки (return *this;) на вызвавший объект.

Перегрузка операций

- C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции.
- Это дает возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить нельзя. Можно перегружать любые операции, существующие в C++, за исключением:

- . .* ?: :: # ## sizeof

Перегрузка операций осуществляется с помощью методов специального вида :

- при перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных;
- для стандартных типов данных переопределять операции нельзя;
- функции-операции не могут иметь аргументов по умолчанию;
- функции-операции наследуются (за исключением =);
- функции-операции не могут определяться как static.

- *Функцию-операцию можно определить тремя способами*: она должна быть либо методом класса, либо дружественной функцией класса, либо обычной функцией. В двух последних случаях функция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс.
- функция-операция, первый параметр которой имеет стандартный тип, не может определяться как метод класса.

- Функция-операция содержит ключевое слово operator, за которым следует знак переопределяемой операции:

- **тип operator операция (список параметров) { тело функции }**

Перегрузка унарных операций

- Унарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

```
class monstr{
   monstr & operator ++()
   {++health; return *this;}
   }
monstr Vasia;
cout << (++Vasia).get_health();
```

- Если операция может перегружаться как внешней функцией, так и функцией класса, какую из двух форм следует выбирать?
- Ответ: используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, int), то перегрузка операции возможна только в форме внешней функции.

Перегрузка операций инкремента

- Операция инкремента имеет две формы: *префиксную* и *постфиксную*. Для первой формы сначала изменяется состояние объекта в соответствии с данной операцией, а затем он (объект) используется в том или ином выражении. Для второй формы объект используется в том состоянии, которое у него было до начала операции, а потом уже его состояние изменяется.
- Чтобы компилятор смог различить эти две формы операции инкремента, для них используются разные сигнатуры, например:

```
Point& operator ++();          // префиксный инкремент
Point operator ++(int);        // постфиксный инкремент

//Покажем реализацию данных операций на примере класса /Point/


Point& Point::operator ++() {
    x++;      y++;
    return *this;
}

Point Point::operator ++(int) {
    Point old = *this;
    x++;      y++;
    return old;
}
```

- Обратите внимание, что в префиксной операции осуществляется возврат результата по ссылке.
- Это предотвращает вызов конструктора копирования для создания возвращаемого значения и последующего вызова деструктора.
- В постфиксной операции инкремента возврат по ссылке не подходит, поскольку необходимо вернуть первоначальное состояние объекта, сохраненное в локальной переменной old. Таким образом, префиксный инкремент является более эффективной операцией, чем постфиксный инкремент.

- Заметим, что ранее во всех примерах использовалась *постфиксная* форма инкремента:
- **for (i = 0; i < n; i++);**
- Дело в том, что пока параметр i является переменной встроенного типа, форма инкремента безразлична: программа будет работать одинаково. Ситуация меняется, если параметр i есть объект некоторого класса — в этом случае префиксная форма инкремента оказывается более эффективной.

Перегрузка бинарных операций

Бинарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

```
class Point{

    bool operator >(const Point &P){
     if( x > P.get_x() && y > P.get_y() ) return true;
     return false;
    }
};
```

Если функция определяется *вне класса*, она должна иметь два параметра типа класса:

```
bool operator >(const Point &P1, const Point &P2){
    if( P1.get_x() > P2.get_x() &&
     P1.get_y() >P2.get_y() )  return true;
    return false;
}
```

```
class Point {
    double x, y;
public:
//. . .
Point operator +(Point&);
};
Point Point::operator +(Point& p) {
    return Point(x + p.x, y + p.y);
}
```

- Независимо от формы реализации операции «+» мы можем теперь написать:

- Point p1(0, 2), p2(-1, 5);
- Point p3 = p1 + p2;

- Встретив выражение p1 + p2, компилятор в случае первой формы перегрузки вызовет метод p1.operator +(p2), а в случае второй формы перегрузки — глобальную функцию operator +(p1, p2);

Перегрузка операции присваивания

- Перегрузка этой операции имеет ряд особенностей.

- Во-первых, если вы не определите эту операцию в некотором классе, то компилятор создаст операцию присваивания по умолчанию, которая выполняет поэлементное копирование объекта.
- В этом случае возможно появление тех же проблем, которые возникают при использовании конструктора копирования по умолчанию.

- Поэтому существует **правило**: если в классе требуется определить конструктор копирования, то должна быть перегруженная операция присваивания, и наоборот.

- Во-вторых, операция присваивания может быть определена только в форме метода класса.
- В-третьих, операция присваивания не наследуется (в отличие от всех остальных операций).

```
class Man {
public:
    Man(char* Name, int by=1950, float p=1000) ;

    ~Man() { delete [] pName; }
…
Man& operator =(const Man&);
…
private:
    char* pName;
    int birth_year;
    float pay;
};
```

```
Man& Man::operator =(const Man& man)
{
   if (this == &man) return *this;  // проверка на
                                    //самоприсваивание
   delete [] pName;                 // уничтожить
                                    //предыдущее значение

   pName = new char[strlen(man.pName) + 1];
   strcpy(pName, man.pName);
   birth_year = man. birth_year;
   pay = man.pay;
   return *this;
}
```

- Необходимо обратить внимание на несколько простых, но важных моментов при реализации операции присваивания:

1. Убедиться, что не выполняется присваивание вида x = x;. Если левая и правая части ссылаются на один и тот же объект, то делать ничего не надо. Если не перехватить этот особый случай, то следующий шаг уничтожит значение, на которое указывает pName, еще до того, как оно будет скопировано;
2. Удалить предыдущие значения полей в динамически выделенной памяти;
3. Выделить память под новые значения полей;
4. Скопировать в нее новые значения всех полей;
5. Возвратить значение объекта, на которое указывает this (то есть *this).

- Возврат из функции указателя на объект делает возможной цепочку операций присваивания:
- Man A(“Alpha”), B(“Bravo”), C(“Charlie”);
- C = B = A;
- Операцию присваивания можно определять только как метод класса.