Семинар 6. Классы, статические поля и методы, конструкторы и деструкторы, статусы доступа, внешнее определение методов, друзья классов, указатель this

1. Классы

Совокупность принципов проектирования, разработки и реализации программ, которая базируется на абстракции данных, предусматривает создание новых типов данных, с наибольшей полнотой отражающих особенности решаемой задачи. В языке Си++ программист может вводить собственные типы данных и определять операции над ними с помощью классов.

1. Классы

Класс – это производный структурированный тип, введённый программистом на основе уже существующих типов. Класс задаёт структурированную совокупность типизированных данных и позволяет определить набор операций над этими данными.

1. Классы

Класс можно определить с помощью спецификации класса:
ключ_класса имя_класса
{поля данных и методы класса
(также называемые как «тело класса»)};

ключ_класса – одно из ключевых слов class, struct, union
имя_класса – произвольно выбираемый пользователем идентификатор

1. Классы

В теле спецификации класса могут быть:

- данные (также: свойства, поля, элементы данных (data members))
- методы (функции класса, компонентная функция (member function))
- классы
- перечисления
- битовые поля
- дружественные функции
- дружественные классы

1. Классы

Пример.

```
struct rectangle{
        double width;
        double height;
        double perimeter() {return width*2 + height*2;}
        void printData(){
                cout << "width = " << width << endl;
                cout << "height = " << height << endl;
        }
};

int main()
{       // Определение объекта класса:
        rectangle R1; // Можно инициализировать: rectangle R1 = {5, 3.5};
        R1.width = 5; R1.height = 3.5; // Присваивание значений в поля данных
        R1.printData(); // Вызов метода
        cout << "\nPerimeter = " << R1.perimeter();
        return 0;
}
```

1. Классы

В предыдущем примере был определён новый тип rectangle. Можно создавать объекты этого типа и создавать производные типы:

- rectangle R1; // создание экземпляра класса (объекта)
- rectangle * R1ptr = &R1; // создание указателя на объект
- rectangle rectArr[5]; // создание массива экземпляров
- rectangle & R1ref = R1; // создание ссылки на объект

1. Классы

Обращение к полям данных объектов:
имя_объекта.имя_класса::имя_поля_данных
имя_объекта.имя_поля_данных
указатель_на_объект_класса -> имя_класса::имя_поля_данных
указатель_на_объект_класса -> имя_поля_данных

Обращение к методам объектов:
имя_объекта.имя_класса::имя_метода(аргументы)
имя_объекта.имя_метода(аргументы)
указатель_на_объект_класса -> имя_класса::имя_метода(арг.)
указатель_на_объект_класса -> имя_метода(аргументы)

1. Классы

Некоторые особенности:

• В отличие от обычного определения данных (переменных), при описании полей класса невозможна их инициализация. Это объясняется тем, что при определении класса ещё не существует участков памяти, соответствующих его полям данных. Память выделяется не для класса, а только для объектов класса.

• Принадлежащие классу функции (методы) имеют полный доступ к полям и методам этого класса.

• Данные и методы класса не обязательно должны быть определены или описаны до (выше) их первого использования в методах класса. Все поля данных и методы класса «видны» во всех операторах тела любого метода класса.

1. Классы

Задание.

1. Дополнить класс rectangle методами вычисления площади прямоугольника, полупериметра, длины диагонали; добавить в этот класс поле angle (угол), хранящий информацию об угле поворота прямоугольника (в градусах), и метод rotate(double grad), изменяющий значение угла поворота на число градусов grad.
2. Создать класс circle (окружность), имеющий поле r – радиус окружности, добавить в этот класс методы получения диаметра, длины окружности, площади круга.

2. Статические поля данных и методы

Каждый объект одного и того же класса имеет собственную копию нестатических полей данных класса.

Поле данных *создаётся в единственном экземпляре и не тиражируется при создании каждого нового объекта класса*, если оно было определено в классе как *статическое*, т.е. имеет атрибут static.

2. Статические поля данных и методы

Статические данные класса можно использовать в программе ещё до определения объектов данного класса.
Доступ к статическому полю данных возможен только после его инициализации (это поле инициализируется как глобальная переменная):

тип имя_класса::имя_поля_данных инициализатор;

2. Статические поля данных и методы

Статические методы сохраняют все основные особенности обычных (нестатических) методов класса. Статический метод может быть вызван с помощью квалифицированного имени:
имя_класса::имя_статического_метода()

2. Статические поля данных и методы

Пример. Программа, использующая статическое поле для определения наценки всех товаров.

```
struct goods{
        char name[40];
        double price;
        static double percent; // Статическое поле данных – одно на все объекты
        void display(){
                cout << name << " price = " << \
                price * (1.0 + goods::percent/100.0) << endl;
        }
};
double goods::percent = 0; // Инициализация статического поля данных
int main()
{
        goods G1 = {"Milk", 50}, G2 = {"Honey", 800};
        G1.display(); G2.display();
        goods::percent = 10; // Изменение значения статического поля данных
        G1.display(); G2.display();
        getchar();
        return 0;
}
```

3. Конструкторы и деструкторы

Для инициализации объектов класса (присваивания полям объекта некоторых значений) в определение этого класса можно включать специальный метод – конструктор. Формат конструктора:

имя_класса(список_параметров)
инициализатор_конструктора
{операторы_тела_конструктора}

3. Конструкторы и деструкторы

Особенности конструктора:

- имя совпадает с именем класса
- конструктор явно или неявно *вызывается при определении каждого объекта класса*
- назначение конструктора – инициализация полей данных
- для конструктора не определяется тип возвращаемого значения
- нельзя получить адрес конструктора, нельзя вызвать как обычный метод

3. Конструкторы и деструкторы

Две синтаксические формы явного вызова конструктора:

имя_класса имя_объекта(арг_конструктора)

или

имя_класса(арг_конструктора)

Пример применения второй формы:

```
rectangle r = rectangle(5.0, 6.0);
Rect = new rectangle(6.0, 7.0);
```

3. Конструкторы и деструкторы

Пример использования конструктора без инициализатора.

```
struct rectangle{
        double width;
        double height;
        rectangle(double w = 0.0, double h = 0.0){
                width = w;
                height = h;
        }
        double perimeter() {return width*2 + height*2;}
        void printData(){
                cout << "width = " << width << endl;
                cout << "height = " << height << endl;
        }
};
```

3. Конструкторы и деструкторы

Виды конструкторов:

- конструктор копирования (единственный)
- конструкторы приведения типов
- конструктор без параметров (единственный, необязательный)
- конструкторы общего вида

Конструктором умолчания называют такой конструктор, при обращении к которому нет необходимости указывать аргументы.

3. Конструкторы и деструкторы

Если в классе программист не определил ни одного конструктора, то по умолчанию формируется конструктор без параметров и конструктор копирования (последний присутствует всегда) со следующими параметрами:

```
T::T();
T::T(const user_type&);
```

, где T – имя класса.

3. Конструкторы и деструкторы

Если в классе программист определил хотя бы один конструктор, то конструктор без параметров не создаётся. Заменить конструктор без параметров может конструктор, имеющий только умалчиваемые значения (см. пример выше). Оба вида конструкторов могут выполнять роль конструктора умолчания.

3. Конструкторы и деструкторы

Конструктор умолчания используется при следующих определениях объектов:

имя_класса имя_объекта;

имя_класса имя_массива_объектов[размер];

указатель_на_объект_класса = new имя_класса;

указатель_на_объекты_класса = new имя_класса[p];

3. Конструкторы и деструкторы

Пример. Конструктор общего вида, изменяющий значение поля-счётчика объектов.

```
struct rectangle{
        double width;
        double height;
        static int counter;
        static void printCounter(){
                cout << "\n counter = " << counter;
        }
        rectangle(double w, double h) // Конструктор класса
        {width = w; height = h; counter++;}
};
int rectangle::counter = 0;
int main()
{
        rectangle::printCounter();
        rectangle R1(5, 3.5); // Здесь вызывается конструктор!
        R1.printCounter();
        rectangle R2(1, 2); // Здесь вызывается конструктор!
        rectangle::printCounter();
        getchar();
        return 0;
}
```

3. Конструкторы и деструкторы

Пример. Конструктор копирования для класса rectangle.

```
struct rectangle{
    …
    rectangle(const rectangle& rect)
    {width = rect.width; height = rect.height; counter++;}
};
int main()
{
    rectangle R1(5, 3.5); // Конструктор общего вида
    …
    rectangle R2(R1); // Конструктор копирования
    …
    return 0;
}
```

3. Конструкторы и деструкторы

Инициализатор конструктора – ещё один способ инициализации полей данных объекта.

Где расположен инициализатор конструктора?

имя_класса(список_параметров)
инициализатор_конструктора
{операторы_тела_конструктора}

3. Конструкторы и деструкторы

Инициализатор конструктора – список инициализаторов полей данных, перед которым следует двоеточие. Элементы списка располагаются через запятую.

```
rectangle(double w, double h)
{width = w; height = h; counter++;}
```

можно записать как

```
rectangle(double w, double h)
: width(w), height(h)
{counter++;}
```

3. Конструкторы и деструкторы

Особенность инициализатора конструктора: инициализация полей выполняется в порядке их размещения в определении класса.

3. Конструкторы и деструкторы

Последовательность действий при создании нового объекта с привлечением конструктора с инициализатором:

1. Выделяется память для объекта
2. Инициализируются нестатические поля данных объекта с помощью инициализатора конструктора
3. Исполняются операторы тела конструктора, причём эти операторы могут изменить значения любых полей

3. Конструкторы и деструкторы

Деструкторы предназначены для удаления объектов. Наиболее важное его действие – освобождение всех ресурсов. Он определяется явно или неявно в каждом классе. Имя деструктора – это имя класса, перед которым помещён символ «тильда» (~)

```
~имя_класса(){ //деструктор не имеет арг-в
    операторы_деструктора
}
```

3. Конструкторы и деструкторы

Деструктор может вызываться явно, как и другие методы класса:
имя_объекта.~имя_класса();
имя_указателя_на_объект -> ~имя_класса();

При явном вызове деструктора объект не уничтожается.

3. Конструкторы и деструкторы

Когда программа покидает блок, в котором объект класса определён, деструктор вызывается без вмешательства программиста. Кроме того, деструктор неявно вызывается в тех случаях, когда объект, размещённый в динамической памяти, удаляется с помощью операции delete.

3. Конструкторы и деструкторы

Пример. Деструктор.

```
struct rectangle{
        double width;
        double height;
        rectangle(double w = 1.0, double h = 1.0)
        : width(w), height(h) {}
        ~rectangle() {cout << "\nDestructor";}
        void print() {cout << "\nwidth=" << width << " height=" << height;}
};
int main()
{
        { // начало блока
                rectangle R1(5, 3.5);
                R1.print();
                R1.~rectangle();
                R1.print();
        } // конец блока
        getchar(); return 0;
}
```

3. Конструкторы и деструкторы

Задание.

1. Для ранее созданного класса circle создать конструктор общего вида, конструктор копирования и конструктор умолчания, не принимающий аргументов, проверить работу всех этих конструкторов.
2. Создать деструктор для класса circle, выводящий сообщение и обнуляющий поле радиуса.

4. Статусы доступа

Доступность полей и методов извне можно контролировать с помощью статусов доступа. Существует три статуса доступа: public, private, protected.

public – открытые поля и методы
private – закрытые (собственные) поля и методы
protected – защищённые поля и методы

4. Статусы доступа

Открытые (public) поля и методы доступны для внешних обращений, закрытые (private) поля и методы доступны только для методов данного класса. При использовании классов без наследования, применение спецификатора protected эквивалентно использованию спецификатора private.

4. Статусы доступа

Забегая вперёд, класс-наследник имеет доступ к полям и методам родительского класса, имеющим статус public и protected, но не private.

4. Статусы доступа

За спецификатором доступа размещается двоеточие. Появление любого из указанных спецификаторов означает, что до конца определения класса, либо до другого спецификатора доступа все поля и методы имеют указанный статус.

4. Статусы доступа

Пример. Использование статусов доступа для полей и методов класса.

```
struct rectangle{
        private:
                double width; // закрытое поле данных
                double height; // закрытое поле данных
        public: // открытые методы
                void setW(double w){width = w;}
                void setH(double h){height = h;}
                void addW(double w){width+=w;}
                void addH(double h){height+=h;}
                double getW(){return width;}
                double getH(){return height;}
                rectangle(double w = 1.0, double h = 1.0)
                : width(w), height(h) {}
                void print()
                {cout << "\nwidth=" << width << " height=" << height;}
};
int main()
{
        rectangle R1(5, 3.5);
        cout << "width = " << R1.getW();
        R1.setW(5.5);
        R1.print();
        R1.addH(5);
        R1.print();
        getchar();
        return 0;
}
```

4. Статусы доступа

В случае использования в качестве ключа класса ключевых слов struct и union все поля и методы по умолчанию являются открытыми (publuc). В случае использования в качестве ключа класса слова class все поля и методы по умолчанию являются закрытыми (private).

4. Статусы доступа

Ещё один пример. Использование статусов доступа для полей и методов класса.

```
class length{
double meters; // Обратите внимание: это поле имеет статус private!
public:
        length(double m) : meters(m) {}
        double getMeters() {return meters;}
        double getCentimeters() {return meters*100;}
        double getInches() {return getCentimeters()/2.54;}
        double getFts() {return meters*0.3048;}
};
int main()
{
        #define N 5
        length L1(N);
        cout << N << " meters = " << L1.getCentimeters() << " centimeters\n";
        cout << N << " meters = " << L1.getInches() << " inches\n";
        getchar();
        return 0;
}
```

5. Внешнее определение методов

Метод должен быть определён, либо описан в теле (спецификации) класса. В отличие от обычных функций, метод имеет доступ ко всем полям данных и методам класса.

Если метод определён внутри спецификации класса, то этот метод по умолчанию считается подставляемым (inline). Если используется внешнее определение методов, метод не считается подставляемым.

5. Внешнее определение методов

При внешнем определении метода в спецификацию класса помещается прототип:
тип имя_метода(аргументы);

Вне тела класса метод определяется следующим образом:
тип имя_класса::имя_метода(аргументы)
{тело метода}

5. Внешнее определение методов

Пример. Внешнее определение методов.

```
struct rectangle{
private:
        double width; // закрытое поле данных
        double height; // закрытое поле данных
public: // открытые методы
        void setW(double w){width = w;}
        void setH(double h){height = h;}
        void addW(double w){width+=w;}
        void addH(double h){height+=h;}
        double getW(){return width;}
        double getH(){return height;}
        rectangle(double w = 1.0, double h = 1.0)
        : width(w), height(h) {}
        void print(); // Описание метода (прототип)
};

void rectangle::print() // Определение метода
{cout << "\nwidth=" << width << " height=" << height;}
```

5. Друзья классов

Дружественная функция – функция, которая не является методом класса, но имеет доступ к его защищённым и закрытым полям данных и методам. Для получения таких прав функция должна быть описана в теле спецификации класса со спецификатором friend.

5. Друзья классов

Дружественная функция – функция, которая не является методом класса, но имеет доступ к его защищённым и закрытым полям данных и методам. Для получения таких прав функция должна быть описана в теле спецификации класса со спецификатором friend.

5. Друзья классов

Пример использования дружественной функции.

```
struct rectangle{
private:
        double width;
        double height;
public:
        rectangle(double w = 1.0, double h = 1.0)
        : width(w), height(h) {}
        void print() {cout << "\nwidth=" << width << " height=" << height;}
        friend void changeRect(rectangle *, double, double);
};

void changeRect(rectangle *r, double w, double h){
r->width = w; r->height = h;
}

int main()
{
rectangle R1(5, 3.5);
changeRect(&R1, 4.4, 3.3);
R1.print();
return 0;
}
```

5. Друзья классов

Особенности дружественных функций:

- Объекты классов должны передаваться в дружественную функцию явно
- При вызове дружественной функции недопустимы операции выбора (например, имя_класса.имя_функции), т.к. дружественная функция не является методом класса
- На дружественную функцию не распространяется действие спецификаторов доступа (private, protected, public)

5. Друзья классов

Особенности дружественных функций (продолжение):

• дружественная функция может быть не только глобальной функцией, но и методом другого класса:

```
class CLASS {… char f2(…); …};
Class CL {
…
friend char CLASS::f2(…);
…};
```

Здесь метод f2() класса CLASS получает доступ ко всем полям и методам класса CL

5. Друзья классов

Особенности дружественных функций (продолжение):

- дружественная функция может быть дружественной по отношению к нескольким классам

```
class CL2; // описание класса
class CL1 {friend void f(CL1, CL2); …};
class CL2 {friend void f(CL1, CL2); …};
void f(…) {…}
```

5. Друзья классов


```
class point;
class line{
  double x1, x2;
public:
  line(double xx1, double xx2)
    :x1(xx1), x2(xx2) {}
  friend bool isInLine(line *, point *);
};
class point{
  double x;
public:
  point(double xx)
    :x(xx) {}
  friend bool isInLine(line *, point *);
};
```

```
bool isInLine(line * L, point * P){
  if (L->x1 < L->x2 &&
     L->x1 < P->x &&
     P->x < L->x2)
     return true;
  else
     if (L->x1 > L->x2 &&
        L->x1 > P->x &&
        P->x > L->x2)
        return true;
     else
        return false;
}
```

5. Друзья классов

Особенности дружественных функций (продолжение):

- класс может быть дружественным другому классу

```
class X2 {friend class X1; …};
class X1 {
  // определение дружественного класса
  // здесь доступны все поля и методы X2
}
```

5. Друзья классов

Задания.

1. Создать два класса: 1) окружность, 2) точка на плоскости, определить дружественную функцию вхождения точки на плоскости в окружность.
2. Добавить класс «квадрат», создать для класса «окружность» и класса «квадрат» дружественные функции определения того, может ли окружность быть вписанной или описанной для квадрата. (https://ru.wikipedia.org/wiki/Вписанная_окружность)

6. Указатель this

Когда метод класса вызывается для обработки данных конкретного объекта, этому методу неявно передаётся указатель на обрабатываемый объект. Этот указатель имеет фиксированное имя this.

Т.е. к полю данных F некоторого класса внутри методов класса можно обратиться следующим образом: this->F.

6. Указатель this

```
class member{
  static member *last_memb;
  member *prev, *next;
  char letter;
public:
  member(char cc) {letter = cc;}
  void add(void){
    if(last_memb==0) this->prev=0;
    else {last_memb -> next = this; this -> prev =
last_memb;}
    last_memb = this;
    this -> next = 0;
  }
  static void print(){
    member * uk; // вспомогательный указатель
    uk = last_memb;
    if(uk==0) cout << "Empty list!";
    else
    while(uk!=0){
      cout << uk -> letter << '\t';
      uk = uk -> prev;
    }
  cout << '\n';
  }};
```

```
member *member::last_memb = 0;
int main()
{
member A('a'); member B('b');
member C('c'); member D('d');
member::print();
A.add(); B.add(); C.add(); D.add();
member::print();
return 0;
}
```