Язык C++

C++ позволяет справиться с возрастающей сложностью программ (в отличие от C).

Автор – Бьёрн Страуструп.

Стандарты (комитета по стандартизации ANSI) – 1998, 2011.

C++:

- лучше C,
- поддерживает абстракции данных,
- поддерживает объектно-ориентированное программирование (ООП).

Парадигмы программирования

Все программы состоят из кода и данных и каким-либо образом концептуально организованы вокруг своего кода и\или данных.

Основные парадигмы (технологии) программирования определяют способ построения программ:

- **процедурно-ориентированная** (при кот. программа – это ряд последовательно выполняемых операций, причём код воздействует на данные, например в программах на С),
- **объектно-ориентированная** (при кот. программа состоит из объектов – программных сущностей, объединяющих в себе код и данные, взаимодействующих друг с другом через определенные интерфейсы, при этом доступ к коду и данным объекта осуществляется только через сам объект, т.е. данные определяют выполняемый код),
- функциональная,
- логическая.

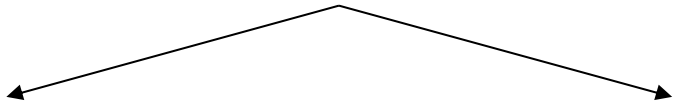
Постулаты ООП.

Абстракция - центральное понятие ООП.

Абстракция позволяет программисту справиться со сложностями решаемых им задач.

Мощный способ создания абстракций –

иерархическая классификация



типовая (**структура классов**)

структурная (**структура объектов**)

Основные механизмы (постулаты) ООП:

- **инкапсуляция**,
- **наследование**,
- **полиморфизм**.

ИНКАПСУЛЯЦИЯ

Инкапсуляция - механизм,

- связывающий вместе код и данные, которыми он манипулирует;

- защищающий их от произвольного доступа со стороны другого кода, внешнего по отношению к рассматриваемому.

Доступ к коду и данным жестко контролируется интерфейсом.

Основой инкапсуляции является **класс**.

Класс - это механизм (пользовательский тип данных) для создания объектов.

Объект класса - переменная типа класс или экземпляр класса.

Любой объект характеризуется **состоянием** (значениями полей данных) и **поведением** (операциями над объектами, задаваемыми определенными в классе функциями, которые называют **методами** класса).

НАСЛЕДОВАНИЕ

Наследование - механизм, с помощью которого один объект (**производного класса**) приобретает свойства другого объекта (**родительского, базового класса**).

Наследование позволяет объекту производного класса наследовать от своего родителя общие атрибуты, а для себя определять только те характеристики, которые делают его уникальным внутри класса.

Производный класс конкретизирует, в общем случае **расширяет** базовый класс.

Наследование поддерживает концепцию иерархической классификации.

Новый класс не обязательно описывать, начиная с нуля, что существенно упрощает работу программиста.

ПОЛИМОРФИЗМ

Полиморфизм - механизм, позволяющий использовать один и тот же интерфейс для общего класса действий.

В общем случае концепция полиморфизма выражается с помощью фразы "один интерфейс - много методов".

Выбор конкретного действия (метода) применительно к конкретной ситуации возлагается на компилятор. Программисту же достаточно запомнить и применять один интерфейс, вместо нескольких, что также упрощает его работу.

Различаются следующие виды полиморфизма:

- **статический** (на этапе компиляции, с помощью перегрузки функций),
- **динамический** (во время выполнения программы, реализуется с помощью виртуальных функций) и
- **параметрический** (на этапе компиляции, с использованием механизма шаблонов).

Декомпозиция задачи

При программировании в объектно-ориентированном стиле на первое место выходит **проектирование** решения задачи, т.е определение того, какие классы и объекты будут использоваться в программе, каковы их свойства и способы взаимодействия.

Как правило, при этом необходимо произвести декомпозицию задачи.

Декомпозиция – научный метод, использующий структуру задачи и позволяющий разбить решение одной большой задачи на решения серии меньших задач, возможно взаимосвязанных, но более простых.

Синтаксис класса

class имя_класса {
[private:]
закрытые члены класса (функции и поля-данные)
public:
открытые члены класса (функции и поля-данные)
protected:
защищенные члены класса
} список_объектов;

Описание объектов – экземпляров класса:

имя_класса список объектов;
// служ. слово class не требуется

Классы C++ отличаются от структур C++ **только** правилами определения **по умолчанию**

- **прав доступа** к первой области доступа членов класса и
- **типа наследования**:

для **структур** – **public**,
для **классов** – **private**.

Действия над объектами классов

Над объектами класса можно производить следующие действия:

- присваивать объекты одного и того же класса (при этом производится почленное копирование членов данных),
- получать адрес объекта с помощью операции &,
- передавать объект в качестве формального параметра в функцию,
- возвращать объект в качестве результата работы функции.
- осуществлять доступ к элементам объекта с помощью операции ‘.’, а если используется указатель на объект, то с помощью операции ‘->’.
- вызывать методы класса, определяющие поведение объекта.

Пример класса

```
class A {
    int a;
public:
    void set_a (int n);
    int get_a ( ) const { return a; }   // Константные методы класса
                                        // не изменяют состояние своего объекта
};

void A::set_a (int n) {
            a = n;
}

int main () {
            A obj1, obj2;
            obj1.set_a(5);
            obj2.set_a(10);
            cout << obj1.get_a ( ) << '\n';
            cout << obj2.get_a ( ) << endl;
            return 0;
}
```

АТД (абстрактный тип данных)

АТД называют тип данных с полностью скрытой (инкапсулированной) структурой, а работа с переменными такого типа происходит только через специальные, предназначенные для этого функции.

В C++ АТД реализуется с помощью классов (структур), в которых нет открытых членов-данных.

Класс A из предыдущего примера является абстрактным типом данных.

О терминологии

Оператор (statement) – действие, задаваемое некоторой конструкцией языка.

Операция (operator, для обозначения операций языка: +, *, =, и др.) – используются в выражениях.

Определение переменной (definition) - при этом отводится память, производится инициализация, определение возможно только 1 раз.

Объявление переменной (declaration) - дает информацию компилятору о том, что эта переменная где-то в программе описана.

Для преобразования типов используются два термина – **преобразование** (conversion) и **приведение** (cast).

Некоторые отличия C++ от C

- Введен логический тип ***bool*** и константы логического типа ***true*** и ***false*.**

- В C++ отсутствуют типы по умолчанию (например, обязательно int main () {…}).

- Локальные переменные можно описывать в любом месте программы, в частности внутри цикла for. Главное, чтобы они были описаны до их первого использования.

  По стандарту C++ переменная, описанная внутри цикла for, локализуется в теле этого цикла.

- В C++ переработана стандартная библиотека.

  В частности, в стандартной библиотеке C++ файл заголовков ввода/вывода назвается **<iostream>,** введены классы, соответствующие стандартным (консольным) потокам ввода – класс **istream** – и вывода – класс **ostream**, а также объекты **cin** (класса istream) и **cout** и **cerr** (класса ostream).

  Через эти объекты доступны операции ввода **>>** из стандартного потока ввода (например, cin >> x ;), и вывода **<<** в стандартный поток вывода (например, cout << ”string” << S << ‘\n’;), при использовании которых не надо указывать никакие форматирующие элементы.

Ссылки

```
int i = 5;
int & yeti = i;        // ссылка обязательно должна быть
                       // инициализирована
i = yeti + 1;
yeti = i + 1;
cout  <<  i  << y eti;          //напечатается  7 7
```

```
void  swap (int & x, int & y) {      }
        int t = x;
        x = y;
        y = t;
}
```

Обращение к функции swap:

```
        int a = 5, b = 6;
        swap (a, b);
```

Значения параметров функции по умолчанию

Пример:

```
void f (int a,  int b = 0, int c =1);
```

Обращения к функции:

```
f(3)          // a = 3, b = 0, c = 1;
f(3, 4)       // a = 3, b = 4, c = 1;
f(3, 4, 5)    // a = 3, b = 4, c = 5.
```

Работа с динамической памятью

*int *p,*m;*

p = new int ; или

p = new int (1); или

m = new int [10]; - для массива из 10 элементов;

массивы, создаваемые в динамической памяти инициализировать нельзя;

…….

delete p; или

delete [ ] m; - для удаления всего массива;

Пространства имен

```
namespace  имя  {
    // объявления
}

#include <iostream>
using namespace std;
```

Указатель this

Иногда для реализации того или иного метода возникает необходимость иметь указатель на «свой» объект, от имени которого производится вызов данного метода.

В C++ введено ключевое слово **this**, обозначающее «указатель на себя», которое можно трактовать как неявное поле данных любого класса:

const <имя класса> * this;

*this – сам объект.

Если this участвует в описании функции, перегружающей **операцию**, то он всегда указывает на **самый левый** (в записи вызова) операнд метода.

В реальности поле this не существует (не расходуется память) и при сборке программы подменяется на соответствующий адрес объекта.

Специальные методы класса

```
class A {
      ......
      public:
                  A ( );
[explicit]    A (int x);              // explicit запрещает неявное
                                      // преобразование int  в A
                  A (A & y);          // A (const A & y);
                  A (int x, int y);
                // A (int x = 0, int y = 0);      // заменяет 1-ый, 2-ой и 4-ый
                                                  // конструкторы
                ~A ();
                ......
};
Int main () {
      A a1,  a2 (10), a3 = a2;
      A a4 = 5,  a5 = A(7), *a6 = new A (1);
```

Класс Box

```
class Box {
    int  l;                    // length – длина
    int w;            // width – ширина
    int h;                     // height – высота
public:
    int volume () { return   l * w * h ; }
    Box (int a, int b, int c ) { l = a; w = b; h = c; }
    Box (int s) { l = w = h = s; }
    Box ( ) { w = h = 1; l = 2; }
    int get_l ( ) { return l; }
    int get_w ( ) { return w; }
    int get_h ( ) { return h; }
};
```

Автоматически сгенерированные конструктор копирования и операция присваивания:

```
Box (const Box & a) {  l = a.l;     w = a.w;              h = a.h; }
```

```
Box &  operator = ( const Box & a) { l = a.l;    w = a.w;   h = a.h;   return * this; }
```

Конструктор копирование и операцию присваивания можно переопределить.

Неплоский класс string

```
class string {
      char * p;      // здесь потребуется динамическая память,
      int size;
public:
      string (const  char * str);
      string (const string & a);
      ~string ( ) { delete [ ] p; }
       string & operator= (const string & a);
     ...
};

string :: string (const char * str) {
     p = new char [ ( size = strlen (str) ) + 1];
     strcpy (p, str);
}

string :: string (const string & a) {
     p = new char [ (size = a.size) + 1];
     strcpy (p, a.p);
}
```

Переопределение операции присваивания

```
string & string :: operator = (const string & a) {

   if  (this == & a) return * this;        //  если  a = a
   delete [ ] p;
   p = new char [ (size = a.size) + 1];
   strcpy (p, a.p);
   return * this;
}
```

При этом: s1 = s2 ~   s1.operator= (s2);

Пример использования класса string

```
void f {
        string  s1 (“Alice”);

        string  s2 = s1;

        string  s3 (“Kate”);
        ...
        s3 = s1;
}
```

{... s1...s2 {...s3...}...s1...s2}

Композиция (строгая агрегация) объектов

```
class Point {
        int x;
        int y;
public:
        Point ( );
        Point ( int, int );
        ...
};
```

```
class Z {
        Point p;
        int z;
public:
        Z ( int  c ) { z = c; };
        ...
};
```

```
Z * z = new Z (1);          // Point ( );  Z(1);
delete z;                   // ~Z();   ~Point();
```

Использование **списка инициализации** при **описании** конструктора:

```
Z :: Z ( int c ) :  p (1, 2)  { z = c; }  или
Z :: Z ( int c ) :  p (1, 2),  z (c)  { }
```

Порядок вызова конструкторов и деструкторов

При вызове **конструктора** класса выполняются:

1. конструкторы базовых классов (если есть наследование),
2. конструкторы умолчания всех вложенных объектов в порядке их описания в классе,
3. собственный конструктор (при его вызове все поля класса уже проинициализированы, следовательно, их можно использовать).

Деструкторы выполняются в обратном порядке:

1. собственный деструктор (при этом поля класса ещё не очищены, следовательно, доступны для использования),
2. автоматически вызываются деструкторы для всех вложенных объектов в порядке, обратном порядку их описания в классе,
3. деструкторы базовых классов (если есть наследование).

Вызов конструктора копирования

1. явно,

2. в случае:

```
Box a (1, 2, 3);
Box b = a;  // a – параметр конструктора копирования,
```

3. в случае:

```
Box c = Box (3, 4, 5);
// сначала создается временный объект и вызывается
// обычный конструктор, а затем работает конструктор
// копирования при создании объекта c; если компилятор
// оптимизирующий,  вызывается только обычный
// конструктор с указанными параметрами;
```

4. при передаче параметров функции по значению (при создании локального объекта);

5. при возвращении результата работы функции в виде объекта.

Вызов других конструкторов

1. явно,
2. при создании объекта (при обработке описания объекта),
3. при создании объекта в динамической памяти (по new), при этом сначала в «куче» отводится необходимая память, а затем работает соответствующий конструктор,
4. при композиции объектов наряду с собственным конструктором вызывается конструктор объекта – члена класса,
5. при создании объекта производного класса также вызывается конструктор и базового класса,
6. при автоматическом приведении типа с помощью конструктора преобразования.

Вызов деструктора

1. явно,

2. при свертке стека - при выходе из блока описания объекта, в частности при обработке исключений, завершении работы функции;

3. при уничтожении временных объектов - сразу, как только завершается конструкция, в которой они использовались;

4. при выполнении операции delete для указателя на объект (инициализация указателя - с помощью операции new), при этом сначала работает деструктор, а затем освобождается память.

5. при завершении работы программы при удалении глобальных/статических объектов.

Конструкторы вызываются в порядке определения объектов в блоке. При выходе из блока для всех автоматических объектов вызываются деструкторы, в порядке, противоположном порядку выполнения конструкторов.

Друзья класса

Друг класса – это функция, не являющаяся членом этого класса, но имеющая доступ к его private и protected членам.

Своих друзей класс объявляет сам в любой зоне описания класса с помощью служебного слова **friend**.

Функция-друг может быть описана внутри класса.

Если функций, имена которых совпадают с объявленной в классе функцией-другом, несколько, то другом считается только та, у которой в точности совпадает прототип.

Другом класса может быть:

- обычная функция: `friend void f (...);`
- функция-член другого класса: `friend void Y::f (..);`
- весь класс: `friend class Y;`

Использование функций - друзей класса

```
class X {
        int a;
        friend void fff (const X*, int); // здесь нет this !
    public:
        void mmm (int);
};

void fff (const X* p, int i) {
        p -> a = i;
}

void X::mmm (int i) {
        a = i;
}

void f () {
        X obj;
        fff (&obj, 10);
        obj.mmm (10);
}
```

Свойства друзей класса

Дружба не обладает ни наследуемостью, ни транзитивностью.

Примеры:

```
class A {
    friend class B;
    int a;
};
```

```
class B {
        friend class C;
};
```

```
class C {
    void f (A* p) {
        p -> a++;  // ошибка, нет доступа к закрытым членам класса A
    }
};
```

```
class D: public B {
    void f (A* p) {
        p -> a++; // ошибка, нет доступа к закрытым членам класса A

    }
};
```

Преимущества использования друзей класса

1. Эффективность реализации (можно обходить ограничения доступа, предназначенные для обычных пользователей).

2. Функция-друг нескольких классов позволяет упростить интерфейс этих классов.

3. Функция-друг допускает преобразование своего первого параметра-объекта, а метод класса - нет.

Перегрузка операций

- Для перегрузки встроенных операций C++ используется ключевое слово **operator**.

- Перегружать операцию можно с помощью
  - функции-члена,
  - функции-друга,
  - обычной функции (что менее эффективно).

- Нельзя перегружать:

  ‘.’ , ‘::’ , ‘?:’ , ‘.*’ , sizeof, и typeid !!!

Пример 1.

```
class complex {
        double re, im;
    public:
        complex (double r = 0, double i = 0)  {     re = r;
                                                    im = i;
        }
        complex operator+ (const complex & a) {
                complex temp (re + a.re, im + a.im);
                return temp;
        }
        ... //(*) operator double () { return re; } – функция преобразования
};
int main () {
        complex x (1, 2), y (5, 8), z;
        double t = 7.5;
        z = x + y;  // O.K. – x.operator+ (y);
        z = z + t;  // O.K. – z.operator+ (complex (t));     если есть  (*), то
                    // неоднозначность: '+' -  double  или перегруженный
        z = t + x;  // Er.! – т.к. первый операнд по умолчанию – типа
    complex.
}
```

Пример 2.

```
class complex {
        double re, im;
    public:
        complex (double r = 0, double i = 0) {
                re = r;
                m = i;
        }
        friend complex operator+ (const complex & a, const complex & b);
        ...
};
complex operator+ (const complex & a, const complex & b) {
                complex temp (a.re + b.re, a.im + b.im);
                return temp;
}
int main () {
        complex x (1, 2), y (5, 8), z;
        double t = 7.5;
        z = x + y;  // O.K. – operator+ (x, y);
        z = z + t;  // O.K. –  operator+ (z, complex (t));
        z = t + x;  //O.K.  – operator+ (complex (t), x);
}
```

Пример 3.

```
class complex {
        double re, im;
    public:
        friend complex operator * (const complex & a,  double b);
        ...
};
complex operator * (const complex & a, double b) {
                    complex temp (a.re * b,  a.im * b);
                    return temp;
}
int main () {
    complex x (1, 2),  z;
    double t = 7.5;
    z = x * t;      // O.K. – x.operator* (t);
    z = t * x;      // Er.! т.к. нет функции преобразования x --> double, но
                    // если бы была, была бы неоднозначность:
                    // * - из double или из complex
}
```

В таких случаях обычно определяют еще одного друга с профилем:

```
        complex operator * (double b,  const complex & a);
```

Замечания

- n-местные операции перегружаются

  a) методом с (n-1) параметром,

  b) функцией-другом или обычной функцией с n параметрами;

- в любом случае сохраняется приоритет, ассоциативность и местность операций;

- операции

  = , [ ] , ( ) и ->

  можно перегрузить **только** нестатическими методами класса, что гарантирует, что первым операндом будет сам объект, к которому операция применяется;

Особенности перегрузки операций ++ и --

```
complex x;
```

префиксная ++: ++ x; ~ x.operator ++ ();

```
complex & operator ++ () {
    re = re + 1;
    im = im + 1;
    return *this;
}
```

постфиксная ++: x ++; ~ x.operator ++ (0);

```
complex operator ++ (int) {
    complex c = * this;
    re = re + 1;
    im = im + 1;
    return  c;
}
```

Пример перегрузки операции «( )» и операции вывода «<<»

```
class Matrix {
        double M [ 3 ] [ 3 ];
public:
        Matrix ();
        double & operator ( ) (int i, int j) {
                return M [ i ] [ j ];
        }
     friend ostream & operator << (ostream & s, const Matrix & a) {
                for (int i = 0;  i < 3 ;  i ++) {
                        for (int j = 0; j < 3; j ++)
                                s  <<  a (i,  j)  <<  ' ';
                        s  <<  endl;
                }
                return s;
    }
};
```