Объектно-ориентированное программирование

Практическое занятие №2.

Классы в языке C++

Автор: И.О. Архипов, к.т.н., доцент

2.1. Присваивание объектов

```
// Пример присваивания объекта
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
void set(int i, int j)   {a = i; b = j;}
void show()   {cout << a << ‘ ‘ << b << "\n";}
};
int main () {
myclass obl, ob2;
obl.set(10, 4);
ob2 = obl;
obl.show();
ob2.show();
return 0;
}
```

2.1. Присваивание объектов

Рассмотрим несколько измененный класс `strtype`

```
// Эта программа содержит ошибку!!!
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class strtype {
char *p;
int len;
public:
strtype(char *ptr);
~strtype();
void show();
};
```

2.1. Присваивание объектов

```
strtype::strtype(char *ptr) {
len=strlen(ptr);
p=(char *) malloc(len+1);
if(!p) {
   cout << "Ошибка выделения памяти\n";
exit(1);
}
strcpy(p, ptr);
}
strtype::~strtype() {
cout << "Освобождение памяти по адресу ";
cout << (unsigned)p << "\n";
free(p);
}
```

2.1. Присваивание объектов

```
void strtype::show() {
cout << p << " – длина:  " << len;
cout << "\n";
}
int main() {
strtype s1("Это проверка");
strtype s2("Мне нравится C++");
s1.show();
s2.show();
// в данном случае присваивание ведет к ошибке
s2 = s1;
s1.show();
s2.show();
return 0;
}
```

2.1. Присваивание объектов

Результат работы программы:

```
Это проверка – длина:  12
Мне нравится C++ – длина:  16
Это проверка – длина:  12
Это проверка – длина:  12
Освобождение памяти по адресу 11553160
Освобождение памяти по адресу 11553160
```

При присваивании одного объекта другому необходимо убедиться в том, что не удаляете нужную информацию, которая может понадобиться в дальнейшем.

2.2. Передача объектов функциям

```
#include <iostream>
using namespace std;
class samp {
int i;
public:
samp(int n)   {i = n;}
int get_i()   {return i;}
void set_i(int t)  {i=t;}
};

int sqr_it1(samp ob){
return ob.get_i()*ob.get_i();
}

void sqr_it2(samp *ob){
set_i(ob->get_i()*ob->get_i());
}
```

2.2. Передача объектов функциям

```
int main() {
samp a(10), b(2);
cout << sqr_it1(a) << "\n";
cout << sqr_it1(b) << "\n";
sqr_it2(&a);
cout << a.get_i() << "\n";
return 0;
}
```

Функция `sqr_it()` получает аргумент типа `samp`.

Функция `sqr_it2()` получает адрес объекта типа `samp`.

2.2. Передача объектов функциям

При создании копии объекта, когда он используется в качестве аргумента функции, **конструктор копии не вызывается**.

Однако, когда копия удаляется, **вызывается ее деструктор**.

```
#include <iostream>
using namespace std;
class samp {
int i ;
public:
samp(int n)  {i = n;
cout << "Работа конструктора\n";}
~samp()      {cout << "Работа деструктора\n";}
int get_i()  {return i;}
};
int sqr_it(samp ob){
return ob.get_i() * ob.get_i();
}
```

2.2. Передача объектов функциям

```
int main() {
samp a (10);
cout << sqr_it(a) << "\n";
return 0;
}
```

Программа выводит следующее:

```
Работа конструктора
Работа деструктора
100
Работа деструктора
```

2.3. Объекты в качестве возвращаемого значения функций

```
// Возвращение объекта из функции
#include <iostream>
#include <cstring>
using namespace std;

class samp {
char s[80];
public:
void show() {cout << s << "\n";}
void set(char *str) {strcpy(s, str);}
};
```

2.3. Объекты в качестве возвращаемого значения функций

```
samp input(){
char s[80];
samp str;
cout << "Введите строку:  ";
cin >> s;
str.set(s);
return str;
}

int main() {
samp ob;
ob = input();
ob.show();
return 0;
}
```

2.3. Объекты в качестве возвращаемого значения функций

Если функция возвращает объект, то для хранения возвращаемого значения создается временный объект. Удаление временного объекта сопровождается вызовом его деструктора.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class samp {
char *s;
public:
samp()   {s = '\0';}
~samp()  {if(s) free(s);
cout << "Освобождение памяти по адресу s\n";}
void show()   {cout << s << "\n";}
void set(char *str);
};
```

2.3. Объекты в качестве возвращаемого значения функций

```
void samp::set(char *str){
s =  (char *) malloc(strlen(str)+1);
if(!s){
   cout << "Ошибка выделения памяти\n";
   exit(1);
}
strcpy(s, str);
}
samp input(){
char s [80];
samp str;
cout << "Введите строку:  ";
cin >> s;
str.set(s);
return str;
}
```

2.3. Объекты в качестве возвращаемого значения функций

```
int main() {
samp ob;
ob = input();  // Это ведет к ошибке в ob.show();
return 0;
}
```

Результат работа программы

```
Введите строку:  QWERTY
Освобождение памяти по адресу s
Освобождение памяти по адресу s
Освобождение памяти по адресу s
```

2.4. Дружественные функции

Дружественные функции не являются членами класса, но тем не менее имеют доступ к его закрытым элементам.

```
// Пример использования дружественной функции
#include <iostream>
using namespace std;

class myclass {
int n, d;
public:
myclass(int i, int j) {n = i; d = j;}
friend int isfactor(myclass ob);
};

int isfactor(myclass ob) {
if(!(ob.n % ob.d)) return 1;
   else return 0;
}
```

2.4. Дружественные функции

```
int main() {
myclass obl(10, 2), ob2(13, 3);
if(isfactor(obl)
       cout << "10 без остатка делится на 2\n";
   else cout << "10 без остатка не делится на 2\n";

if(isfactor(ob2)
       cout << "13 без остатка делится на 3\n";
   else cout << "13 без остатка не делится на 3\n";

return 0;
}
```

2.4. Дружественные функции

1. Дружественная функция не является членом класса, для которого она дружественна.

```
obl.isfactor();   // неправильно,
          // isfactor() – это не функция-член
```

2. Дружественная функция может иметь доступ к членам класса только через объект, который объявлен внутри функции или передан ей.

3. Дружественная функция не наследуется. Если в базовый класс включается дружественная функция, то эта дружественная функция не является таковой для производных классов.

4. Функция может быть дружественной более чем к одному классу.

2.4. Дружественные функции

Разработаем абстракции легковых и грузовых автомобилей и функцию, сравнивающую их скорости.

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

class truck;  // предварительное объявление

class car {
int passengers;
int speed;
public:
car(int p, int s) {passengers = p;  speed = s;}
friend int sp_greater(car c, truck t);
};
```

2.4. Дружественные функции

```
class truck {
int weight;
int speed;
public:
truck(int w, int s) {weight = w; speed = s;}
friend int sp_greater(car c, truck t);
};

int sp_greater(car c,  truck t) {
return c.speed – t.speed;
}
```

2.4. Дружественные функции

```
int main() {
int t;
car c1(6, 55), c2(2, 120);
truck t1(10000, 55), t2(20000, 72);
cout << "Сравнение значений c1 и t1:\n";

t = sp_greater(c1, t1);
if(t<0)  cout << "Грузовик быстрее, \n";
   else if(t==0) cout << "Скорости одинаковы, \n";
    else cout << "Легковая машина быстрее, \n";
cout << "\nСравнение значений c2 и t2:\n";

t = sp_greater(c2,  t2);
if(t<0) cout << "Грузовик быстрее, \n";
   else if(t==0) cout << "Скорости одинаковы, \n";
    else cout << "Легковая машина быстрее, \n";
return 0;
}
```

2.4. Дружественные функции

Функция может быть членом одного класса и дружественной другому.

Перепишем предыдущий пример так, чтобы функция `sp_greater()` являлась членом класса `car` и дружественной классу `truck`

```
#include <iostream>
using namespace std;
class truck;  // предварительное объявление

class car {
int passengers;
int speed;
public:
car(int p, int s) {passengers = p; speed = s;}
int sp_greater(truck t);
};
```

2.4. Дружественные функции

```
class truck {
int weight;
int speed;
public:
truck(int w, int s) {weight = w; speed = s;}

// оператор расширения области видимости
friend int car::sp_greater(truck t);
};

int car::sp_greater(truck t) {
return speed – t.speed;
}
```

2.4. Дружественные функции

```
int main() {
int t;
car c1(6, 55);
truck t1(10000, 55);
cout << "Сравнение значений c1 и t1:\n";

// вызывается как функция-член класса car
t = c1.sp_greater(t1);

if(t<0)  cout << "Грузовик быстрее,\n";
   else if(t==0) cout << "Скорости одинаковы,\n";
        else
       cout << "Легковая машина быстрее,\n";
return 0;
}
```

2.5. Массивы объектов

```
//Пример массива объектов:
#include <iostream>
using namespace std;
class samp {
int a;
public:
void set_a(int n) {a = n;}
int get_a() {return a;}
};
int main() {
samp ob[4];
int i;
for(i=0; i<4; i++) ob[i].set_a(i);
for(i=0; i<4; i++) cout << ob[i].get_a();
cout << "\n";
return 0;
}
```

2.5. Массивы объектов

Если класс содержит конструктор, массив объектов может быть инициализирован.

```
// Инициализация массива
#include <iostream>
using namespace std;

class samp {
int a;
public:
samp(int n) {a = n;}
int get_a() {return a;}
};
```

2.5. Массивы объектов

```
int main() {
samp ob[4] = {-1, -2, -3, -4};
int i;
for(i=0; i<4; i++)
   cout << ob[i].get_a() << ‘ ‘;
cout << "\n";
return 0;
}
```

Фактически синтаксис списка инициализации — это сокращение следующей конструкции:

```
samp ob[4] = {samp(-1), samp(-2), samp(-3), samp(-4)};
```

2.5. Массивы объектов

```
// Создание двумерного массива объектов
#include <iostream>
using namespace std;
class samp {
int a;
public:
samp(int n) {a = n;}
int get_a() {return a;}
};
```

2.5. Массивы объектов

```
int main() {
samp ob[4][2] = {1, 2, 3, 4, 5, 6, 7, 8};
int i;
for(i=0; i<4; i++){
   cout << ob[i][0].get_a() << ‘ ‘;
   cout << ob[i][1].get_a() << "\n";
}
cout << "\n";
return 0;
}
```

Эта программа выводит на экран следующее:

```
1 2
3 4
5 6
7 8
```

2.5. Массивы объектов

```
//Инициализация массива если конструктор имеет
//более одного аргумента
#include <iostream>
using namespace std;
class samp {
int a, b;
public:
samp(int n, int m) {a = n; b = m;}
int get_a() {return a;}
int get_b() {return b;}
};
```

2.5. Массивы объектов

```
int main () {
samp ob[4][2] = {
   samp(l,2), samp(3,4),
   samp(5,6), samp(7,8),
   samp(9,10), samp(ll,12),
   samp(13,14), samp(15,16)
};
int i;
for(i=0; i<4; i++){
   cout << ob[i][0].get_a() << ‘ ‘;
   cout << ob[i][0].get_b() << "\n";
   cout << ob[i][1].get_a() << ‘ ‘;
   cout << ob[i][1].get_b() << "\n";
}
cout << "\n";
return 0;
}
```

2.6. Указатель this

Указатель `this` автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов.

Например, рассмотрим следующую инструкцию:

```
ob.f1();  // предположим, что ob – это объект
```

Функции `f1()` автоматически передается указатель на объект ob. Этот указатель и называется `this`.

Указатель `this` передается только функциям-членам. Дружественным функциям указатель `this` не передается.

2.6. Указатель this

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;
class inventory {
char item[20];
double cost;
int on_hand;
public:
inventory(char *i, double c, int k) {
   strcpy(item, i);
   cost = c;
   on_hand = k;
}
void show();
};
```

2.6. Указатель this

```
void inventory::show() {
cout << item;
cout << ":  $" << cost;
cout << "  On hand:  " << on_hand << "\n";
}
int main() {
inventory ob("wrench", 4.95, 4);
ob.show();
return 0;
}
```

2.6. Указатель this

```
// Демонстрация указателя this (вариант 2)
#include <iostream>
#include <cstring>
using namespace std;
class inventory {
char item[20];
double cost;
int on_hand;
public:
inventory(char *i, double c, int k) {
   strcpy(this->item, i);
   this->cost = c;
   this->on_hand = k;
}
void show();
};
```

2.6. Указатель this

```
void inventory::show() {
cout << this->item;
cout << ":  $" << this->cost;
cout << "  On hand:  " << this->on_hand << "\n";
}
int main() {
inventory ob("wrench", 4.95, 4);
ob.show();
return 0;
}
```

Внутри функции show() следующие две инструкции равнозначны:

```
cost = 123.23;
this->cost = 123.23;
```

2.7. Операторы new и delete

Обща форма записи операторов:

```
p_var = new type;
delete p_var;
```

`type` — спецификатор типа объекта, для которого необходимо выделить память;
`p_var` — указатель на этот тип.

Если свободной памяти недостаточно для выполнения запроса, произойдет одно из двух:

1. либо оператор new возвратит нулевой указатель,
2. либо будет сгенерирована исключительная ситуация.

2.7. Операторы new и delete

Преимущества операторов `new` и `delete` перед функциями `malloc()` и `free()`:

1. оператор `new` автоматически выделяет требуемое количество памяти для хранения объекта заданного типа (не нужно использовать `sizeof` для подсчета требуемого числа байтов);
2. оператор `new` автоматически возвращает указатель на заданный тип данных (не нужно выполнять приведение типов);
3. операторы `new` и `delete` можно перегружать, что дает возможность простой реализации собственной модели распределения памяти;
4. допускается инициализация объекта, для которого динамически выделена память;
5. нет необходимости включать в программы заголовок `<cstdlib>`.

2.7. Операторы new и delete

```
// Динамическое выделение памяти объектам
#include <iostream>
using namespace std;
class samp {
int i, j;
public:
void set_ij(int a, int b)   {i = a; j = b;}
int get_product() {return i*j;}
};
int main() {
samp *p;
p = new samp;  // выделение памяти объекту
if(!p) {
    cout << "Ошибка выделения памяти\n";
    return 1;
}
p->set_ij(4,5);
cout << "Итог равен:" << p->get_product() << "\n";    return
0;
}
```

2.7. Операторы new и delete

Форма инициализации динамически размещаемого объекта:

```
p_var = new type (начальное_значение);
```

Форма динамического размещения объекта:

```
p_var = new type [size];
```

Форма удаления динамически размещённого объекта:

```
delete [] p_var;
```

2.7. Операторы new и delete

```
#include <iostream>
using namespace std;
class samp {
int i, j;
public:
samp(int a, int b) {i = a; j = b;}
int get_product() {return i*j;}
};
int main() {
samp *p;
p = new samp(6,  5);
if(!p) {
   cout << "Ошибка выделения памяти\n";
   return 1;
}
cout << "Итог равен:" << p->get_product() << "\n";
delete p;
return 0;
}
```

2.7. Операторы new и delete

```
// Динамическое выделение памяти для массива объектов
#include <iostream>
using namespace std;
class samp {
int i,j;
public:
void set_ij (int a, int b) {i = a; j = b;}
int get_product() {return i*j;}
};
```

2.7. Операторы new и delete

```
int main () {
samp *p;
int i;
p = new samp [10];  // размещение массива объектов
if(!p) {
   cout << "Ошибка выделения памяти\n";
   return 1;
}
for(i=0; i<10; i++)
   p[i].set_ij(i, i);
for(i=0; i<10; i++){
   cout << "Содержимое  [" << i << "] равно:";
   cout << p[i].get_product() << "\n";
}
delete [] p;
return 0;
}
```

2.7. Операторы new и delete

Предыдущая программа выводит на экран следующее:

```
Содержимое   [0]равно:0
Содержимое   [1]равно:1
Содержимое   [2]равно:4
Содержимое   [3]равно:9
Содержимое   [4]равно:16
Содержимое   [5]равно:25
Содержимое   [6]равно:36
Содержимое   [7]равно:49
Содержимое   [8]равно:64
Содержимое   [9]равно:81
```

2.7. Операторы new и delete

```
   // Динамическое выделение памяти для массива объектов
#include <iostream>
   using namespace std;
   class samp {
   int i, j;
   public:
   void set_ij(int a, int b) {i = a; j = b;}
   ~samp() { cout << "Удаление объекта...\n";}
   int get_product() {return i*j;}
   };
```

2.7. Операторы new и delete

```
int main() {
samp *p;
int i ;
p = new samp [4];  // размещение массива объектов
if(!p){
   cout << "Ошибка выделения памяти\n";
   return 1;
}
for(i=0; i<4; i++)
p[i].set_ij(i, i);
for(i=0; i<4; i++){
   cout << "Содержимое  [" << i << "] равно:";
   cout << p[i].get_product() << "\n";
}
delete [] p;
return 0;
}
```

2.7. Операторы new и delete

Предыдущая программа выводит на экран следующее:

```
Содержимое [0] равно: 0
Содержимое [1] равно: 1
Содержимое [2] равно: 4
Содержимое [3] равно: 9
Удаление объекта...
Удаление объекта...
Удаление объекта...
Удаление объекта...
```

2.8. Ссылки

Ссылка – это скрытый указатель.

Ссылку допустимо использовать тремя способами:

1. ссылку можно передать в функцию;
2. ссылку можно возвратить из функции;
3. можно создать независимую ссылку.

2.8. Ссылки

Разработаем функцию, в которой параметром является указатель

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

void f(int *n);  // использование параметра-указателя

int main() {
int i = 0;
f(&i);
cout << "Новое значение i: " << i << 'n\';
return 0;
}

void f(int *n) {
*n = 100;   // занесение числа 100 в аргумент,
          // на который указывает указатель n
}
```

2.8. Ссылки

Автоматизируем процесс передачи параметра в функцию с помощью параметра–ссылки

```
#include <iostream>
using namespace std;
void f(int &n);  // объявление параметра-ссылки
int main() {
int i = 0;
f(i) ;
cout << "Новое значение i: " << i << 'n\';
return 0;
}
// Теперь в функции f() используется параметр-ссылка
void f(int &n){
n = 100;    // занесение числа 100 в аргумент,
      // используемый при вызове функции f()
}
```

2.8. Ссылки

Преимущества параметра–ссылки по сравнению с параметром–указателем:

1. с практической точки зрения нет необходимости получать и передавать в функцию адрес аргумента (При использовании параметра-ссылки адрес передается автоматически);
2. по мнению многих программистов, параметры-ссылки предлагают более понятный и элегантный интерфейс, чем неуклюжий механизм указателей;
3. при передаче объекта функции через ссылку копия объекта не создается (это уменьшает вероятность ошибок, связанных с построением копии аргумента и вызовом ее деструктора).

2.8. Ссылки

```
//Демонстрация ссылки
#include <iostream>
using namespace std;
void swapargs(int &x, int &y);
int main() {
int i, j;
i = 10; j = 19;
cout << "i: " << i << ", ";
cout << "j: " << j << "\n";
swapargs(i, j);
cout << "После перестановки:  ";
cout << "i: " << i << ", ";
cout << "j: " << j << "\n";
return 0;
}
void swapargs(int &x, int &y) {
int t;
t = x; x = y; y = t;
}
```

2.8. Ссылки

```
//Вариант функции swapargs() со ссылками
void swapargs(int &x, int &y){
int t;
t = x;
x = y;
y = t;
}

//Вариант функции swapargs() с указателями
void swapargs(int *x, int *y){
int t;
t = *x;
*x = *y;
*y = t;
}
```

2.9. Передача ссылки на объект

Особенности передачи объектов по ссылке:

1. При передаче объекта по ссылке копия объекта не создается, и при возвращении функцией своего значения деструктор не вызывается.
2. Изменения объекта внутри функции влияют на исходный объект, указанный в качестве аргумента функции.
3. Ссылка не указатель, хотя она и указывает на адрес объекта. При передаче объекта по ссылке для доступа к его членам используется оператор точка (`.`), а не стрелка (`->`).

2.9. Передача ссылки на объект

```
#include <iostream>
using namespace std;
class myclass {
int who;
public:
myclass(int n) {who = n;
      cout << "Работа конструктора “;
      cout << who << "\n";}
~myclass()   {cout << "Работа деструктора ";
            cout << who << "\n";}
int id  () {return who;}
};
void f(myclass &ob){
cout << "Получено" << ob.id() << "\n";
}
```

2.9. Передача ссылки на объект

```
int main () {
myclass x(1);
myclass y(2);
f(x);
f(y);
return 0;
}
```

```
Работа конструктора 1 Получено 1
Работа конструктора 2 Получено 2
Работа деструктора 1
Работа деструктора 2
```

2.10. Ссылка в качестве возвращаемого значения функции

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

int &f();
int x;

int main() {
f() = 100; // присваивание возвращаемой ссылке
cout << x << "\n";
return 0;
}

// Возвращение ссылки на целое
int &f(){
return x;  // возвращает ссылку на x
}
```

2.10. Ссылка в качестве возвращаемого значения функции

Следует быть внимательным при возвращении ссылок, чтобы объект, на который вы ссылаетесь, не вышел из области видимости.

Рассмотрим изменённую функцию `f()`:

```
// Возвращение ссылки на целое
int &f(){
int x;       // x – локальная переменная
return x;    // возвращение ссылки на x
}
```

Переменная x становится локальной переменной функции `f()` и выходит из области видимости после выполнения функции.

Ссылку, возвращаемую функцией `f()`, уже нельзя использовать.

2.10. Ссылка в качестве возвращаемого значения функции

```
// Пример защищенного массива
#include <iostream>
#include <cstdlib>
using namespace std;
class array {
int size;
char *p;
public:
array(int num);
~array() {delete [] p;}
char &put(int i);
char get(int i);
};
```

2.10. Ссылка в качестве возвращаемого значения функции

```
array::array(int num) {
p = new char[num];
if(!p) {
   cout << "Ошибка выделения памяти\n";
   exit(1);
}
size = num;
}
// Заполнение массива
char &array::put(int i){
if(i<0 || i>=size){
   cout << «Нарушены границы массива!!!\n";
   exit(1);
}
return p[i];  // возврат ссылки на p[i]
}
```

2.10. Ссылка в качестве возвращаемого значения функции

```
char array::get(int i){ // Получение элемента массива
if(i<0 || i>=size)   {
   cout << “Нарушены границы массива!!!\n”;
   exit(1);
}
return p[i];  // возврат символа
}
int main() {
array a(10);
a.put(3) = 'X';
a.put(2) = 'R';
cout << a.get(3) << a.get(2) << "\n";
// Нарушим границы массива
a.put(11) = '!';
return 0;
}
```

2.11. Независимые ссылки и ограничения на применение ссылок

Независимая ссылка — это ссылка, которая является другим именем переменной.

Независимая ссылка должна быть инициализирована при объявлении.

Ограничения на применение ссылок всех типов:

1. Нельзя ссылаться на другую ссылку;
2. Нельзя получить адрес ссылки;
3. Нельзя создавать массивы ссылок и ссылаться на битовое поле;
4. Ссылка должна быть инициализирована до того, как стать членом класса, возвратить значение функции или стать параметром функции;

2.11. Независимые ссылки и ограничения на применение ссылок

```
// Независимая ссылка ref служит другим именем
// переменной x
#include <iostream>
using namespace std;
int main() {
int x;
int &ref = x;  // создание независимой ссылки
x = 10;
ref = 10;
ref = 100;
cout << x << ' ' << ref << "\n";
return 0;
}
```