**Ссылки**

Ссылку (reference) можно считать скрытым указателем и можно употреблять как еще одно имя переменной.

Ссылку допустимо использовать тремя способами:

1. ссылку можно передавать в функцию;
2. можно возвращать из функции;
3. можно создавать независимую ссылку.

Для объявления параметра-ссылки перед именем переменной ставится знак амперсанда (&). При вызове функции перед аргументом не надо ставить знак & (ей автоматически передается адрес аргумента).

**Пример**: */\* с указателями и ссылкой \*/*

#include<iostream>

using namespace std;

void swap(int, int);

void swap1(int\*, int\*);

void swap2(int&, int&);

int main()

{

int a = 10, b = 20;

cout << "a=" << a << " b=" << b << endl;

swap(a, b);

cout << "a=" << a << " b=" << b << endl;

a = 10, b = 20;

swap1(&a, &b);

cout << "a1=" << a << " b1=" << b << endl;

a = 10, b = 20;

swap2(a, b);

cout << "a2=" << a << " b2=" << b << endl;

system("pause");

return 0;

}

void swap(int c, int d)

{

int t = c;

c = d;

d = t;

cout << "c=" << c << " d=" << d << endl;

}

void swap1(int \*c, int \*d)

{

int t = \*c;

\*c = \*d;

\*d = t;

cout << "c1=" << \*c << " d1=" << \*d << endl;

}

void swap2(int &c, int &d)

{

int t = c;

c = d;

d = t;

cout << "c2=" << c << " d2=" << d << endl;

}

**Замечание:** Ссылка не указатель, поэтому при передаче объекта по ссылке для доступа к его членам используется оператор . (точка), а не -> (стрелка).

**Пример**: // *Использование передачи объекта по ссылке*

#include<iostream>

using namespace std;

class myclass {

int x;

public:

myclass(int n) :x(n) { cout << "Create " << x << endl; }

~myclass() { cout << "Delete " << x << endl; }

int get\_x() { return x; }

};

void func(myclass &);

int main()

{

myclass ob(1);

func(ob);

cout << "Varible x in main(): " << ob.get\_x() << endl;

system("pause");

return 0;

}

void func(myclass &ob1)

{

cout << "Local varible x: " << ob1.get\_x() << endl;

}

Результаты выполнения программы:

*Create 1*

*Local varible x: 1*

*Varible x in main(): 1*

*Delete 1*

**Замечание**: Если аргумент (объект) вызывается по ссылке, то копия объекта не создается. Это означает, что после возврата управления аргумент функции не уничтожается и деструктор не вызывается.

Ссылка в качестве возвращаемого значения функции.

**Пример**: // *Ссылка в качестве возвращаемого значения функции*

#include <iostream>

using namespace std;

int &f();

int x;

int main()

{

f() = 100; // присваивание 100 ссылке, возвращаемой функцией f()

cout << x << "\n";

system("pause");

return 0;

}

// Возвращение ссылки на целое

int &f()

{

return x; // возвращает ссылку на x

}

Независимые ссылки (обычно не используются).

Независимая ссылка – это ссылка, которая во всех случаях является просто другим именем переменной.

**Замечание:** Независимая ссылка инициализируется при ее объявлении.

**Пример**:

#include <iostream>

using namespace std;

int main()

{

int x;

int &ref = x; // создание независимой ссылки

x = 10; // эти две инструкции

ref = 10; // функционально идентичны

ref = 100;

cout << x << ' ' << ref << "\n"; // здесь дважды печатается число 100

system("pause");

return 0;

}

Ограничения, относящиеся к ссылкам всех типов:

* нельзя ссылаться на другую ссылку;
* нельзя получить адрес ссылки;
* нельзя создавать массивы ссылок;
* ссылка до ее использования должна быть инициализирована.

**Дружественные функции**

Дружественные функции используются тогда, когда существует необходимость использования одной функцией закрытых элементов двух или более классов.

Это можно сделать, объявив функцию дружественной к этому классу, используя ключевое слово friend

**Пример**: // Функция fr()объявлена как дружественная классу B.

class B {

…

public:

friend void fr(void);

…

}

Дружественной функцией может быть не только внешняя функция, но и функция-член другого класса.

* Хотя дружественная функция знает о закрытых элементах класса, для которого она является дружественной, доступ к ним она может получить только через объект этого класса (функция-член делает это непосредственно).
* Дружественная функция – это не член класса и она не может быть задана через имя объекта. Она должна вызываться так же, как и обычная функция.
* Дружественная функция не наследуется (то есть, дружественная функция не является таковой для производных классов).
* Функция может быть дружественной более чем к одному классу.
* Функция может быть членом одного класса и дружественной другому.

**Задание**: Написать программу, в которой используется два объекта для получения некоторого "секретного" кода. Известно, что код состоит из последовательности 10 целых чисел. Значения нечетных номеров этой последовательности хранятся в закрытом массиве объекта класса alpha, значения четных номеров хранятся в закрытом массиве объекта класса omega (организовать соответствующий ввод данных).

**Пример**:/\* Функция, дружественная двум классам \*/

#include <iostream>

using namespace std;

class omega; // предварительное объявление

int key[10];

class alpha {

int first[5];

public:

alpha() {

cout << "Input first 5 numbers:" << endl;

for (int i = 0; i < 5; i++)

{

cout << "i=" << i + 1 << "\t ";

cin >> first[i];

cout << endl;

}

}

friend int \*concat(alpha, omega);

};

class omega {

int second[5];

public:

omega() {

cout << "Input second 5 numbers:" << endl;

for (int i = 0; i < 5; i++)

{

cout << "i=" << i + 1 << "\t ";

cin >> second[i];

cout << endl;

}

}

friend int \*concat(alpha, omega);

};

int \*concat(alpha a, omega b)

{

int j = 0, k = 0;

for (int i = 1; i <= 10; i++)

if (i % 2)

{

key[i - 1] = a.first[j];

j++;

}

else

{

key[i - 1] = b.second[k];

k++;

}

return key;

}

int main()

{

alpha ob1;

omega ob2;

concat(ob1, ob2);

for (int i = 0; i < 10; i++)

cout << "i= " << i + 1 << "\t " << key[i] << endl;

system("pause");

return 0;

}

**Пример**:// Функция-член одного класса и дружественная другому

#include <iostream>

using namespace std;

class omega; // предварительное объявление

int key[10];

class alpha {

int first[5];

public:

alpha() {

cout << "Input first 5 numbers:" << endl;

for (int i = 0; i < 5; i++)

{

cout << "i=" << i + 1 << "\t ";

cin >> first[i];

cout << endl;

}

}

int \*concat(omega);

};

class omega {

int second[5];

public:

omega() {

cout << "Input second 5 numbers:" << endl;

for (int i = 0; i < 5; i++)

{

cout << "i=" << i + 1 << "\t ";

cin >> second[i];

cout << endl;

}

}

friend int\* alpha::concat(omega);

};

int\* alpha::concat(omega b)

{

int j = 0, k = 0;

for (int i = 1; i <= 10; i++)

if (i % 2)

{

key[i - 1] = first[j];

j++;

}

else

{

key[i - 1] = b.second[k];

k++;

}

return key;

}

int main()

{

alpha ob1;

omega ob2;

ob1.concat(ob2);

for (int i = 0; i < 10; i++)

cout << "i= " << i + 1 << "\t " << key[i] << endl;

system("pause");

return 0;

}

**Перегрузка конструкторов**

Каждому способу объявления объекта класса должна соответствовать своя версия конструктора класса. (Иначе произойдет ошибка компиляции).

Таким образом, основное использование перегрузки конструктора – это обеспечение возможности выбора способа инициализации.

# Функции и объекты в качестве параметров

Объекты передаются в функции как обычные переменные, причем работает механизм передачи параметров по значению. А это означает, что в функции создается *копия* объекта.

**Пример: //** *Передача объектов в функцию в качестве параметров*

#include <iostream>

using namespace std;

class myclass {

int x;

public:

myclass(int n) :x(n) { cout << "Create " << x << endl; }

~myclass() { cout << "Delete " << x << endl; }

int get\_x() { return x; }

};

void func(myclass);

int main()

{

myclass ob(10);

func(ob);

cout << "Varible x in main(): " << ob.get\_x() << endl;

system("pause");

return 0;

}

void func(myclass ob1)

{

cout << "Local varible x: " << ob1.get\_x() << endl;

}

Результаты выполнения программы:

*Create 1*

*Local varible x: 1*

*Delete 1*

*Varible x in main(): 1*

*Delete 1*

**Комментарий к примеру**: Из полученных результатов видно, что конструктор отработал один раз, а деструктор два раза. Это объясняется тем, что при создании копии аргумента обычный конструктор не вызывается, а вызывается конструктор копий, который и создает новую (побитовую) копию аргумента. При завершении работы функции вызывается деструктор копии объекта. А при завершении работы, когда уничтожается сам объект, деструктор вызывается во второй раз. Здесь могут возникнуть проблемы, поэтому в общем случае надо явно определять конструктор копирования.

**Возвращение объектов из функций**

Функция может возвращать объект другой функции в качестве возвращаемого значения.

**Пример**:

#include<iostream>

using namespace std;

class myclass {

int x;

public:

void set\_x(int n) { x = n; }

int get\_x() { return x; }

};

myclass func();

int main()

{

myclass ob;

ob = func();

cout << ob.get\_x() << endl;

return 0;

}

myclass func()

{

myclass y;

y.set\_x(25);

return y;

}

**Замечание**: Если объекты принадлежат одному классу, то их можно присваивать друг другу. По умолчанию присваивание происходит в результате побитового копирования.

**Создание и использование конструктора копирования**

Одной из важнейших форм перегруженного конструктора является конструктор копирования. Он используется для решения следующей проблемы: если один объект *инициализируется* другим, то создается побитовая копия присваиваемого объекта. Если объект содержит указатель на выделенную область памяти, то в копии указатель будет ссылаться на ту же самую область памяти, на которую ссылается исходный указатель. Если копия меняет содержимое области памяти, то это коснется и исходного объекта.

Существует две ситуации, когда один объект может присваиваться другому:

1. при выполнении оператора присваивания;
2. при инициализации, которая происходит следующих случаях:

* при объявлении объекта:
* **myclass x = y;**
* при создании копии объекта, передаваемого функции в качестве параметра: **func(y);**
* при создании временного объекта (например, при возврате значения функции):
* **y = func();**

**Замечание**: Конструктор копирования применяется только для инициализации (при создании объекта). При присваивании одного объекта другому конструктор копирования не вызывается (если оператор присваивания не перегружен).

**Пример**:

myclass a;

…

myclass b;

b = a; // Конструктор копирования не вызывается

**Основная форма конструктора копирования:**

**имя\_класса (const имя\_класса &obj) {**

// тело конструктора

}

**Пример**: // В этой программе ошибка (связанная с динамическим выделением памяти)

#include <iostream>

#include <cstring>

#include <cstdlib>

using namespace std;

class strtype {

char \*p;

public:

strtype(char \*s);

~strtype() {delete [] p;}

char \*get() {return p;}

};

strtype:: strtype(char \*s)

{

int l;

l=strlen(s)+1;

p=new char[l];

strcpy(p, s);

}

void show(strtype x)

{

char \*s;

s=x.get();

cout << s << "\n";

}

int main()

{

strtype a("First"), b("Second");

show(a);

show(b);

return 0;

}

**Комментарий**: Когда функция **show()** заканчивает работу, объект x удаляется. Отрабатывает деструктор, который освобождает память x.p. Но эту память продолжает использовать объект, используемый при вызове функции. Возбуждается ошибка.

// Правильный вариант предыдущей программы

/\* В этой программе используется конструктор копирования, что позволяет передавать функции объекты типа strtype. \*/

#include <iostream>

#include <cstring>

#include <cstdlib>

using namespace std;

class strtype {

char \*p;

public:

strtype(char \*s); // конструктор

strtype(const strtype &o); // конструктор копий

~strtype() {delete [] p;} // деструктор

char \*get() {return p;}

};

// Обычный конструктор

strtype::strtype(char \*s)

{

int r;

r=strlen(s)+1;

p=new char[r];

strcpy(p, s);

}

// Конструктор копий

strtype::strtype(const strtype &o)

{

int r;

r=strlen(o.p)+1;

p=new char[r]; //выделение памяти для новой копии

strcpy(p, o.p);// копирование строки в копию

}

void show(strtype x)

{

char \*s;

s=x.get();

cout << s << "\n";

}

int main()

{

strtype a("First"), b("Second");

show(a);

show(b);

return 0;

}

**Аргументы по умолчанию**

Если при вызове функции соответствующий аргумент не задан, аргумент по умолчанию позволяет присвоить параметру значение по умолчанию. Это есть форма скрытой перегрузки функций. Чтобы передать параметру аргумент по умолчанию, надо в инструкции определения функции приравнять параметр тому значению, которое надо передать, когда при вызове функции соответствующий аргумент не будет указан.

**Пример**:

void f(int a=0, int b=0);

Теперь эту функцию можно вызвать тремя способами:

С двумя заданными параметрами;

С первым заданным аргументом (в этом случае параметр b по умолчанию станет равным 0);

Без аргументов. Параметры a и b по умолчанию =0.

f(); // a и b =0

f(10); // a=10, b=0

f(10,25); // a=10, b=25

**Пример**:

#include <iostream>

using namespace std;

void f(int a = 0, int b = 0)

{

cout << "a: " << a << ", b: " << b;

cout << '\n';

}

int main()

{

f();

f(10);

f(10, 99);

return 0;

}

**Использование аргументов по умолчанию вместо перегрузки конструктора**

**Пример**:

#include <iostream>

using namespace std;

class myclass {

int x;

public:

myclass(int n = 0) { x = n; }

int getx() { return x; }

};

int main()

{

myclass o1(10); // объявление с начальным значением

myclass o2; // объявление без начального значения

cout << "o1: " << o1.getx() << '\n';

cout << "o2: " << o2.getx() << '\n';

return 0;

}

**Замечание:** Все параметры, принимающие значения по умолчанию, должны располагаться правее обычных аргументов.

**Замечание:** Аргументы, задаваемые по умолчанию, должны задаваться *только один раз*. Либо в прототипе, либо в ее определении.

**Замечание:** Аргументы по умолчанию должны быть константами или глобальными переменными. Они не могут быть локальными переменными или другими параметрами.