**Тема. Массивы объектов. Указатели. Ссылки.**

Массивы объектов.

**Массив объектов** – это просто массив, каждый элемент которого – объект.

Указатели и объекты, массивы объектов.

***Указатель на объект*** – это *переменная*, которая хранит адрес объекта в памяти компьютера. Для объявления указателя на объект используется тот же синтаксис, что и в случае объявления *указателя на переменную* любого типа:

**Samp** \* p; // объявление указателя на тип **Samp**

Помимо объявления, указатели требуют ***инициализации***, так как в противном случае они содержат *непредсказуемые* значения.

Доступ к отдельному **public**-полю объекта можно получить, используя *операцию* *точки* **(.),** связывающую поле с именем объекта **– прямой доступ,** или через указатель на этот объект, используя *операцию**стрелки* **(–>),** т.е. **косвенный доступ**.

Выражение **(\*p)** разыменовывает указатель, разрешая доступ к содержимому памяти по адресу, хранящемуся в **p**.

**Указатели и массивы объектов.** После создания массива объектов некоторого класса указателю может быть присвоен адрес массива, при этом тип указателя должен совпадать с типом массива.

Адресная арифметика с объектами. Указатель this.

**Адресная арифметика с объектами**. Под *адресной арифметикой* (*арифметика указателей*) понимают действия над указателями, связанные с использованием *адресов памяти объектов, массивов объектов*. Арифметика указателей с объектами выполняется ***относительно объекта***.

**Указатель this**. Функции каждого объекта класса имеют доступ к некому указателю **this,** который*ссылается на сам объект***.** Указатель**thi*s*** *неявно передаётся каждой функции при её вызове*. Таким образом, любая функция может узнать адрес своего ***родного*** объекта. **\*this** представляет сам объект.

***Доступ к полям объекта через указатель*** **this**. Когда происходит вызов функции, значением указателя **this** становится адрес объекта, для которого эта функция вызвана.

Указатель **this** можно интерпретировать, как любой другой указатель на объект, соответственно, его можно использовать для получения доступа к *полям-данным* объекта, на который он ссылается, используя, например, *операцию* *стрелки* **(->).**

Ссылка на объект.

**Ссылка на объект. Э**то ещё одно имя объекта. Если объявлен объект **ob** класса **Samp:**

Samp **ob**; то можно определить *ссылку* на объект как

Samp **& ref** = ob;

и тогда **ob** и **ref** обозначают один и тот же объект. Это означает, что если, **public**-*переменной* **x**объекта **ob** присвоить, например, значение **5**, выполнив оператор **ob.x = 5;** то и **ref.x** будет равно **5**. Фактически, *ссылка – это как бы адрес объекта* (поэтому при определении ссылки используется символ **&** – знак *операции адреса*), и в этом смысле она сходна с указателем на объект, однако у ссылок есть свои особенности.

Динамические объекты, массивы объектов.

**Динамические объекты.** В **C++** можно создавать *динамические объекты*. Продолжительность их жизни не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в динамической области памяти.

Для создания ***динамического объекта*** (как и для других типов данных) используют *операцию* **new** и указатель. Например:

Samp \* p; p = **new** Samp; **или**

Samp \* p = **new** Samp;

Созданный таким образом объект существует до тех пор, пока память не будет явно освобождена *операцией* **delete**. В качестве операнда должен быть задан адрес, возвращённый *операцией* **new**(значение указателя **p**):

delete **p**;

Доступ к полям объекта:

p->y = 7; (\*p).y = 7; (если **y** public)

p->setx(1);

При создании динамических объектов возможна их **инициализация**, если в классе есть конструктор с параметрами:

Samp \* p = **new** Samp(5, 6); // **x = 5 y = 6**

**Динамическими объекты** создаются и уничтожаются в процессе выполнения программы. Для доступа к членам объекта применяется *операция стрелки* (**->)** или *операция разыменования* указателя **(\*)**.

+Константные функции

**Массивы объектов.** Объекты – это переменные, и они имеют те же возможности и свойства, что и переменные любых других типов. Поэтому вполне допустимо создавать ***массивы объектов***. Синтаксис объявления массива объектов и доступ к его элементам совершенно аналогичны обычным массивам.

**Массив объектов** – это просто массив, каждый элемент которого – объект.

**// Пример 1.** Создание и использование одномерного массива объектов.

// Явная инициализация одномерного массива объектов.

// Использование конструктора по умолчанию и конструктора с параметром.

#include <iostream>

using namespace std;

class **Samp{**

int x;

**public:**

Samp(){ };

Samp(int n){x = n;}

voidsetx(int n){x = n;}

int getx(){return x;}

**};**

int **main(){**

**const** int n = 4;

Samp ob1[n]; // массив объектов из **4**-х элементов

for(int i = 0; i < ; i++) // присваивание полям элементов массива

ob1[i].setx(i); // значений от **0** до **3**

for(int i = 0; i < n; i++) // возврат значений полей объектов

cout<< ob1[i].getx()<<' '; // и вывод их на экран – **0 1 2 3**

cout<<endl;

Samp ob2[n] = { 1, 2, 3, 4 }; // вызов конструктора **4** раза

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

cout<< ob2[i].getx()<<' '; // **1 2 3 4**

}

**// Пример 2.** Явная инициализация двумерного массива объектов.

// Использование конструкторов с одним и двумя параметрами.

#include <iostream>

#include <iomanip>

using namespace std;

class **Samp{**

int x;

int y;

**public:**

Samp(int n){x = n;}

Samp(int n, int m){x = n; y = m;}

int getx(){return x;}

int gety(){return y;}

**};**

int **main(){**

const int n=3, m=2;

Samp ob1[n][m] = { 1, 2, 3, 4, 5, 6 }; // **первый** формат инициализации

for(int i = 0; i < n; i++){ // **1 2**

for(int j = 0; j < m; j++) **// 3 4**

cout<< ob1[i][j].getx()<<' '; // **5 6**

cout<<endl;

}

Samp ob2[n][m] = { // **второй** формат инициализации

Samp(1, 2), Samp(3, 4),

Samp(5, 6), Samp(7, 8),

Samp(9, 10), Samp(11, 12)

};

for(int i = 0; i < n; i++){

for(int j = 0; j < m; j++)

cout<<setw(5)<< ob2[i][j].getx()<<' '<< ob2[i][j].gety();

cout<<endl;

}

}

**Результат для ob2:**

1 2 3 4

5 6 7 8

9 10 11 12

Следует отметить, что **первый** формат инициализации для массива **ob1**, представленный в программе, используется программистами чаще, чем второй формат для массива ob2 (более длинный его эквивалент), однако следует помнить, что он работает для *двумерных массивов объектов*, *конструкторы которых* *принимают только* ***один*** *аргумент*. При инициализации массива объектов, конструкторы которых принимают несколько аргументов, необходимо использовать более длинный формат инициализации.

**// Пример 3.** Использование конструкторов по умолчанию и с двумя параметрами.

// Поля класса – символьный массив и переменная **unsigned**.

**// Вариант 1.**

#include <iostream>

#include <cstring>

// using namespace std; // **!!!**

class Student {

char name[256]; // имя

unsigned age; // возраст

public:

Student(const char\* name = "unknown", unsigned age = 0);

Student(char\*, unsigned); // конструктор с двумя параметрами

const char\* get\_name() const { return name; } // return имя

unsigned get\_age() const { return age; } // return возраст

};

Student::Student(const char\* nam, unsigned ag) {

strcpy\_s(name, 256, nam); // инициализация поля **nam**e

age = ag; // инициализация поля **age**

}

int **main()** {

using std::cout; // **!!!**

const size\_t max\_size = 3;

Student stud\_arr[max\_size] = { // инициализация

Student("Wasia", 22),

Student("Petia", 24),

Student("Olesa", 26)

};

**/\*** Student stud\_arr[max\_size] = { // инициализация

Student(), // конструктор по умолчанию

Student("Petia"), // конструктор с одним параметром

Student("Olesa", 16) // конструктор с двумя параметрами

};

**\*/**

for (size\_t i = 0; i < max\_size; i++) {

cout << stud\_arr[i].get\_name(); // return имя

cout << " ";

cout << stud\_arr[i].get\_age(); // return возраст

cout << '\n';

}

}

**Результат:**

Wasia 22

Petia 24

Olesa 26

**/\***

unknown 0

Petia 0

Olesa 16

**\*/**

**// Пример 4.**  Использование конструкторов по умолчанию и с двумя параметрами.

// Поля класса – указатель на **char** и переменная **unsigned**.

**// Вариант 2.**

#include <iostream>

#include <cstring>

// using namespace std;

class Student {

char\* name; // имя

unsigned age; //возраст

public:

Student(char\*, unsigned); // конструктор с двумя параметрами

const char\* get\_name() const { return name; } // return имя

unsigned get\_age() const { return age; } // return возраст

};

Student::Student(const char\* nam, unsigned ag) {

int len = strlen(nam);

name = new char[len + 1];

strcpy\_s(name, len + 1, nam);

age = ag;

}

int **main()** {

//то же самое, что и в **Вариант 1**

}

**Указатели и объекты. *Указатель на объект*** – это *переменная*, которая хранит адрес объекта в памяти компьютера. Для объявления указателя на объект используется тот же синтаксис, что и в случае объявления *указателя на переменную* любого типа:

**Samp** \* p; // объявление указателя на тип **Samp**

Помимо объявления, указатели требуют ***инициализации***, так как в противном случае они содержат *непредсказуемые* значения.

Доступ к отдельному **public**-полю объекта можно получить, используя *операцию* *точки* **(.),** связывающую поле с именем объекта **– прямой доступ,** или через указатель на этот объект, используя *операцию**стрелки* **(–>),** т.е. **косвенный доступ**.

// **Пример 5**. Использование указателя на объект.

#include <iostream>

using namespace std;

class **Samp**{

int x; // **private**- переменная класса

**public**:

int y; // **public**-переменная класса

void setx(int n){x = n;}

void show();

};

void Samp::show(){

cout<<" x = "<<x<<" y = "<<y<<endl;

}

int **main()**{

Samp ob, \*p; // создание объекта **ob** и указателя **p**

ob.setx(1); ob.y = 1; // установка **x** и **y**

ob.show(); // x = 1 y = 1

p = &ob; // инициализация указателя

p->setx(5); p->y = 5 // используется операция стрелка **(->)**

p->show(); // x = 5 y = 5

(\*p).y = 55; // ***разыменование*** указателя, для доступа

(\*p).setx(55); // используется операция **(.)**

(\*p).show(); // x = 55 y = 55

}

Выражение **(\*p)** разыменовывает указатель, разрешая доступ к содержимому памяти по адресу, хранящемуся в **p**.

**Указатели и массивы объектов.** После создания массива объектов некоторого класса указателю может быть присвоен адрес массива, при этом тип указателя должен совпадать с типом массива.

**// Пример 6.** Использование указателя на массив объектов.

#include <iostream>

using namespace std;

class **Samp{**

int x;

**public:**

Samp(int n){x = n;} // конструктор

int getx(){return x;} // возвращает **x**

**};**

int **main(){**

**const** int n=4;

Samp ob[n] = { 1, 2, 3, 4 }; // вызов конструктора **n** раз

Samp \* p = ob; // **p** присваиваем адрес массива **ob**

for(int i = 0; i < n; i++) // чтение массива объектов **ob**

cout<< p[i].getx()<<' '; // **1 2 3 4**

cout<<endl;

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

cout<<ob[i].getx()<<' '; // **1 2 3 4**

cout<<endl;

// p = ob; // можно опустить

for (int i = 0; i < n; i++) { **// 1 2 3 4**

cout<<p->getx()<<' ';

p++;

}

cout << endl;

**// p = ob;** // **p** присваивается адрес массива **ob**

// ...

}

В примере **6** вместо **p[i].getx()** можно использовать следующие записи:

(p+i)-> **getx()**;или (\*(p+i)). **getx()**;

**Адресная арифметика с объектами**. Под *адресной арифметикой* (*арифметика указателей*) понимают действия над указателями, связанные с использованием *адресов памяти объектов, массивов объектов*. Арифметика указателей с объектами выполняется ***относительно объекта***.

**// Пример 7.** Операции с указателями на объекты.

#include <iostream>

using namespace std;

class **Samp{**

int x;

**public:**

Samp(int n){x = n;} // конструктор с параметром

void show(){ // функция вывода

cout<<" x = "<<x<<endl;

}

**};**

int **main(){**

Samp ob(5); // создание объекта, **ob**

Samp \*p, \*q; // **p, q** – указатели

p = &ob; // инициализация **p**

q = p; // **q** присваивается значение **p**

cout<<"\n p = "<<p<<" q = "<<q<<endl;

p->show(); q->show(); **//** **5** **5**

if(p == q) cout<<"\nYes"; else cout<<"\nNo";

Samp ob1(6); // создание объекта, **ob1**

Samp\* t = &ob1;

cout<<" \np = "<<p<<" t = "<<t<<endl;

p->show(); t->show(); **//** **5** **6**

if(p == t) cout<<"\nYes"; else cout<<"\nNo"<<endl;

}

**Результат:**

p = 004FFD10 q = 004FFD10

x = 5 x = 5

**Yes**

p = 004FFD10 t = 004FFCEC

x = 5 x = 6

**No**

**Указатель this**. Функции каждого объекта класса имеют доступ к некому указателю **this,** который*ссылается на сам объект***.** Указатель**thi*s*** *неявно передаётся каждой функции при её вызове*. Таким образом, любая функция может узнать адрес своего ***родного*** объекта.

**// Пример 8.** Указатель **this.**

#include <iostream>

using namespace std;

class **Adres**{

int y;

**public:**

void showAdres(){cout<<"adres object = "<<**this**<<endl;}

};

int **main()**{

Adres ob1, ob2;

ob1. showAdres(); // вывод адреса объекта **ob1**

ob2. showAdres(); // вывод адреса объекта **ob2**

}

В функции **main()** создаются **2** объекта типа **Adres**. Затем выводятся их адреса с помощью функции **showAdres()**, которая просто выводит значение указателя **this**. При этом следует отметить, что **\*this** представляет сам объект.

***Доступ к полям объекта через указатель*** **this**. Когда происходит вызов функции, значением указателя **this** становится адрес объекта, для которого эта функция вызвана.

Указатель **this** можно интерпретировать, как любой другой указатель на объект, соответственно, его можно использовать для получения доступа к *полям-данным* объекта, на который он ссылается, используя, например, *операцию* *стрелки* **(->).**

**// Пример 9.** Использование указателя **this** для доступа к полям объекта.

#include <iostream>

using namespace std;

class **Point**{

int x;

int y;

**public:**

Point(int x, int y){**this**->x = x; **this**->y = y;}

void show(){cout<<"Coords: x = "<< this->x<<"\t y = "<<this->y <<endl;

cout<<"Coords: x = "<<x<<"\t y = "<<y <<endl;}

};

int **main()**{

Point p1(20, 50);

p1.show();

}

**Результат:**

Coords: x = 20 y = 50

Coords: x = 20 y = 50

**Ссылка на объект. Э**то ещё одно имя объекта. Если объявлен объект **ob** класса **Samp:**

Samp **ob**; то можно определить *ссылку* на объект как

Samp **& ref** = ob;

и тогда **ob** и **ref** обозначают один и тот же объект. Это означает, что если, **public**-*переменной* **x**объекта **ob** присвоить, например, значение **5**, выполнив оператор **ob.x = 5;** то и **ref.x** будет равно **5**. Фактически, *ссылка – это как бы адрес объекта* (поэтому при определении ссылки используется символ **&** – знак *операции адреса*), и в этом смысле она сходна с указателем на объект, однако у ссылок есть свои особенности.

**Динамические объекты.** В **C++** можно создавать *динамические объекты*. Продолжительность их жизни не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в динамической области памяти.

Для создания ***динамического объекта*** (как и для других типов данных) используют *операцию* **new** и указатель. Например:

Samp \* p; p = **new** Samp; **или**

Samp \* p = **new** Samp;

Созданный таким образом объект существует до тех пор, пока память не будет явно освобождена *операцией* **delete**. В качестве операнда должен быть задан адрес, возвращённый *операцией* **new**(значение указателя **p**):

delete **p**;

Доступ к полям объекта:

p->y = 7; (\*p).y = 7; (если **y** public)

p->setx(1);

При создании динамических объектов возможна их **инициализация**, если в классе есть конструктор с параметрами:

Samp \* p = **new** Samp(5, 6); // **x = 5 y = 6**

**Динамическими объекты** создаются и уничтожаются в процессе выполнения программы. Для доступа к членам объекта применяется *операция стрелки* (**->)** или *операция разыменования* указателя **(\*)**.

**// Пример 10.** Использование динамических объектов в программе.

#include <iostream>

using namespace std;

#include <cstring>

**const** int n = 20;

class **Person**{

char\* name; // поле класса – указатель на **char**

**public:**

Person(char\* str); // конструктор

void show(){cout<<name<<endl;}

~Person(){cout <<"Destructor called for person " <<name<<endl;}

};

Person::Person(char\* str){ // определение конструктора

int len = strlen(str);

name = new char[len+1];

**strcpy\_s**(name, len+1, str);

}

int **main(){**

**//** символьный массив

char str[n];

cout << "Vvedi name: "; gets\_s(str); // введём **Lena**

Person\* pob1 = new Person(str);

pob1->show();

// динамическая строка

int m;

cout << "Enter number simvolov v stroke: "; cin>>m;

cin.ignore();

char\* p = new char[m+1];

cout << "Vvedi name: "; gets\_s(p, m+1); // введём **Vova**

Person\* pob2 = new Person(p);

pob2->show();

delete pob1; // вызывается деструктор для объекта **ob1**

delete pob2; // вызывается деструктор для объекта **ob2**

cout << "End of main" <<endl;

}

**Результат:**

Enter name: **Lena**

Lena

Enter number simvolov v stroke: 30

Enter name: **Vova**

Vova

Destructor called for person Lena

Destructor called for person Vova

**End of Main**

Всё, что делает данный деструктор в примере, это выводит на консоль соответствующее сообщение. При этом выполнение самого деструктора **еще не** **удаляет сам объект**. Непосредственно удаление объекта производится в ходе явной фазы удаления, которая следует **после** выполнения деструктора.

Стоит также ещё раз отметить, что для любого класса, который не определяет собственный деструктор, компилятор сам создаёт деструктор. Например, если бы класс **Person** не имел ***явно*** определенного деструктора, то для него автоматически создавался бы следующий деструктор ***по умолчанию***:

~Person(){}

**Динамические массивы объектов.**

Samp \*p = **new** Samp[размер массива];

delete **[] p**;

Доступ к элементам динамического массива осуществляется аналогично доступа к элементам обычного динамического массива.

**// Пример 11.** Создание одномерного динамического массива объектов.

#include <iostream>

using namespace std;

class **Samp{ //** класс **Samp**

int x;

**public:**

Samp(){x = 0;} // конструктор по умолчанию

void set(int m){x = m;} // функция установки **x**

int get(){return x;} // возврат **x**

**};**

int **main(){**

int size; // **size** – размер динам. массива

cout<<"Vvedi size: "; cin>>size; // **4**

Samp\* p = new Samp[size]; // динам. массив объектов

for(int i = 0; i < size; i++){

cout<<p[i].get()<<' '; **// 0 0 0 0**

}

cout<<endl;

for(int i = 0; i < size; i++){

p[i].set(i);

// (p+i)->set(i);

// \*(p+i).set(i);

cout<<p[i].get()<<' '; **// 0 1 2 3**

}

cout<<endl;

cout<<"\*\*\*\*\*\*"<<endl; // \*\*\*\*\*\*

delete [] **p**;

}

Для задания значений полям элементов динамического массива и их вывода на консоль можно использовать и другие способы организации циклов:

1. for (Samp \*t = p; t != p + size; t++)

(\*t).set(5);

for (Samp \*t = p; t != p + size; t++)

cout <<(\*t).get()<<' '; **// 5 5 5 5**

cout<<endl;

1. for(int i = 0; i < size; i++){

p->set(i);

p++;

}

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

p--;

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

cout<<p[i].get()<<' '; **// 0 1 2 3**

cout<<endl;

1. for(int i = 0; i < size; i++)

(\*(p+i)).set(i\*3);

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

cout<<p[i].get()<<**' '; //** **0 3 6 9**

cout<<endl;

// **Пример 12**. Определить класс для нахождения суммы цифр натурального

// числа **n**. Создать динамический массив объектов данного класса и для

// каждого элемента массива (объекта) найти сумму цифр числа.

// Числа вводятся **в main().**

**// Вариант 1.**

#include <iostream>

#include <iomanip>

using namespace std;

class **Cifra**{

int n;

int sum;

**public:**

Cifra(){}; // конструктор по умолчанию **!!!**

Cifra(int n); // конструктор с параметром

void summa(); // функция получения суммы цифр числа

void show(); // функция вывода

};

Cifra::Cifra(int a){n = a;}

void Cifra::summa(){ // определение функции **summa()**

sum = 0;

int rab = n;

while(rab){

sum += rab % 10; rab /= 10;

}

}

void Cifra::show(){

cout<<"n = "<<setw(7)<<n<<setw(10)<<" summa cifr = "<<sum<<endl;

}

int **main(){**

int k, chislo;

cout<<"Enter size array: "; cin>>k;

Cifra \* p = new Cifra[k]; // создание динам. массива объектов.

for(int i = 0; i < k; i++){

cout<<"Enter number: "; cin>>chislo;

Cifra ob(chislo); // создание одного объекта

p[i] = ob;

}

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

p[i].summa();

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

p[i].show();

}

**Результат:**

Enter size array**: 3**

Enter number: 12345

Enter number: 5432

Enter number: 123

n = 12345 summa cifr = 15

n = 5432 summa cifr = 14

n = 123 summa cifr = 6

// **Пример 13**. Определить класс для нахождения суммы цифр натурального числа **n**.

// Создать динамический массив объектов данного класса.

// Числа вводятся в конструкторе.

// **Вариант 2**;

#include <iostream>

#include <iomanip>

using namespace std;

class **Cifra**{ // конструктора в классе нет

int n;

int sum;

**public:**

Cifra();

void summa();

void show();

};

Cifra::Cifra(){ // используется конструктор

cout<<"Enter number: "; cin>>n; // для инициализации массива

}

void Cifra::summa(){ // определение функции **summa()**

sum = 0; int rab = n;

while(rab){

sum += rab % 10; rab /= 10;

}

}

void Cifra::show(){

cout<<"n = "<<setw(5)<<n<<setw(10)<<" summa cifr = "<<sum<<endl;

}

int **main(){**

int k;

cout<<" Enter size array: "; cin>>k;

**Cifra\* p = new Cifra[k];** // создание динам. массива объектов.

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

p[i].summa();

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

p[i].show();

}

**Результат:**

Enter size array**: 2**

Enter number: 12345

Enter number: 5432

n = 12345 summa cifr = 15

n = 5432 summa cifr = 14