Оглавление

Раздел 1. Динамическая область памяти.

1. Указатели. Динамические переменные.

Указатель - это переменная, значением которой является адрес ячейки памяти. Или, указатель – это ссылка на какой-либо участок динамической памяти.

Переменная, на которую ссылается указатель – *динамическая переменная*, т.е. переменная из динамически распределяемой памяти.

Обычно указатели объявляют для некоторого типа данных, такие указатели называются *типизированными*. Определяя тип данных указателя, мы получаем возможность с помощью него получить доступ к объекту, на который он указывает.

Указатели принято использовать:

1) при работе с большими объемами данных: данные передаются по ссылке, что ускоряет работу с ними, т.к. нет необходимости копировать их, как при передаче по значению;
2) при работе с данными неизвестного размера, например, когда заранее не известен размер массива;
3) при использовании временных буферов памяти;
4) для создания сложных структур, таких как связные списки и т.п.

Указатели делятся на две категории: указатель на объект и указатель на функции.

Указатели на объект.

Объявление указателя имеет вид:

```
тип_указателя * имя_указателя;
```

* (звездочка) – это унарная операция раскрытия ссылки, также ее называют *разыменованием* указателя.

При объявлении указателя компилятор выделяет несколько байт в соответствии с указанным типом указателя.

Инициализировать указатель можно при объявлении, тогда объявление указателя примет вид:

```
тип_указателя * имя_указателя = значение_указателя;
```

Такой способ объявления называется явным. К имени указателя принято добавлять приставку *ptr*, что позволяет легче ориентироваться в переменных.

Указатель может принимать следующие значения:

1) явно заданный адрес участка памяти;
2) нулевое значение, соответствующее пустому указателю (*NULL*);
3) другой указатель;

4) выражение, позволяющее получить адрес объекта (& (амперсанд) - операция взятия адреса).

Инициализировать указатель можно в процессе написания программы:

```
имя_указателя = значение_указателя;
```

Пример инициализации при создании:

```
int a=6; //создали переменную типа int
int *ptra=&a; //инициализировали указатель на переменную типа int
```

Мы инициализировали переменную, доступ к которой может осуществляться как по имени *a*, так и с помощью указателя, применяя операцию разыменования *.

```
cout << “a=”<<a; //выведет a=6
cout <<”*ptra=”<<*ptra; //выведет *ptra=6
```

Если попытаться вывести значение *ptra* не применяя операции разыменования, то мы получим фактический адрес ячейки памяти, в которой хранится переменная *a*.

Инициализировать указатели всегда лучше при объявлении, это поможет избежать некоторых ошибок, связанных с особенностями компилятора. Присвоение значения в неопределённый участок памяти может вызвать аварийные ситуации. Для того чтобы избежать данной ошибки необходимо использование оператора *new*.

Оператор *new* позволяет связать указатель с пустым участком памяти.

```
указатель = new тип_данных;
```

При такой записи компилятор выделит участок памяти необходимый под заданный нами тип данных.

Пример:

```
int *ptr;//создали указатель типа int
ptr =new int; //выделили память для переменной типа int и связали с
указателем
```

После того как выделенная память станет не нужна, мы можем высвободить ее с помощью *delete*. Даже если не высвобождать память явно, то она освободится ресурсами ОС по завершению работы программы. Рекомендуется не забывать про операцию *delete*.

```
delete ptr;
```

Операторы *new* и *delete* используются для создания динамического массива данных, подробнее о нем будет рассказано ниже.

Указатели на указатели.

Частным случаем объекта, на который указывает указатель, может быть другой указатель.

Если указатель1 ссылается на указатель2, то в нем будет храниться адрес указателя2. При этом, чтобы получить значение, на которое ссылается указатель2, необходимо разыменовывать указатель1 необходимое количество раз. Делается это с помощью * используемой столько раз, каково количество указателей.

Пример:

```
int a=6;
int *ptra=&a;
int **ptr_ptra=&ptra;
cout<<**ptr_ptra; //выведет 6
```

Динамический массив данных.

Динамический массив данных используют для более эффективной работы с памятью. Допустим заранее известно, что в массиве будет не более 100 элементов. Объявив массив на 100 элементов, мы можем столкнуться с тем, что по факту у нас будет задействовано в программе только 20 элементов, а остальные ячейки памяти останутся не заняты. В масштабе 100 элементов проблема выглядит не такой уж и большой, но если представить, что изначально выделили память на несколько миллионов элементов, а при работе заполнили из них только несколько десятков, то можно оценить преимущество динамических массивов.

Рассмотрим пример создания одномерного динамического массива:

```
float *ptrarray = new float[10]; //объявили одномерный динамический массив на 10 элементов типа float
/* использование массива*/
delete[] ptrarray; //высвободили занимаемую память, [] – говорит о том, что память занята одномерным массивом
```

Теперь рассмотрим создание двумерного динамического массива:

```
float **ptrarray = new float* [2]; // две строки в массиве
for(int i=0,i<2,i++){
ptrarray[i]=new float [5]; // пять столбцов
}
/* использование массива*/
//высвобождение памяти
for(int i=0,i<2,i++)
delete[] ptrarray[i];
```

Сначала мы объявили указатель второго порядка ***ptrarray*, который ссылается на массив указателей *float* *[2]. После чего на каждом шаге цикла, выделяется память под пять элементов. В результате получается динамический массив *ptrarray*[2][5].

Важно запомнить, что объявление и высвобождение занимаемой памяти двумерного динамического массива происходит в цикле.

Указатели на функцию.

Указателю можно присвоить в качестве значения адрес функции, т.е. ее точку входа. После присваивания указатель можно будет использовать для вызова функции. Объявление имеет вид:

```
тип_возвращаемого_значения имя_функции(){ тело_функции; }
тип_возвращаемого_значения(*указатель_на_функцию)()=&имя_функции;
```

Пример:

```
void function( int count) {}; //объявление функции с параметром coint типа int
void (*ptrfunction)(int count) = function; //объявление указателя на функцию
ptrfunction(6); //вызов функции с параметром 6
```

Во время присваивания адреса функции указателю амперсанд можно упустить, тогда компилятор автоматически приведет функцию к ее адресу.

2. Процедуры и функции для работы с динамической памятью.

Для начала разберёмся, что такое динамическая память. *Динамически выделяемая память*, это память, которая резервируется не на этапе компиляции, а на этапе выполнения.

Мы уже рассмотрели пример динамического выделения памяти для массивов данных с помощью оператора *new*. Теперь рассмотрим выделение памяти с помощью библиотечной функции *malloc*().

Функция *malloc*() находится в библиотеке *stdlib.h,* она выделяет блок памяти указанного размера в байтах и возвращает указатель на начало выделенного блока. Память выделяется из оперативной памяти доступной для всех программ.

Для работы с функцией нам надо знать сколько необходимо выделить байт. Для решения данного вопроса есть несколько способов: высчитать вручную, основываясь на знаниях сколько байт занимают разные типы данных, либо использовать функцию *sizeof*().

Рассмотрим оба способа.

Таблица 1. Размер основных типов данных:

Тип	Количество байт	Диапазон принимаемых значений
bool	1	0 / 255
char	1	0 / 255
int	4	-2 147 483 648 / 2 147 483 647
float	4	-2 147 483 648.0 / 2 147 483 647.0

double	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0

Выделение памяти под тип данных *int* первым способом будет иметь вид:

```
int *ptr = malloc(4);
```

Выделение с помощью функции *sizeof()*:

```
int *ptr = malloc(sizeof(int));
```

Видно, что второй способ наиболее простой, так как нет необходимости запоминать сколько в байтах занимают нужные типы данных. Хотя для понимания процесса данную информацию полезно знать.

После выделения динамической памяти выделенный участок становиться недоступным, следовательно, после того, как память перестанет быть нужной, его необходимо явно высвободить.

Высвобождение памяти осуществляется с помощью функции *free*(). Данная функция делает участок, на который указывал указатель, доступным для других программ, поэтому хорошим тоном является обнуление указателя после его применения. Это позволяет избежать некоторых ошибок, так как без обнуления указатель продолжает ссылаться на участок памяти, который возможно уже используется другой программой.

Пример высвобождения памяти:

```
free(ptr);
*ptr=NULL;
```

Для работы с динамической памятью существуют функции *realloc*() и *calloc*().

Функция *calloc*(*num, size*) выделяет блок памяти для массива размером — *num* элементов, каждый из которых занимает *size* байт, и инициализирует все свои биты нулями. Высвобождение памяти осуществляется с помощью *free*().

```
int *ptrarray = (int*) calloc(num,sizeof(int));
```

Функция *calloc*() практически не отличается от *malloc*(), поэтому подробнее рассматривать ее не будем.

Рассмотрим функцию *realloc*(). Данная функция выполняет перераспределение блоков памяти и имеет следующий вид:

```
void * realloc( void * ptrmem, size_t size );
```

ptrmem – указатель на блок ранее выделенной памяти, *size* – необходимый размер.

Если *ptrmem* равен *NULL,* то функция будет работать как *malloc*(), т.е. вернет указатель на начало выделенного блока памяти.

Если *size* равен 0, то функция работает как *free*(), т.е. будет высвобождена занятая память и вернется нулевой указатель.

Если размер нового блока меньше старого, то функция отбрасывает данные которые не поместились.

3. Динамические структуры данных (списки, стек, очередь)

Для начала разберемся с понятием структуры.

Структуры данных

Структура – это объединение в единое целое множества именованных элементов, в общем случае разных типов.

Структуры применяются чаще всего для создания баз данных, где каждая запись должна иметь несколько полей разного типа.

Структура имеет вид:

```
struct имя_структурного_типа_данных{
        Тип_данных1 имя_элемента_структуры1;
        Тип_данных2 имя_элемента_структуры2;
…
        Тип_данныхN имя_элемента_структурыN;
};
```

Данное описание вводит новый произвольный тип данных, называемый *структурным типом*. Элементы структуры могут в свою очередь быть структурным типом.

После того, как мы определили структурный тип, мы можем описать структурированные объекты:

```
struct имя_структурного_типа_данных имя_структуры;
```

Если нет необходимости определять несколько раз в разных частях программы структурированные объекты, то допускается создание неименованных структур с непосредственным объявлением объектов структуры:

```
struct{
        Тип_данных1 имя_элемента_структуры1;
…
        Тип_данныхN имя_элемента_структурыN;
} имя_структуры;
```

Необходимо различать: имя структурного типа и имя структуры. Имя структурного типа задает внутреннее строение структуры, но не создает объект.

Для обращения к объектам, входящим в объявленную структуру используется конструкция уточненного имени:

```
имя_структуры. имя_элемента_структуры
```

При объявлении объектов структур, возможна их инициализация, следующим образом:

```
struct имя_структурного_типа_данных имя_структуры=
{значение_элемента_структуры1, …, значение_элемента_структурыN};
```

Такая запись эквивалентна:

```
struct имя_структурного_типа_данных имя_структуры;
имя_структуры. имя_элемента_структуры= значение_элемента_структуры;
```

На структурный тип разрешено определять указатель на структуру:

```
struct имя_структурного_типа *имя_указателя_на_структуру =
&имя_структуры;
```

Тогда к элементам структуры, возможно обращаться с помощью операции ->, операции доступа к элементу структуры.

```
имя_указателя_на_структуру->имя_элемента_структуры
```

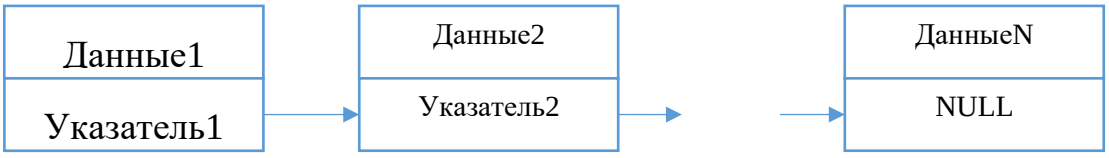
Доступ к элементу структуры можно получить с помощью операции разыменования, применённой только к указателю на структуру:

```
(*имя_указателя_на_структуру). имя_элемента_структуры
```

Как уже говорилось ранее, элементы структурного типа данных в свою очередь, могут быть структурным типом данных, причем тем же самым. С помощью данного свойства создаются списки.

Односвязный список

Односвязный список содержит, помимо информации, указатель на следующий элемент и в общем виде выглядит так:




```
struct Имя_структурного_типа{
    Тип_данных имя_элемента_структуры;
    …
```

```
struct Имя_структурного_типа *next;
};
```

Существует два способа построения односвязного списка: либо новые элементы помещаются в конец списка, либо они помещаются по определённому алгоритму.

Давайте рассмотрим первый способ.

Для начала нам необходим структурный тип:

```
struct list{
        int data;
        struct list *next;
} temp=NULL;
```

Теперь опишем функцию, которая будет добавлять элементы к списку:

```
void add (struct list *newitem, struct list **last){ // *newitem – указатель на новую запись, last – указатель на последний элемент
        if(!(*last))
      *last = newitem; // то есть *newitem первый элемент
        else
              (*last)->next = newitem; // добавляем в конец списка
        newitem ->next = NULL;
        *last = newitem;
}
```

Вызов данной функции будет иметь вид:

```
struct list newitem={10,NULL};
add(newitem,temp.next);
```

Второй вариант добавления элемента в список заключается в том, чтобы поместить элемент, например, в середину списка. Для такого действия необходимо выполнить следующие этапы:

1) создать добавляемый узел, заполнить его;
2) переустановить указатель узла, предшествующего добавляемому, на добавляемый;
3) установить указатель у добавляемого на следующий идущий элемент.

Таким образом функция будет иметь следующий вид:

```
struct list *additem(struct list *lst, int number){
        struct list *temp, *p;
        temp = (struct list*)malloc(sizeof(list));
        p = lst->ptr;
        lst->ptr = temp;
        temp->field = number;
        temp->ptr = p;
```

```
        return(temp);
}
```

В результате работы функции будет возвращен указатель на добавленный элемент.

Важной операцией по работе со списками является удаление элементов из него. Напишем функцию, которая будет возвращать указатель на элемент, следующий за удаляемым.

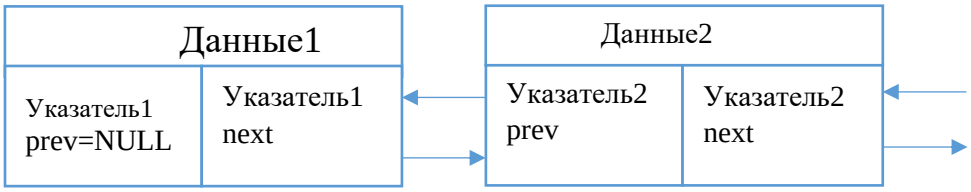
```
struct list *delete(struct list *lst, struct list *root){
        struct list *temp;
        temp = root;
        while(temp->ptr != lst){ // прохождение по списку до элемента, предшествующему удаляемому
                temp = temp->ptr;
        }
        temp->ptr = lst->ptr; // переставляем указатель
        free(lst); // освобождаем память из-под удаляемого элемента
        return(temp);
}
```

Функция выполняет следующие этапы:

1) установка указателя предыдущего узла на узел, следующий за удаляемым;
2) освобождение памяти из-под удаляемого элемента.

Двухсвязный список

Двухсвязный список основан на односвязном, за исключением того, что в элементе хранятся ссылки на следующий и на предыдущий элементы.



Двухсвязные списки имеют вид:

```
struct имя_структурного_типа{
        тип_данных имя_элемента_структуры;
        …
        struct имя_структурного_типа *prev;
struct Имя_структурного_типа *next;
};
```

Например, узел может быть представлен в виде следующей структуры:

```
struct list {
	int field;
	struct list *next;
	struct list *prev;
}
```

Главное для списка – его инициализация для создания корневого узла.

```
struct list *init(int a){
	struct list *lst;
	lst = (struct list*)malloc(sizeof(struct list));
	lst->field = a;
	lst->next = NULL;
	lst->prev = NULL;
	return(lst);
}
```

Добавление любого узла в рассмотренный нами вид списка разбивается на следующие этапы:

1) создание добавляемого узла и заполнение его полей;
2) переустановка указателя следующего узла у предшествующего добавляемому элементу на добавляемый элемент;
3) переустановка указателя предыдущего узла у следующего добавляемому элементу на добавляемый элемент;
4) установка указателей у добавленного элемента.

Напишем функцию для добавления элемента в список:

```
struct list* add(struct list *lst, int number){
	struct list *temp, *p;
	temp = (struct list*)malloc(size(list));
	p = lst->next;
	lst->next = temp;
	temp->field = number;
	term->next = p;
	temp->prev = lst;
	if(p!=NULL){
		p->prev = temp;
	}
	return(temp);
}
```

Удаление элемента из двусвязного списка основано на том, что необходимо установить указатель *next* предыдущего узла на следующий элемент за удаляемым и изменить указатель *prev* по аналогичному правилу. Не забываем про освобождение памяти из-под удаляемого элемента.

Раздел 2. ООП

1. Основные принципы ООП: инкапсуляция, наследование (общая форма наследования), полиморфизм.

Наследование – конструирование новых, более сложных производных классов-потомков, из уже имеющихся базовых классов-родителей, с помощью добавления новых полей и/или методов.

Полиморфизм – механизм, обеспечивающий возможность определения различных описаний некоторого единого метода (единого по названию) для классов различного уровня иерархии.

Инкапсуляция – механизм скрытия данных внутри объекта.

2. Классы. Объекты.

В ООП представление данных в программе и их взаимодействие имеет свои ограничения.

Класс – это произвольный структурированный тип, введённый на основе уже существующих типов. Определение классов в общем случае похоже на определение структуры, т.к. структура является частным случаем класса:

```
class имя_класса { список_компонентов };
```

Определение класса должно располагаться в глобальной области.

Класс может содержать в себе следующие виды полей:

1) данные;
2) функции;
3) классы;
4) перечисления;
5) битовые поля;
6) дружественные функции и классы;
7) имена типов.

Список компонентов называется *телом класса*. Принадлежащие классу функции называются *методами*. Данные класса называют *полями данных*.

Так как класс является типом данных, то он служит для определения объектов:

```
имя_класса имя_объекта;
```

К компонентам объекта класса можно обращаться с помощью «*квалифицированных*» имен:

```
имя_объекта.имя_класса :: имя_компонента
```

Имя класса при использовании операции указания области действия :: может быть упущено.

Возможно использовать уточненное имя как в структурах:

```
имя_объекта.имя_элемента
```

С помощью уточненного имени возможен вызов методов:

```
имя_объекта.имя_функции();
```

При определении класса можно задать *статический компонент*. Статические компоненты создаются в единственном экземпляре для всех объектов класса.

```
class имя_класса {
    static тип_данных имя_компонента;
};
```

Для инициализации статического компонента используется конструкция, расположенная в глобальной области после определения класса:

```
тип_данных имя_класса::имя_компонента = значение;
```

При работе с классами возможно создание *массивов объектов*.

Для создания статического массива объекта будет использована следующая структура:

```
имя_класса имя_массива_объектов[количество_объектов];
```

Работа с полями объекта будет выглядеть так:

```
имя_массива_объектов[номер_объекта].имя_компонента = значение;
```

3. Конструктор класса

Для инициализации объектов любого класса существует специальный метод, принадлежащий классу и имеющий вид:

```
имя_класса (список_формальных_параметров) {
    операторы_тела_конструктора;
};
```

Класс содержащий конструктор будет иметь вид:

```
имя_класса {
    тип_компонента_класса имя_компонента_класса1;
...
    имя_класса (тип_параметра параметр1=значение по умолчанию, …) {
        имя_компонента_класса1= параметр1;
};
};
```

Обратите внимание, что имя класса и имя конструктора класса должны совпадать.

Конструктор класса будет автоматически вызываться при определении или размещении в памяти с помощью *new*. Если не указывать явно конструктор, то он создается автоматически без параметров.

Конструктор по умолчанию – конструктор, который может быть вызван без аргументов.

Для конструктора не определяется тип возвращаемого значения в соответствии с синтаксисом. В списке формальных параметров можно задать значения по умолчанию. При автоматическом вызове используются значения параметров по умолчанию. В конструкторе можно изменять значения инициализированного статического компонента.

Рассмотрим задание параметров по умолчанию, на примере комплексных чисел:

```
class complex1 {
float a; //вещественная часть
	float b; //мнимая часть
public:
	complex1(float a1 = 0.0, float b1 = 0.0) { //конструктор принимающий значения вещественно и мнимой части по умолчанию равной 0.0
		a = a1; b = b1;
	}
	void display(void) {
		printf("%f+i*%f\n",a,b);
	}
};
complex1 x;
x.display();
complex1 y(5);
y.display();
complex1 z(6,7);
z.display();
```

Вывод:

0.00000+*i**0.00000

5.00000+*i**0.00000

6.00000+*i**7.00000

В классе может быть несколько конструкторов, это называется *перегрузкой*. Запомните, что только один конструктор должен содержать параметры по умолчанию для всех параметров. При перегрузке нужный конструктор будет вызываться самостоятельно в зависимости от параметров, используемых при инициализации.

При работе с конструктором есть несколько ограничений:

1) нельзя получить его адрес;

2) параметрами не может быть собственный класс, но может быть ссылка на него;

3) нельзя вызывать конструктор как обычную компонентную функцию.

Для явного вызова конструктора используют конструкцию:

```
имя_класса имя_объекта(параметры);
```

В рассмотренном ранее примере с комплексными числами, мы использовали способ инициализации конструктора с помощью передачи значения параметров в тело конструктора. Существует еще один способ инициализации с помощью списка инициализаторов данных объекта – *конструктор с параметрами*. При таком способе инициализации конструктор будет иметь вид:

```
имя_класса (список_формальных_параметров)
:список_инициализаторов_данных{
    Операторы_тела_конструктора;
};
```

Для комплексных чисел конструктор примет вид:

```
complex1(float a1 = 0.0, float b1 = 0.0) : a(a1), b(b1) { };
```

Выражение инициализации, заключенное в круглые скобки может содержать в себе различные операции, допустимые для данного типа данных.

Конструкторы с параметрами удобны при наследовании, так как они позволяют не прописывать их логику, если она была уже реализована в родителе.

4. Деструктор

При вызове конструктора выделяется динамическая память, поэтому есть необходимость высвобождать память, если класс перестает быть нужным. Для высвобождения памяти используется специальная функция – *деструктор класса*. Деструктор имеет вид:

```
~имя_класса () {операторы_тела_деструктора};
```

Название деструктора всегда начинается с символа ~(тильда). Деструктор не может иметь возвращаемое значение. Вызов деструктора совершается неявно, когда объект класса уничтожается.

Деструкторы базовых классов выполняются в порядке, обратном перечислению классов в определении производного класса.

5. Доступность компонентов класса

Для того, чтобы выполнялся основной принцип абстракции данных – инкапсуляция данных внутри объектов, необходимо сделать так, чтобы компоненты класса не были доступны в любом месте программы. Для этого предусмотрены спецификаторы доступа. Изначально все компоненты

введенные с помощью слова *struct* общедоступны, т.е. имеют спецификатор доступа *public*. Существуют также спецификаторы доступа: *private* – собственные компоненты класса, и *protected* - защищенные компоненты класса, т.е. доступ открыт только классам производным от данного. Все компоненты класса начинающиеся со служебного слова *class* являются собственными (*private*).

После спецификатора следует двоеточие, все компоненты класса, указанные от одного спецификатора до другого, имеют указанный статус.

В примере с комплексными числами мы использовали спецификатор *public*, чтобы сделать функции общедоступными.

6. Ключевое слово this.

this – служебное слово, указатель на конкретный объект, в котором происходит та или иная работа в данный момент времени. Обращение к нему происходит следующим образом:

```
this->pole;
this->func().
```

Указатель *this* не требуется объявлять, он существует всегда как неявный. При входе в тело функции, принадлежащей классу, *this* инициализируется значением адреса того объекта, для которого была вызвана функция.

Данное ключевое слово удобно использовать, когда в методе присутствуют переменные, имена которых совпадают с именами полей класса. В этом случае обращение к переменной происходит как обычно – по ее имени, а обращение к полям класса – через ключевое слово *this* (*this->param*).

Служебное слово *this* используют, когда в теле функции необходимо явно задать адрес того объекта, для которого она вызвана.

7. Иерархия классов. Наследование

Для построения иерархии классов используются защищённые компоненты классов. Это необходимо при порождении классов на основе других.

При наследовании используется следующая терминология: *базовые (порождающие, родительские) классы* – уже имеющиеся классы, *производные (порожденные, классы-потомки, наследники)* - новые классы, создаваемые на основе базовых. Любой производный класс, в свою очередь, может быть базовым для другого, при этом у производного класса будет возможность доступа ко всем своим базовым классам.

При наследовании в новый класс передаются все компоненты базового класса, кроме конструктора и деструктора. При создании объекта производного класса, сначала вызывается конструктор базового класса, а затем производного. Наследуемые компоненты не перемещаются в

производный класс, а остаются в базовом. Все, что не может выполнить производный класс с помощью своих методов, передается в базовый класс автоматически.

При наследовании, некоторые имена методов или данные базового класса, могут быть переопределены в производном, при этом для доступа к переопределенным компонентам из базового класса используют операцию :: - операцию указания области видимости.

Наследование может быть нескольких типов. Типы определяются правами доступа к полям класса. Шаблон создаваемого класса выглядит следующим образом:

```
class имя_нового_класса-потомка : вид_наследования имя_класса-родителя {
    тело класса;
}
```

По умолчанию вид наследования *private* для *class*. Возможно наследование сразу из нескольких классов, тогда базовые классы перечисляются через запятую.

Режимы доступа при наследовании классов.

Таблица 2. Режимы доступа при наследовании классов.

Спецификатор доступа перед базовым классом	Объявление компонентов в классе-родителе	Видимость компонентов в классе-потомке
private	Private	Не доступны
	Protected	private
	Public	private
protected	Private	Не доступны
	Protected	protected
	Public	protected
public	Private	Не доступны
	Protected	protected
	Public	public

8. Методы – элементы класса.

Методы класса –это принадлежащие классу функции, они иногда называются *компонентными функциями*. Методы, так же, как и функции могут возвращать какие-либо значения.

Пример описания методов:

```
class Print {
	void Hello(){
		cout << “Hello! ” << endl;
	}
	void Bye(){
		cout << “Bye! ” << endl;
	}
	int Summ(int a, int b){ // метод, принимающий два параметра и возвращающий их сумму
		return a + b;
	}
};
```

Доступ к методам (их вызов) происходит аналогично доступу к полям класса:

```
Print temp = new Print(); // создаем объект класса
temp.Hello(); // на экране появится надпись «Hello! »
temp.Bye(); // на экране появится надпись «Bye! »
cout << Summ(4,5) << endl; // на экран будет выведена надпись «9»
```

Перекрытие методов.

Перекрытие методов позволяет дочернему классу обеспечить специфическую реализацию метода из родительского класса.

Пример перекрытия методов:

```
class A {
	public: void fun(int);
};
class B: A {
	void fun(long);
};
B obj;
obj.fun(1L); // аргумент функции – единица типа long
```

В данном примере будет запущена функция из класса *B*, в которой принимается аргумент типа *long*.

Можно вызвать метод родительского класса в наследуемом классе с помощью конструкции:

```
родительский_класс: метод (аргументы);
```

Например:

```
class A {
	public:
		void print(){
			cout << «I’m A» << endl;
		}
};
class B : public A {
	void print(char *str){
		cout << str << endl;
		A::print(); // вызываем метод родительского класса
	}
};
B obj;
obj.print(“Hello”);
```

На экран будет выведено:

Hello

I’m A

Статические методы.

Статический метод означает, что он один на все экземпляры класса. Статические методы могут использоваться без создания экземпляра класса и могут работать со статическими переменными и с объектами класса, при явном указании на объект.

При использовании статических методов, говорят о *раннем связывании,* когда объект и вызов функции связываются между собой на этапе компиляции, т.е. на этапе компиляции уже известно, какая именно функция будет вызвана. Примером могут служить обычные системные функции или вызываемые статические методы класса.

```
class A {
	public:
		static void print(){
			cout << «I’m static method» << endl;
		}
};
class B : public A {};
B obj;
obj.print();
```

В консоль будет выведено:

I'm static method

Виртуальные методы. Таблица виртуальных методов.

Виртуальный метод – метод, объявленный в базовом классе, в классе наследнике он имеет то же имя и сигнатуру параметров, но выполняется по-другому. Это возможно благодаря механизму *динамического (позднего) связывания* - объект и вызов функции связывается уже во время исполнения программы.

Классы, включающие в себя виртуальные функции, называются *полиморфными*.

Любая, нестатическая функция базового класса, может быть сделана виртуальной с помощью спецификатора *virtual*.

Тип возвращаемого параметра виртуальной функции и у базового, и у наследуемого класса должен совпадать. Если объявить в производном классе функцию с таким же именем, таким же возвращаемым значением, но с другими параметрами, то это уже не виртуальная функция, а новый метод производного класса.

Пример создания виртуального метода:

```
class Language {
public:
	virtual void PrintError(int n){ // объявляем виртуальный метод
		cout << “Null” << endl;
}
};
class CPP : public Language {
	void PrintError(int n){
		if(n == 1){
			cout << “Fatal error!” << endl;
		}
}
};
void main(){
	Language *obj = new CPP();
	obj->PrintError(1);
	return 0;
}
```

В данном примере, мы объявили виртуальную функцию *PrintError* в базовом классе, и переопределили ее в дочернем. Такое возможно в том случае, если объект будет создаваться от родительского класса, но с использованием конструктора дочернего класса. Это показывает возможность в объекте родительского класса содержать объекты любого дочернего класса. Это используется в основном для массивов на основе базового класса.

В консоль будет выведено после выполнения примера:

Fatal error!

Основное применение виртуальных методов - для создания массива на основе базового класса, содержащего разные объекты производных классов. Если при этом, не использовать виртуальные методы, то всегда будут вызываться функции исключительно базового класса.

Пример:

```
class Language {
public:
	virtual void PrintError(int n) {
		cout << "Null" << endl;
	}
};

class CPP : public Language {
	void PrintError(int n) {
		if (n == 1) {
			cout << "Fatal error from CPP!" << endl;
		}
	}
};

class CSHARP : public Language {
	void PrintError(int n) {
		if (n == 1) {
			cout << "Fatal error from C#!" << endl;
		}
	}
};

int main()
{
	Language *obj[2];
	obj[0] = new CPP();
	obj[1] = new CSHARP();
	obj[0]->PrintError(1);
	obj[1]->PrintError(1);
	system("PAUSE");
	return 0;
}
```

Вывод в консоль:

Fatal error from CPP!

Fatal error from C#!

Таблица виртуальных методов – предназначена для вызова нужных реализаций виртуального метода во время исполнения программы.

Применение виртуальных функций, связанное с использованием указателей на объекты.

Создадим указатель на объект от класса *Language*, для предыдущего примера:

```
Language *lang = new Language;
```

Теперь мы можем вызвать виртуальный метод, если он чем-либо заполнен:

```
lang->PrintError(0);
```

Стоит отметить, что класс родитель может содержать объект любого типа-потомка. Например:

```
Language *lng = new CPP;
```

Виртуальные деструкторы.

Деструктор полиморфного базового класса должен быть виртуальным, только тогда обеспечивается корректное разрушение объекта производного класса через указатель на базовый класс.

```
class A{
public:
	A(){cout << "Конструктор А" << endl;};
	virtual ~A() { cout << "Деструктор А" << endl;};
virtual void print() = 0;
};
class B : public A{
public:
	B(){cout << " Конструктор В " << endl;};
	~B(){cout << "Деструктор В" << endl;};
void print() {}
};
int main(){
	A *ptr = new B;
delete ptr;
return 0;
}
```

Вывод:

Конструктор *A*

Конструктор *B*

Деструктор *B*

Деструктор *A*

В функции *main* указателю присваивается адрес динамически создаваемого объекта производного класса. Затем, через этот указатель, объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Уничтожение объекта производного класса через указатель на базовый класс с не виртуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу.

Дружественные функции.

Дружественные функции класса – функции, которые не являются компонентом некоторого класса, имеющие доступ ко всем его компонентам. Для получения доступа функция должна быть описана в теле класса со спецификатором *friend*. При наличии данного спецификатора, класс предоставляет функции права доступа к защищенным (*private*) и внутренним (*protected*) компонентам.

Важно отметить, что дружественные функции при вызове не получают указателя *this*. При вызове дружественной функции нельзя использовать операции выбора через уточненное имя и указатель на объект, т.к. дружественная функция не является компонентом класса. На дружественную функцию не действуют спецификаторы доступа. Дружественная функция может быть глобальной.

Пример:

```
class A {
	int x;
	friend void fun(A *, int);
	public:
	A(int x1) : x(x1) {};
	void print(void){
		printf("%d\n",x);
	}
};
void fun(A *ptr, int x){
	ptr->x=x;
}
int main(){
	A temp(2);
	temp.print();
	fun(&temp,10);
```

```
	temp.print();
	return 0;
}
```

Выведет:

2

10

Если убрать в классе описание дружественной функции, то компилятор выдаст ошибку доступа к *x*, т.к. он является приватным.

9. Дружественные классы.

Класс может быть дружественным другому классу, т.е. все функции одного класса являются дружественными для другого.

```
class A {
friend class B;
тело_класса;
};
class B {
тело_класса;
void fun;
};
```

Функция *fun*() из класса *B*, является другом класса *A*. Все компоненты класса *A* доступны в классе *B*.

Пример использования дружественных классов:

```
class A {
private:
	int count = 100;
public:
	friend class B;
};

class B {
public:
	void fun(A *obj) {
		cout << obj->count << endl;
	}
};
int main()
{
	A *object = new A;
	B *obj = new B;
```

```
        obj->fun(object);
        system("PAUSE");
    return 0;
}
```

Данный пример выводит в консоль «100» и это демонстрирует то, что класс *B* имеет доступ к закрытому (*private*) блоку полей класса *A*.

10. Абстрактный класс.

Абстрактный класс – класс, в котором есть хотя бы один чистый (обнуленный) виртуальный метод. Объекты таких классов создавать запрещено. Он служит основой для производных классов.

Чистые виртуальные функции.

Чистые виртуальные функции – функции, которые объявляются в базовом классе как виртуальные и не содержащие описания выполняемых действий. Шаблон объявления:

```
virtual возвращаемый_тип имя функции (список_параметров) = 0;
```

Конструкция =0 называется *чистым спецификатором*.

Чистая виртуальная функция недоступна для вызова, единственная ее задача - служить основой для подменяющих ее функций в производных классах.

Если в производном классе чистая виртуальная функция не подменяется, то данный класс становиться абстрактным.

Пример:

```
class A {
protected:
        virtual void fan(int) = 0;
        void print(int);
};
class B: public A{
        …
        void fan(int);
};
class C: public A{
        void print(int);
};
```

A и *C* являются абстрактными классами, в то время как *B* не абстрактный.

11. Функции

Параметры и аргументы функции. Передача объектов как аргументов функций. Использование объекта в качестве возвращаемого значения.

Объекты можно передавать как аргументы функции и использовать в виде полей других классов.

Приведем пример использования объектов в качестве полей в других объектах:

```
class Temp {
	int a;
}
class MyClass {
	Temp obj;
}
```

После создания объекта класс *MyClass*:

```
MyClass *object = new MyClass;
```

Мы можем обратиться к полю *a* класса *Temp* следующим образом:

```
object->obj.a = 2;
```

Пример использования объекта как аргумента функции:

```
void printObj(MyClass *object){
	cout << object->obj.a << endl; // выведем значение поля объекта
}
```

Подставляемые (inline) функции.

Подставляемые функции – это такие функции, которые подставляют свое тело на место вызова.

Шаблон объявления *inline* функции:

```
inline возвращаемый_тип имя_функции (список_параметров ){
тело_функции; }
```

Данный механизм позволяет ускорить выполнение кода, т.к. на вызов функции уходит много времени. При этом подставляемые функции могут значительно увеличить размер программы.

Пример использования:

```
#include <iostream>
using namespace std;
inline void hello(){
cout << “Hello World” << endl;
}
```

```
int main(){
	hello(); // вызываем inline функцию
}
```

Компилятор при первичной обработке представит код в следующем виде, который будет быстрее компилироваться:

```
#include <iostream>
using namespace std;
int main(){
	cout << “Hello World” << endl;
}
```

Точно также будет происходить подстановка функций из подключаемых библиотек.

Существуют случаи, когда *inline* функция будет трактоваться как обычная:

1) слишком большой размер встраиваемой функции (по количеству операций);
2) встраиваемая функция является рекурсивной;
3) обращение к встраиваемой функции размещено до ее определения;
4) встраиваемая функция вызывается более одного раза;
5) встраиваемая функция содержит цикл, переключатель или оператор перехода.

Перегрузка функций

Перегруженные функции – это функции, имеющие одинаковое имя, но разную сигнатуру функций. Это может потребоваться если функция должна, к примеру, подсчитывать сумму элементов в массиве независимо от его типа. Тогда потребуется написать несколько разных функции с одним и тем же именем.

Приведем пример для *int* и *float*.

```
void count (int n, int mass[]){
	int rez = 0;
	for(int i=0; i<n; i++){
		rez=rez+mass[i];
	}
	printf("%d\n",rez);
}
void count (int n, float mass[]){
	float rez = 0;
	for(int i=0; i<n; i++){
		rez=rez+mass[i];
```

```
        }
        printf("%f\n",rez);
}
int main() {
int i[]={2, 3, 4};
float f[]={2.5, 4.1, 3.4};
count(3,i);
count(3,f);
        return 0;
}
```

При перегрузке функции надо с осторожностью использовать параметры по умолчанию.

12. Перегрузка стандартных операций

Перегрузка стандартных операций дает возможность определить поведение операторов (таких как +, -, *, / и других), для собственных классов. Для перегрузки создается отдельный метод, который имеет следующую структуру:

```
тип_возвращаемого_значения operator знак_операции (параметры_функции)
{
        тело функции;
}
```

При перегрузке бинарной операции, входящей в число компонентов класса, у нее должен быть только один параметр функции. При этом если операция будет переопределяться не для класса, то возможно использование нескольких параметров.

Пример класс с перегрузкой операции:

```
class Complex {
        public:
        int a;
        int b;
        Complex(int a1=0, int b1=0) {
              a=a1;
              b=b1;
};
        Complex operator +(Complex& a);
};
Complex Complex::operator + (Complex& a){
                Complex temp;
                temp.a=a.a + this->a;
                temp.b=a.b + this->b;
```

```
        return temp;
};
int main(){
Complex a(3,4);
Complex b(8,50);
Complex c = a+b;
printf("%d+i*%d",c.a,c.b);
        return 0;
}
```

В результате в объекте c будет поле *a* = 11 и поле *b* = 54.

Существуют операции, для которых невозможна перегрузка: . , * ? : :: *sizeof* # ##

13. Шаблоны функций.

Шаблоны функций – это инструкции, согласно которым создаются шаблонные функции. Шаблон функции имеет следующий вид:

```
template <список_параметров_шаблона> определение_функции
```

Каждый формальный параметр шаблона обозначается служебным словом *class*, за которым следует имя параметра. В определении функции тип возвращаемого значения и тип любого параметра обозначается именем параметра шаблона. Параметры шаблона можно использовать в теле функции в качестве локального типа. Все параметры шаблона функции должны быть обязательно использованы в определении функции. Список параметров шаблона не должен быть пустым.

Главная особенность в использовании шаблонов функций – это то, что они могут быть только глобальными.

На основании шаблона компилятор автоматически формирует конкретные функции по использованному вызову функции.

Шаблоны более удобный механизм перегрузки функций, без необходимости подготовки нескольких вариантов функции. Компилятор все сделает автоматически.

Пример создания шаблонной функции:

```
template <class T>
void sum(T a, T b){
        return a + b;
}
```

Теперь в эту функцию можно отправлять переменные любого типа, для которых действительна операция сложения через операцию +.

Для шаблонов функции возможна перегрузка: написание шаблонов с одинаковыми именами, но разным списком параметров.

Пример использования шаблонов функции:

```
template <class T>
T summ(T a, T b) {
	return a + b;
}
int main()
{
	cout << summ(3.2, 9.8) << endl;
	cout << summ(3, 9) << endl;
	system("PAUSE");
	return 0;
}
```

На консоль будет выведено следующее:

13

12

Благодаря шаблонам, мы создали в примере такую функцию, которая может принимать переменные и значения любого типа, над которыми возможна операция сложения. Главное, чтобы передавались значения одного типа, т.к. мы у обоих аргументов указали *T*, который в данном случае в шаблоне указывает на используемый тип. *C++* при выполнении определяет тип аргументов функции и заменяет *T* на *int*/*double*/*float* и т.п. имена типов.

14. Шаблоны классов.

Шаблоны классов – это инструкции, согласно которым можно создавать классы с полями различного типа. Шаблон класса почти аналогичен шаблону функций:

```
template <список_параметров_шаблона> определение_класса
```

В шаблоне класса имя класса является параметризированным именем семейства классов.

Шаблоны классов также, как и шаблоны функции могут быть только глобальными.

Рассмотрим пример написания шаблона класса:

```
template <class T>
class Stack {
	T a;
};
```

Теперь при создании объекта необходимо задать тип:

```
Stack <int> obj;
```

В данном примере поле *a* будет определено типом *int* и компилятор будет видеть определение объекта *obj* следующим образом:

```
class Stack{
        int a;
} obj;
```