Лабораторная работа 2.

Разработка класса Планета и создание класса по варианту (Приложение 4)

Оглавление

Цель работы:

- создание программных объектов пользовательских типов с использованием классов;
- создание много-файловых проектов (заголовочный файл для класса, файл с определением методов класса и файл с main-функцией);
- ввод-вывод объектов пользовательских типов в файл;
- перегрузка операций потокового ввода-вывода (<<, >>), операций отношения (<, ==) для пользовательских типов;
- перегрузка конструктора копирования и операции присваивания;
- сортировка массивов объектов пользовательских типов, хранящихся в файле, с использованием перегруженных операций.

Задание.

1.1

1. Ознакомиться с приведенным ниже материалом (Введение и Пример 1). Перейти в примере 1 от структуры к классу.
2. Разработать класс «Планета» для планет солнечной системы (4 характеристики планет разного типа приведены в Приложении 1). Имя планеты должно иметь тип char*.
3. Создать много-файловый проект и отладить программу, которая создает один объект класса «Планета» и выводит значения его полей на экран. (Пример программы приведен в Приложении 3).
4. Организовать интерфейс пользователя с программой в виде меню, позволяющий выполнять следующие действия:
   - чтение БД «Солнечная система» из файла;
   - запись БД «Солнечная система» в файл;
   - сортировка БД;
   - добавление новой планеты в БД;
   - удаление планеты из БД;
   - редактирование БД;
   - вывод БД на экран.
5. Создать текстовый файл (в блокноте) с данными о планетах солнечной системы и сохранить его в папке проекта. Первоначально, для отладки, введите две записи.
6. Добавить в программу ввод – вывод объекта класса «Планета» в текстовый файл.

1.2

1. Ознакомиться с содержанием раздела «Перегрузка операций» и выполнить приведенный в нем пример 2.
2. Перегрузить конструктор копирования, деструктор и оператор присваивания.

3. Вставьте в конструкторы и деструктор печать типа «Создание (Удаление) ID ***n*** » , где ***n*** - номер объекта, для которого они вызываются. (Для реализации этого пункта задания разберите и выполните пример из Приложения 2.).
4. Перегрузить операцию “ >> “ для класса «Планета» и ***ifstream*** и прочитать данные о планетах из файла в массив «Солнечная система» из объектов класса «Планета».
5. Перегрузить операцию “ << “ для классов «Планета» и ***ofstream*** и вывести на экран данные из массива.

1.3

1. Перегрузить операции сравнения “ < и ==” для класса «Планета», использовав для этого значение одного из полей.
2. Отсортировать массив планет солнечной системы, хранящийся в файле, с использованием перегруженных операций.

1.4

На основе разработанного класса Планета выполнить задание по варианту (не менее 4 характеристик в классе разного типа).

Требования к отчету

Отчет должен содержать следующие разделы:

- «Постановка задачи», в котором на основании задания уточняются задачи, для решения которых предполагается использовать разрабатываемый класс.
- «Разработка интерфейса класса», в котором описываются и обосновываются состав полей и методов класса, прототипы методов. Интерфейс класса должен обеспечить решение всех предполагаемых задач. При разработке интерфейса класса надо руководствоваться принципом: второстепенные детали или детали реализации должны быть упрятаны (инкапсулированы) внутрь класса.
- «Текст программы», в котором приведены исходные тексты разработанной программы. При защите лабораторной работы студент должен уметь объяснить назначение каждого оператора разработанной им программы.
- «Анализ результатов», в котором приводятся тестовые примеры, распечатки результатов выполнения программой тестовых примеров и анализ результатов.

Введение

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленного программного продукта. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Существенная черта промышленной программы — ее *сложность:* один разработчик не в состоянии охватить все аспекты системы, поэтому в ее создании участвует целый коллектив. Следовательно, к первичной сложности самой задачи, вытекающей из предметной области, добавляется управление процессом разработки с учетом необходимости координации действий в команде разработчиков.

Так как сложные системы разрабатываются в расчете на длительную эксплуатацию, то появляются еще две проблемы: *сопровождение* системы (устранение обнаруженных ошибок) и ее *модификация,* поскольку у заказчика постоянно появляются новые требования и пожелания. Иногда затраты на сопровождение и модификацию сопоставимы с затратами на собственно разработку системы.

Способ управления сложными системами был известен еще в древности — разделяй и властвуй. То есть выход — в *декомпозиции* системы на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо.

В рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса.

Объектно-ориентированная программа строится в терминах объектов (типа «класс») и их взаимосвязей, а декомпозиция представляет собой иерархию классов, при которой потомки наследуют свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Иерархия должна строиться таким образом, чтобы классы, стоящие выше в иерархии, содержали бы общие свойства классов – потомков.

Откуда же берутся классы? Исключительно из головы программиста, который, анализируя предметную область, вычленяет из нее отдельные объекты. Для каждого из этих объектов определяются свойства, существенные для решения поставленной задачи. Затем каждому реальному объекту предметной области ставится в соответствие программный объект.

Почему *объектно-ориентированная декомпозиция* оказалась более эффективным средством борьбы со сложностью процессов проектирования и сопровождения программных систем, чем *функциональная* декомпозиция? Тому есть много причин.

Оценка качества декомпозиции проекта

Со *сложностью приложения* трудно что-либо сделать — она определяется целью создания программы. А вот *сложность реализации* можно попытаться контролировать. Первый вопрос, возникающий при декомпозиции: на какие компоненты (модули, функции, классы) нужно разбить программу? Очевидно, что с ростом числа компонентов сложность программы растет, поскольку необходима кооперация, координация и коммуникация между компонентами. Особенно негативны последствия неоправданного разбиения на компоненты, когда оказываются разделенными действия, по сути тесно связанные между собой.

Вторая проблема связана с организацией взаимодействия между компонентами. Взаимодействие упрощается и его легче взять под контроль, если каждый компонент рассматривается как некий «черный ящик», внутреннее устройство которого неизвестно, но известны выполняемые им функции, а также «входы» и «выходы» этого ящика. Вход компонента позволяет ввести в него значение некоторой *входной переменной,* а выход — получить значение некоторой *выходной переменной.* В программировании совокупность входов и выходов черного ящика определяет *интерфейс* компонента. Интерфейс реализуется как набор некоторых функций (или запросов к компоненту), вызывая которые *клиент* либо получает какую-то информацию, либо меняет состояние компонента.

Слово «клиент» означает просто-напросто компонент, которому понадобились услуги другого компонента, исполняющего в этом случае роль *сервера.* Взаимоотношение *клиент/сервер* на самом деле очень старо и использовалось уже в рамках структурного программирования, когда функция-клиент пользовалась услугами функции-сервера путем ее вызова.

Подытожим сказанное о проблемах разбиения программы на компоненты и организации их взаимодействия. Для оценки качества программного проекта нужно учитывать, кроме всех прочих, следующие два показателя:

- *Сцепление* внутри компонента — показатель, характеризующий степень взаимосвязи отдельных его частей. Простой пример: если внутри компонента решаются две подзадачи, которые легко можно разделить, то компонент обладает слабым (плохим) сцеплением.
- *Связанность)* между компонентами — показатель, описывающий интерфейс между компонентом-клиентом и компонентом-сервером. Общее число входов и выходов сервера есть мера связанности. Чем меньше связанность между двумя компонентами, тем проще понять и отслеживать в будущем их взаимодействие. А так как в больших проектах эти компоненты часто разрабатываются разными людьми, то очень важно уменьшать связанность между компонентами.

Следует заметить, что описанные показатели, конечно, имеют относительный характер, и пользоваться ими следует благоразумно. Например, фанатичное следование первому показателю (сильное сцепление) может привести к дроблению проекта на очень большое количество мелких функций, и сопровождающий программист вряд ли помянет вас добрым словом.

Что принесло с собой ООП

Первым бросающимся в глаза отличием ООП от структурного программирования является использование классов. *Класс* — это тип, определяемый программистом, в котором объединяются структуры данных и функции их обработки. Конкретные переменные типа данных «класс» называются *экземплярами класса,* или *объектами.* Программы, разрабатываемые на основе концепций ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Класс содержит константы и переменные, называемые *полями,* а также выполняемые над ними операции и функции. Функции класса называются *методами.* Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Поля и методы являются *элементами,* или *компонентами* класса.

Эффективным механизмом ослабления связанности между программными компонентами в случае объектно-ориентированной декомпозиции является так называемая инкапсуляция.

Инкапсуляция — это ограничение доступа к данным и их объединение с методами, обрабатывающими эти данные. Доступ к отдельным частям класса регулируется с помощью специальных ключевых слов: public (открытая часть), private (закрытая часть) и protected (защищенная часть).

Методы, расположенные в открытой части, формируют *интерфейс* класса и могут свободно вызываться клиентом через соответствующий объект класса. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — из его собственных методов, а также из методов классов-потомков. Инкапсуляция повышает надежность программ, предотвращая непреднамеренный ошибочный доступ к полям объекта. Кроме этого, программу легче модифицировать, поскольку при сохранении интерфейса класса можно

менять его реализацию, и это не затронет внешний программный код (код клиента).

С понятием инкапсуляции тесно связано понятие *сокрытия информации.* С другой стороны, понятие сокрытия информации соприкасается с понятием *разделения ответственности* между клиентом и сервером. Клиент не обязан знать, *как* реализованы те или иные методы в сервере. Для него достаточно знать, что делает данный метод и как к нему обратиться. При хорошем проектировании имена методов обычно отражают суть выполняемой ими работы, поэтому чтение кода клиента для сопровождающего программиста превращается просто в удовольствие.

Заметим, что класс одаривает своего программиста-разработчика надежным «укрытием», обеспечивая локальную (в пределах класса) область видимости имен. Теперь можно сократить штат бригады программистов: специалист, отвечающий за согласование имен функций и имен глобальных структур данных между членами бригады, стал не нужен. В *разных* классах методы, реализующие схожие подзадачи, могут преспокойно иметь *одинаковые* имена. То же относится и к полям разных классов.

С ООП связаны еще два инструмента, грамотное использование которых повышает качество проектов: наследование классов и полиморфизм.

Наследование — механизм получения нового класса из существующего. Производный класс создается путем дополнения или изменения существующего класса. Благодаря этому реализуется концепция повторного использования кода. С помощью наследования может быть создана иерархия родственных типов, которые совместно используют код и интерфейсы.

Полиморфизм дает возможность создавать множественные определения для операций и функций. Какое именно определение будет использоваться, зависит от контекста программы. Вы уже знакомы с одной из разновидностей полиморфизма в языке C++ — перегрузкой функций.

В реальном проекте, разработанном на базе объектно-ориентированной декомпозиции, находится место и для алгоритмически-ориентированной декомпозиции (например, при реализации сложных методов).

От структуры — к классу

Прообразом класса в C++ является структура в C. В то же время в C++ структура обрела новые свойства и теперь является частным видом класса, все элементы которого по умолчанию являются открытыми. Со структурой struct в C++ можно делать все, что можно делать с классом. Тем не менее в C++ структуры обычно используют лишь для удобства работы с небольшими наборами данных без какого-либо собственного поведения.

Пример 1.

```
/*В текстовом файле хранится база отдела кадров предприятия. На предприятии не более
100 сотрудников.
Каждая строка файла содержит запись об одном сотруднике. Первая запись в файле –
фактическое число сотрудников. Формат записи:
      фамилия (не более 20 позиций),
      год рождения (4 позиции),
      оклад (не более 8 позиций).
Написать программу, которая позволяла бы выводить на экран сведения о сотрудниках,
добавлять и удалять сотрудников из БД, корректировать данные о сотрудниках.
*/

#include <iostream>
#include <iomanip>
#include <fstream>

using namespace std;

#define l_name  20

struct Man {
      char name[l_name];
      int birth_year;
      float pay;
};

int read_dbase(char* filename,Man* arr,int& n);
int  menu();
int menu_f();
void print_dbase(Man* arr,int n);

int  write_dbase(char* filename, Man* arr, int n);
/*
```

```
int add(Man arr, int n);
int edit(Man* arr, int n);
int remove(Man* arr, int n);
*/
int find(Man* arr,int n, char* name);
int find(Man* arr,int n, int birth_year);
int find(Man* arr,int n, float pay);
void find_man(Man* arr,int n);

//---------------------------------------  Главная функция
int main()
{
	const int N=100;
	Man arr[N];
	char *filename="dbase.txt";
	int n;
 //чтение БД в ОП
if (read_dbase(filename,arr,n)){
	cout<<"Ошибка чтения БД"<<endl;
	return 1;
}
print_dbase(arr,n);
while (true) {
	switch (menu())
	{
	//case 1: add(arr,n ); break;
	//case 2: remove(arr,n); break;
	case 3: find_man(arr,n);  break;
	//case 4: edit(arr,n); break;
	case 5: print_dbase(arr,n); break;
	case 6: write_dbase(filename,arr,n); break;
	case 7: return 0;
	default: cout<<" Недопустимый номер операции"<<endl; break;
	}
 }
 return 0;
}
////////////////////////////////////////////////////////
int menu( )
{
cout<<" ============== ГЛАВНОЕ МЕНЮ ========================\n";
	cout<<"1 - добавление сотрудника\t 4 - корректировка сведений"<<endl;
	cout<<"2 - удаление сотрудника\t\t 5 - вывод базы на экран"<<endl;
	cout<<"3 - поиск сотрудника\t\t 6 - вывод базы в файл"<<endl;
	cout<<"\t\t\t\t 7 - выход"<<endl;
	cout<<"Для выбора операции введите цифру от 1 до 7"<< endl;
int resp;
cin>>resp;
cin.clear();
cin.ignore(10,'\n');
return resp;
}

// ------------------------------------   Чтение базы из файла
int read_dbase(char* filename, Man* arr, int& n)
{
ifstream fin(filename,ios::in);
if (!fin)
	{
	cout<<"Нет файла "<< filename <<endl;
	return 1;
	}
fin>>n;
if(n>100)
{
```

```
	cout<<"Переполнение БД. n= "<< n <<endl;
	return 1;
	}
for(int i=0;i<n;i++)
	fin>>arr[i].name>>arr[i].birth_year>>arr[i].pay;

fin.close();
return 0;
}
//-----------------------------------    Вывод базы в файл
int write_dbase(char *filename, Man* arr,int n)
{
	ofstream fout(filename,ios::out);
	if (!fout)
	{
		cout<<"Ошибка открытия файла"<<endl;
		return 1;
	}
	fout<<n;
	for(int i=0; i<n; i++)
		fout<< arr[i].name<<' '<< arr[i].birth_year<<' '<< arr[i].pay<<endl;

	fout.close();
	return 0;
}
//------------------------------ Вывод базы на экран
void print_dbase(Man* arr,int n)
{
cout<<" База Данных "<<endl;
for(int i=0;i<n;i++)
	cout<<setw(3)<<i+1<<". "<<arr[i].name<<setw(20-strlen(arr[i].name)+6)
		<<arr[i].birth_year<<setw(10)<<arr[i].pay<<endl;
}
//------------------------------Поиск сотрудника в списке по фамилии
int find(Man* arr , int n, char* name) //возвращает индес элемента с данными о
                                       //сотруднике в БД,реализованной в виде массива
{
int ind=-1;
for(int i=0;i<n;i++)
	if (!strcmp(arr[i].name, name))
		{
		cout<<arr[i].name<<setw(20-strlen(arr[i].name)+6)
			<<arr[i].birth_year<<setw(10)<<arr[i].pay<<endl;
			ind=i;
		}
return ind;
}
//------- Поиск и вывод более старших по возрасту сотрудников
int find(Man* arr , int n, int birth_year)
{
int ind=-1;
for(int i=0;i<n;i++)
	if (arr[i].birth_year < birth_year)
	{
		ind=i;
		cout<<arr[i].name<<setw(20-strlen(arr[i].name)+6)
			<<arr[i].birth_year<<setw(10)<<arr[i].pay<<endl;
	}
return ind;
}
//-------- Поиск и вывод сотрудников с окладом, большим чем "pay"
int find(Man* arr , int n, float pay)
{
int ind=-1;
for(int i=0;i<n;i++)
```

```
	if (arr[i].pay > pay)
	{
		ind=i;
		cout<<arr[i].name<<setw(20-strlen(arr[i].name)+6)
			<<arr[i].birth_year<<setw(10)<<arr[i].pay<<endl;
	}
return ind;
}
//-------------------------------------
int menu_f()
{
	cout<<"\n----------------- ПОИСК -----------------\n";
	cout<<"1 - поиск по фамилии    2 - по году рождения\n"
		<<"3 - по окладу         4 - конец поиска\n ";
		cout<<"Для выбора операции введите число от 1 до 4\n";
	int resp;
	cin>> resp;
	cin.clear();
	cin.ignore(10,'\n');
	return resp;
}
//-------------------------------------    Поиск
void find_man(Man* arr, int n)
{
char buf[l_name];
int birth_year;
float pay;

while(true)
	{
	switch (menu_f())
		{
		case 1: cout<<"Введите фамилию сотрудника\n";
				cin>> buf;
				if(find(arr, n, buf)<0)
				cout<<"Сотрудника с фамилией "<< buf<<" в списке нет\n";
				break;
		case 2: cout<<"Введите год рождения"<< endl;
				cin>> birth_year;
				if(find(arr, n, birth_year)<0)
					cout<< "В списке нет сотрудников, родившихся до "
						<<birth_year<<" года\n";
				break;
		case 3: cout<<"Введите оклад"<< endl;
				cin>> pay;
				if(find(arr, n, pay)<0)
					cout<< "В списке нет сотрудников с окладом, большим "
						<< pay<<" руб.\n";
				break;
		case 4: return;
		default:
				cout<<"Неверный ввод\n";
		}

	}
}
```

В программе, предложенной для решения задачи, при структурном программировании для хранения сведений об одном сотруднике использовалась бы структура Man:

```
struct Man {
	char name[l_name];
	int birth_year;
	float pay;
};
```

Начнем с того, что преобразуем эту структуру в класс, так как мы предполагаем, что наш новый тип будет обладать более сложным поведением, чем просто чтение и запись его полей:

```
class Man {
       char name[l_name];
       int birth_year;
       float pay;
};
```

Замечательно. Это у нас здорово получилось! Все поля класса по умолчанию — закрытые (private). Так что если клиентская функция main() объявит объект Man man, а потом попытается обратиться к какому-либо его полю, например: man.pay = value, то компилятор быстро пресечет это безобразие, отказавшись компилировать программу. Поэтому в состав класса надо добавить методы доступа к его полям. Эти методы должны быть общедоступными, или открытыми (public).

Однако предварительно вглядимся внимательнее в определения полей. В решении задачи на языке Си поле name объявлено как статический массив длиной l_name. Это не очень гибкое решение. Мы хотели бы, чтобы наш класс Man можно было использовать в будущем в разных приложениях. Например, если предприятие находится в России, то значение l_name = 20, по-видимому, всех устроит, если же приложение создается для некой восточной страны, может потребоваться, скажем, значение l_name = 200. Решение состоит в использовании динамического массива символов с требуемой длиной. Поэтому заменим поле char name[l_name] на поле char* pName. Сразу возникает вопрос: кто и где будет выделять память под этот массив? Вспомним один из принципов ООП: все объекты должны быть самодостаточными, то есть полностью себя обслуживать.

Таким образом, в состав класса необходимо включить метод, который обеспечил бы выделение памяти под указанный динамический массив при создании объекта (переменной типа Man). Метод, который автоматически вызывается при создании экземпляра класса, называется *конструктором.* Компилятор безошибочно находит этот метод среди прочих методов класса, поскольку его имя всегда совпадает с именем класса.

Парным конструктору является другой метод, называемый *деструктором,* который автоматически вызывается перед уничтожением объекта. Имя деструктора отличается от имени конструктора только наличием предваряющего символа ~(тильда).

Ясно, что если в конструкторе была выделена динамическая память, то в деструкторе нужно побеспокоиться об ее освобождении. Напомним, что объект, созданный как локальная переменная в некотором блоке { }, уничтожается, когда при выполнении достигнут конец блока. Если же объект создан с помощью операции new, например:

```
Man* pMan = new Man;
```

то для его уничтожения применяется операция delete, например: `delete pMan;` Итак, наш класс принимает следующий вид:

```
class Man {
public:
       Man(int l_name = 20) { pName = new char[l_name]: }      // конструктор
       ~Man() { delete[] pName: }                              // деструктор
private:
       char* pName;
       int birth_year :
       float pay :
};
```

Обратим ваше внимание на одну *синтаксическую деталь* — объявление класса должно обязательно завершаться точкой с запятой (;). Если вы забудете это сделать, то получите от компилятора длинный список маловразумительных сообщений о чем угодно, но только не об истинной ошибке.

Рассмотрим теперь одну важную *семантическую деталь:* в конструкторе класса параметр l_name имеет значение по умолчанию (20). Если все параметры конструктора имеют значения по умолчанию или если конструктор вовсе не имеет параметров, он называется *конструктором по умолчанию.* Зачем понадобилось специальное название для такой разновидности конструктора? Разве это не просто удобство для клиента — передать некоторые значения по умолчанию одному из методов класса? Нет! Конструктор — это особый метод, а конструктор по умолчанию имеет несколько специальных областей применения.

Во-первых, такой конструктор используется, если компилятор встречает определение массива

объектов, например: `Man man[25];` Здесь объявлен массив из 25 объектов типа Man, и каждый объект этого массива при создании вызывает конструктор по умолчанию! Поэтому если вы забудете снабдить класс конструктором по умолчанию, то вы *не сможете объявлять массивы* объектов этого класса. Исключение представляют классы, в которых нет ни одного конструктора, так как в таких ситуациях конструктор по умолчанию создается компилятором.

Вернемся к приведенному выше описанию класса. В нем методы класса *определены* как *встроенные* (inline) функции. При другом способе методы только *объявляются* внутри класса, а их реализация записывается вне определения класса, как показано ниже:

```
// Man.h (интерфейс класса)
class Man {
public:
       Man(int I_name = 30);       // конструктор
       ~Man();                     // деструктор
private:
       char* pName;
       int    birth_year;
       float pay;
};
// Man.cpp (реализация класса)
#include "Man.h"
Man::Man(int l_name) { pName = new char[l_name]; }
Man::~Man() { delete[] pName; }
```

При внешнем определении метода перед его именем указывается имя класса, за которым следует операция доступа к области видимости ::. Выбор способа определения метода зависит в основном от его размера: короткие методы можно определить как встроенные, что может привести к более эффективному коду. Впрочем, компилятор все равно сам решит, может он сделать метод встроенным или нет.

Продолжим процесс проектирования интерфейса нашего класса. Какие методы нужно добавить в класс? С какими сигнатурами? На этом этапе очень полезно задаться следующим вопросом: *какие обязанности* должны быть возложены на класс Man?

Первую обязанность мы уже реализовали: объект класса хранит сведения о сотруднике. Чтобы воспользоваться этими сведениями, клиент должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Начнем с методов, обеспечивающих доступ к полям класса. Для считывания значений полей добавим методы GetName(), GetBirthYear(), GetPay(). Очевидно, что аргументы здесь не нужны, а возвращаемое значение совпадает с типом поля.

Для записи значений полей добавим методы SetName(), SetBirthYear(), SetPay(). Чтобы определиться с сигнатурой этих методов, надо представить себе, как они будут вызываться клиентом.

Константные методы. Обратите внимание, что заголовки тех методов класса, которые *не должны изменять поля* класса, снабжены модификатором const после списка параметров. Если вы по ошибке попытаетесь в теле метода что-либо присвоить полю класса, компилятор не позволит вам это сделать. Другое достоинство ключевого слова const — оно четко показывает сопровождающему программисту намерения разработчика программы. Например, если обнаружено некорректное поведение приложения и выяснено, что «кто-то» портит одно из полей объекта класса, то сопровождающий программист сразу может исключить из списка подозреваемых методы класса, объявленные как const. Поэтому использование const в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования.

Отладочная печать в конструкторе и деструкторе. Вывод сообщений типа «Constructor is working», «Destructor is working» очень помогает на начальном этапе освоения классов. Да и не только на начальном — мы сможем убедиться в этом, когда столкнемся с проблемой локализации неочевидных ошибок в программе.

> Вставляйте отладочную печать типа «Здесь был я!» в тела конструкторов и деструкторов, чтобы увидеть, как работают эти невидимки. Использование этого приема особенно полезно при поиске трудно диагностируемых ошибок.

Перегрузка операций

Любая операция, за исключением “::”, “?:”, “.”, “.*”, определенная в C++, может быть перегружена для созданного вами класса. Это делается с помощью функций специального вида, называемых *функциями-операциями* (операторными функциями). Общий вид такой функции:

возвращаемый_тип operator # (список параметров) { тело функции }

где вместо знака # ставится знак перегружаемой операции.

Функция-операция может быть реализована либо как функция класса, либо как внешняя (обычно дружественная) функция. В первом случае количество параметров у функции-операции на единицу меньше, так как первым операндом при этом считается сам объект, вызвавший данную операцию.

Например, покажем два варианта перегрузки операции сложения для класса Point.

Первый вариант — в форме метода класса:

```
class Point {
	double x, y;
public:
	//. . .
	Point operator +(Point&);
};
Point Point::operator +(Point& p)
{
	return Point(x + p.x, y + p.y);
}
```

Второй вариант — в форме внешней глобальной функции, причем функция, как правило, объявляется дружественной классу, чтобы иметь доступ к его закрытым элементам:

```
class Point {
	double x, y;
public: //. . .
	friend Point operator +(Point&, Point&);
};
Point operator +(Point& p1, Point& p2)
{
	return Point(p1.x + p2.x, p1.y + p2.y);
}
```

Независимо от формы реализации операции “+” мы можем теперь написать:

```
Point p1(0, 2), p2(-1, 5);
Point p3 = p1 + p2;
```

Следует понимать, что, встретив выражение p1 + p2, компилятор в случае первой формы перегрузки вызовет метод p1.operator +(p2), а в случае второй формы перегрузки — глобальную функцию operator +(pl, p2).

Результатом выполнения данных операторов будет точка p3 с координатами x = -1, y = 7. Заметим, что для инициализации объекта p3 будет вызван конструктор копирования по умолчанию, но он нас устраивает, ***поскольку в классе нет полей-указателей.*** (Подумайте, почему указатели нельзя просто скопировать, чтобы получить копию объекта?)

Если операция может перегружаться как внешней функцией, так и функцией класса, какую из двух форм следует выбирать? Ответ: используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, int), то перегрузка операции возможна только в форме внешней функции. (Вспомните, что по умолчанию с помощью указателя ***this*** в методы класса неявно, скрытым первым параметром, передается адрес объекта, вызвавшего метод).

Пример 2.

Пример перегрузки операции << для вывода в файл некоторой структуры element.

```
//overload.cpp  -  запись  структур в  файл перегруженной
//     операцией <<
#include <iostream>
#include <fstream>
```

```
using namespace std;
struct element {     // Определение некоторой структуры
	int nk, nl;
	float zn;
};
//  Операция-функция,   расширяющая действие операции <<
ofstream& operator<<(ofstream& out, element el) {
	out << ' ' << el.nk << ' ' << el.nl << ' ' << el.zn << '\n';
	return out;
}
int main()
{
	const int numbeEl = 5; // Количество структур в массиве
	element arel[numbeEl] = { 1, 2, 3.45, 2, 3, 4.56,
		22, 11, 45.6, 3, 24, 4.33, 3, 6, -5.3 };
	// Определяем поток и связываем его с новым файлом abc:
	ofstream filel("abc.txt", ios::app);
	if (!filel)
	{
		cout << "Неудача при открытии файла abc.\n";
		return 1;
	}
	// Запись в файл abc массива структур:
	for (int i = 0; i < numbeEl; i++)
		filel << arel[i];
	return 0;
}
```

Результат выполнения программы - создание файла с именем `abc.txt` в текущем каталоге и запись в этот файл элементов массива из пяти структур **element.** Содержимое файла abc**:**

```
 1    2    3.45
 2    3    4.56
 22   11   45.6
 3    24   4.33
 3    6    5.3
```

Приложение 1. Исходные данные

Название	Диаметр	Жизнь	Спутники
Mercury	4878	0	0
Venus	12104	0	0
Earth	12774	1	1
Mars	6786	1	2
Jupiter	142796	0	16
Saturn	120000	0	17
Uranus	51108	0	5
Neptune	49600	0	2
Pluto	2280	0	1

Приложение 2. Статические члены класса.

```
class gamma
{
private:
	static int total;    //всего объектов
	                              //(только объявление)
	int id;              //ID текущего объекта
```

```
public:
	gamma()		//конструктор без аргументов
	{
		total++;	//увеличить счетчик объектов
		id = total;	//id равен текущему значению total
		cout << "Создание ID " << id << endl;
	}
	~gamma()	//деструктор
	{
		total--;
		cout << "Удаление ID " << id << endl;
	}
	static void showtotal() // статическая функция
	{
		cout << "\nВсего: " << total << endl;
	}
	void showID() // нестатическая функция
	{
		cout << "\nID: " << id << endl;
	}
};
//---------------------------------
int gamma::total = 0;
void main()
{
	gamma::showtotal();
	gamma g1;
	gamma g1.showtotal();
	gamma g2, g3;
	g3.showtotal();
	g1.showID();
	g2.showID();
	g3.showID();
	cout << "Конец программы" << endl;
}
//-----------------------------------------------------------
```

Приложение 3. Пример программы, использующей класс Planet

Файлы:
planet.h – интерфейс класса,
planet.cpp – определение методов класса.

```
// sunsys.cpp
#include <fstream>
#include "planet.h"
#include "planet.cpp"
//#include "ConsolCyr.h"
using namespace std;

int read_db(char*, Planet*, const int);
int  menu();
void print_db(Planet*, int);
int write_db(char*, Planet*, int);
int find(Planet*, int);
void sort_db(Planet*, int);
const int Size = 12;
const int l_record = 80;

int main()
{
	char *file_name = "sunsys.txt";
	Planet planets[Size];
	int n_planet;
	int ind;
```

```
	while (true) {
		switch (menu())
		{
		case 1: n_planet = read_db(file_name, planets, Size);
			break;
		case 2: write_db(file_name, planets, n_planet); break;
		case 3: if ((ind = find(planets, n_planet)) >= 0)
			planets[ind].edit();
				else
					cout << "Такой планеты нет" << endl;
			break;
		case 4: print_db(planets, n_planet); break;
		case 5: sort_db(planets, n_planet); break;
		case 6: return 0;
		default: cout << " Неправильный ввод" << endl; break;
		}
	}
	return 0;
}
//---------------------------------------------------------
```

Приложение 4. **Варианты для выполнения п.1.4.**

1	квартира, как объект для агентства
2	автобус в автопарке
3	анкета для опроса населения
4	компьютер
5	кандидат, участвующий в выборах
6	железнодорожный билет
7	файл на диске
8	книга в библиотеке
9	политическая партия
10	кафедра института
11	автомобиль
12	авиабилет
13	статья в журнале
14	магазин
15	абонент телефонной станции
16	управление каталогом в файловой системе
17	пациент в поликлинике
18	дом, как объект ЖЭКа
19	строительная бригада
20	пищевой набор диеты
21	дорога
22	музыкальный альбом
23	программное обеспечение
24	заявки на выполнение работ
25	товары в магазине
26	участник соревнований
27	фильмотека
28	винотека