Лабораторная работа №4.
Работа с базовыми и производными классами

Цель работы: изучение принципов наследования в объектно-ориентированной парадигме программирования.

Содержание

Работа с базовыми и производными классами

Введение

Объектно-ориентированный подход к проектированию программных систем является элементом так называемых наукоемких технологий проектирования программ. Использование этого подхода дает возможность на порядок по сравнению с обычным директивным программированием сократить трудоемкость отладки программ и внесения изменений в программу при ее последующем развитии. Платой за это является наукоемкость проектирования, т.е. уделение весьма большой части времени на детальную проработку предметной области программы, составление структуры данных и их взаимосвязи, а также проектирование программной архитектуры.

Вместе с тем, объектно-ориентированное программирование существенно отличается от классических методов программирования, в том числе структурного и модульного программирования. При этом коренным образом ломается понятие об алгоритме как о последовательности выполнения операторов языка программирования, записанных друг за другом. В объектно-ориентированных программах все данные разбиваются на отдельные группы и строго связываются с программами (функциями), предназначенными для обработки этих данных. Любая из функций как бы присоединяется к тем данным, для обработки которых она предназначена. Такое объединение данных с программами в единое целое носит название инкапсуляции. Сам результат объединения является самостоятельным объектом программы и почти всегда действительно соответствует какому-либо из объектов той предметной области, для которой написана программа. Такая структура программы дает возможность не только быстро локализовать логические ошибки, но и с высокой эффективностью вносить изменения в программу при ее доработке для получения новых версий.

1.1 Простое наследование

В C++ существуют специальные средства передачи всех определяемых пользователем свойств класса (как данных, так и функций-методов) другим классам, наследующим свойства данного.

Один класс может наследовать все составляющие другого класса. Класс, передающий свои компоненты другому, называют **базовым классом**. Класс, принимающий эти компоненты, называется **производным классом**. Способность класса пользоваться методами, определенными для его предков, составляет сущность принципа наследуемости свойств. Например, можно определить в программе класс **«Человек»** с компонентами **«Фамилия»**, **«Имя»**, **«Отчество»** и **«год рождения»**. Функции-методы, которые могут понадобиться при работе с объектами этого класса, такие, как: **«ЗадатьФамилию»**, **«ДатьГодРождения»**, **«ВывестиНаЭкранФИО»** и т.д., также будут определены в этом классе. Если в программе понадобится определить какой-либо другой объект, скажем, **«Работник»** или **«Студент»**, становится очевидным, что этот класс будет частным случаем класса **«Человек»**,

поскольку все компоненты класса **«Человек»** необходимо полностью переписать в класс **«Работник»** или **«Студент»**. Намного удобнее в этом случае просто каким-либо образом при определении таких производных классов сослаться на базовый класс, сказав тем самым, что все его компоненты заимствуются. В том же случае, если нужно доопределить в производном классе собственные компоненты, например **«Зарплата»** или **«НомерЗачетнойКнижки»**, их можно описать в производном классе заново. Такой механизм наследования действительно существует в C++.

Производный класс строится на базе уже существующего с помощью конструкции следующего вида:

```
class Base {
// Элементы базового класса
};

class Derived: [модификатор доступа] Base {
// Элементы производного класса
};
```

При определении производного класса за его именем следуют разделитель - двоеточие (:), затем необязательный модификатор доступа и имя базового класса. Модификатор доступа определяет область видимости наследуемых компонентов для производного класса и его возможных потомков.

Рассмотрим пример.

```
// Базовый класс
class Level0 {
private: int a;
protected: int b;
public: int c;
void f0();
};

// Производный класс
class Level1: public Level0 {
private: int d;
protected: int e;
public: int f;
void f1();
};
```

В приведенном примере производный класс **Level1** наследует компоненты базового класса **Level0**. Производный класс содержит все компоненты базового, а также компоненты, определенные в самом производном классе.

1.2. Доступ к наследуемым компонентам

Доступность различных составляющих (компонентов) класса в производном классе определяется соответствующим ключем доступа, задаваемым словами **private**, **public** или **protected**.

Частный (**private**) компонент класса доступен только другим компонентам и друзьям этого класса, тогда как общий (**public**) компонент доступен и вне данного класса. Частные компоненты базового класса для производных классов являются недоступными.

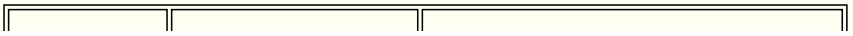
Программист может позволить производным классам доступ к конкретным компонентам базового. C++ имеет также третью категорию доступности компонентов класса, называемую защищенной (**protected**). Защищенные компоненты недоступны ни для каких частей программы, за исключением компонентов производных классов.

Класс может быть унаследован как **public** или как **private**. При этом модификатор **private** трансформирует компоненты базового класса с атрибутами доступа **public** и **protected** в компоненты **private** производного класса, в то время как **private-компоненты** становятся недоступны в производном классе.

Модификатор наследования **public** не изменяет уровня доступа. Производный класс наследует все компоненты своего базового класса, но может использовать только те из них, которые определены с атрибутами **public** и **protected**.

Разные типы наследования влияют только на доступ по умолчанию компонентов базового класса в производном классе. Правила наследования доступа показаны в табл.4.1.

Таблица 4.1



Доступ наследования	Доступ компонентов в базовом классе	Доступность компонентов базового класса в производном классе
public	public protected private	public protected недоступен
private	public protected private	private private недоступен

При объявлении класса-потомка с помощью ключевого слова **class** статусом доступа по умолчанию является **private**, а при объявлении с помощью ключевого слова **struct** - **public**, то есть

```
struct D: B{... };
```

означает:

```
struct D: public B{ public:...};
```

Компонент, наследуемый как **public**, сохраняет тот же тип доступа, что был у него в базовом классе. В следующем фрагменте допустимыми являются только заданные типы доступа.

```
                    // Базовый класс
                class Level0 {
                       private: int a;
                      protected: int b;
                       public: int c;
                           int e;
                         void f0();
                           };

                  // Производный класс
              class Level1a: public Level0 {
                      protected: int d;
                       public: int f;
                         void f1();
                           };

        // Обычная функция - имеет доступ только к public-компонентам
                          void fn() {
                            Level0 L0;
                           Level1a L1;
                            L0.e = 1;
                       // public-компонент
                            L1.e = 1;
        // public-компоненты из Level0 являются также public и в Level1a
                            L1.f = 2;
                            L1.f0();
                            L1.f1();
                             }

                      // Компонентные функции
                        void Level0::f0() {
                    // имеет доступ ко всему Level0
                             a = 1;
                             b = 2;
                             c = 3;
                              }

                       void Level1a::f1() {
                             b = 1;
                    // доступа к a не имеет
                             c = 2;
                             d = 3;
                    // имеет доступ ко всему Level1a
                             e = 4;
                             f = 5;
                              f0();
                              }
```

В следующих частных производных классах **L1.c** и **L1.f0()** внешней функции **fn()** не доступны, поскольку они являются частными, хотя **L0.c** и **L0.f0()** продолжают оставаться доступными.

Доступность компонентов для компонентных функций **f0()** и **f1()** остается неизменной.

```
class Level1b: private Level0 {
          private: int d;
        protected: int e;
          public: int f;
            void f1();
              };

// аналогично Level1b
class Level1c: Level0 {
          private: int d;
        protected: int e;
          public: int f;
            void f1();
              };

// Общая функция
// Доступа к L1.c или L1.f0() теперь нет
          void fn() {
            Level0 L0;
           Level1b L1;
            L0.c = 1;
             L0.f0();
L1.f = 1;
             L1.f1();

              }
```

Производный класс может изменять доступность компонентов базового класса. Однако производный класс не может сам обеспечить себе доступ к компоненту, который ему недоступен из-за того, что базовый класс образован как private, например:

```
// конкретно объявляет переменную c как public
      class Level1d: private Level0 {
                public:
 Level0::c;
                 int f;
               void f1();
                 };

// Общая функция
// Доступ к c теперь возможен, но f0 остается недоступной
          void fn() {
            Level0 L0;
           Level1d L1;
            L0.c = 1;
             L0.f0();
L1.c = 1;
             L1.f = 2;
             L1.f1();
              }
```

При объявлении **Level1d** как **private-производного**, умолчание для доступности переменной **c** изменяется с **public** на **private**. Однако, объявив специальным образом переменную **c** как **public**, умолчание можно переопределить, делая **L1.c** доступной из обычной функции **fn()**. **Level1d** не может обеспечить сам себе доступ к компоненту a, который является частным (**private**) в базовом классе.

1.3. Конструкторы производных классов

Для некоторых производных классов требуются конструкторы. Если у базового класса есть конструктор, он должен вызываться при объявлении объекта, и если у этого конструктора есть параметры, их необходимо предоставить.

Параметры конструктора базового класса указываются в определении конструктора производного класса. Вызов конструктора базового класса следует непосредственно после имени конструктора производного класса, перед открывающей фигурной скобкой.

```
class Level0 {
  private: int a;
protected: int b;
  public: int c;
    void f0();
 Level0(int v0) {
    a = b = c = v0;
        }
      };

class Level1: public Level0 {
         private: int d;
       protected: int e;
         public: int f;
           void f1();
Level1(int v0, int v1): Level0(v0) {
          d = e = f = v1;
              }
           };

       // Общая функция
          void fn() {
           Level0 L0(1);
          Level1 L1(1,2);
              }
```

Конструктор производного класса может инициализировать **protected-** и **public-компоненты** базового класса, не выполняя вызова конструктора. C++ вызывает конструктор по умолчанию базового класса, если этого не делает сам конструктор производного класса.

Следующий фрагмент программы даст тот же результат, что и предыдущее определение конструктора.

```
// по умолчанию - Level(v0)
Level1(int v0, int v1): (v0) {

        d = e = f = v1;
             }
```

Конструкторы объемлемых (см. следующий параграф) классов можно вызывать в той же строке, что и конструктор базового класса. Следующий конструктор **Level1** эквивалентен двум предыдущим:

```
Level1(int v0, int v1): Level(v0),d(v1),e(v1),f(v1) { }
```

1.4. Объемлющие классы

Сравним производные классы с классом **Level1**, который объемлет, а не наследует объект класса **Level0**. Для таких классов используют название "**объемлющие классы**", например:

```
class Level1 {
public: Level0 L0;
  private: int d;
 protected: int e;
      public:
        int f;
      void f1();
      };

// Непривелегированная функция
          void fn() {
            Level1 L1;
           L1.L0.c =1;
            L1.f = 2;
           L1.L0.f0();
            L1.f1();
             }

       // Компонентная функция
     void Level1::f1() {
```

```
                                        L0.c = 1;
                                    d = 2; e = 3; f = 4;
                                        L0.f0();
                                         }
```

Доступность компонентов производного и объемлющего классов аналогична. **Level0::a** недоступен для компонентов класса **Level1**, а **Level0::c** доступен. Защищенный (protected) компонент **Level0::b** не доступен для более объемлющего класса.

Основное различие между объемлющим и производным классами состоит в способе доступа к наследуемым элементам. Всякий раз при доступе к элементу **Level0** он задается конкретно, например **L0.c**, **L0.f0()** и т.д. Производный же класс ссылается к этим компонентам как к собственным.

Производный класс использует компоненты своего базового класса, в то время как объемлющий класс просто предоставляет место компонентам другого класса.

1.5. Примеры связных списков

Класс связного списка [3] является довольно популярным базовым классом, на котором построено множество других классов. Рассмотрим реализацию класса список.

```
                          //--------------------------------------
// Программа для обработки объектов классов "список", "двусвязный список", "закольцованный список"
                          //--------------------------------------

                                  #include <stdio.h>
                                  #include <stdlib.h>
                                  #include <string.h>
                                 #include <iostream.h>
                                  #include <alloc.h>
                                  #include <conio.h>
                                    #define IMAX 4

                                   // Класс "Список"
                        class List{
                                        protected:
                                  // Значение элемента списка
                                         float Value;
                               // Ссылка на следующий элемент списка
                                         List *Next;
                                         public:
                                    void AddElList(char);
                                     void OutpList(char);
                                     void DelElList(int);
                                  void AddElList(float, char);
                                       void CreateList();
                                    // Конструктор класса
                              List(const char *Ident){
                                        //Запрос на ввод значения
                                 cout <<"Lead the value of the first ";
                                    cout <<"element of list "<<Ident<<'\n';
                                          // Чтение первого элемента
                                               cin >> Value;
                                         //1-й элемент ссылается на NULL
                                    Next = NULL;
                                                   }
                                   // Конструктор без параметров
                              List(){
                                     // Чтение значения нов. элемента
                               Value = 0;
                                      // Новый элемент ссылается на NULL
                                                Next = NULL;
                                                   }
                                    // Деструктор класса
                                                ~List();
                                               };

                              //--------------------------------------
                               // Реализация деструктора класса List
                              //--------------------------------------
```

```
List::~List() {
// Текущий элемент списка
List *CurrEl,
// Следующий элемент
*TNext;
// Первый элемент - Объект
CurrEl = this;
while ((CurrEl->Next!= NULL) && (CurrEl->Next!= this)){
// Сохраним адрес следующего элемента
TNext = CurrEl->Next;
free(CurrEl);
// Следующий элемент сделать текущим
CurrEl = TNext;
};
// Удалить последний элемент
free(CurrEl);
cout << "Object deleted" <<'\n';
getch();
}

//-------------------------------------
// Функция добавления элемента в конец односвязного списка
//-------------------------------------

void List::AddElList(char R){
// Текущий элемент списка
List *CurrEl,
// Новый элемент списка
// Выделение памяти под новый элемент
*NewEl = new List;

// Текущий элемент - Объект
CurrEl = this;
List* KeyWord;
KeyWord = R ? this: NULL;
// Переход в конец списка
while (CurrEl->Next!= KeyWord){
CurrEl = CurrEl->Next;
}
cout << "Lead the value of new element of list ";
// Ввод значения нового элемента
cin >> NewEl->Value;
// Новый элемент ссылается на NULL
NewEl->Next = KeyWord;
// Новый элемент - в конец списка
CurrEl->Next = NewEl;
}

//-------------------------------------
// Функция вывода на экран односвязного списка
//-------------------------------------

void List::OutpList(char R){
// Счетчик элементов списка
int Count = 1;
// Текущий элемент списка
List *CurrEl;
// Текущий элемент - Объект
CurrEl = this;
void* KeyWord;
KeyWord = R ? this: NULL;
while (CurrEl!= KeyWord){
// Вывод элемента списка
cout << Count << "-th element of list = ";
cout << CurrEl->Value << '\n';
CurrEl = CurrEl->Next;
Count++;
}
}
```

```
//-------------------------------------
// Функция удаления i-го элемента списка
//-------------------------------------

     void List::DelElList(int i){
       // Счетчик элементов списка
            int Count = 1;
        // Текущий элемент списка
            List *CurrEl,
          // Предыдущий элемент
               *PrevEl;
       // Текущий элемент - Объект
            CurrEl = this;
        // Преход к i-му элементу
           while (Count < i){
      // Сохранение предыдущий элемента
            PrevEl = CurrEl;
          CurrEl = CurrEl->Next;
                Count++;
                 }
  // предыдущий элемент ссылается на след.
         PrevEl->Next = CurrEl->Next;
                free(CurrEl);
                    }

//-------------------------------------
// Функция добавления элемента в конец списка с заданием элемента из программы
//-------------------------------------

               void List::AddElList(float Val, char R){
                            List *CurrEl,
                           *NewEl = new List;
                       // Текущий элемент - Объект
                            CurrEl = this;
                            List* KeyWord;
                        KeyWord = R ? this: NULL;
                        // Переход в конец списка
                     while (CurrEl->Next!= KeyWord){
                         CurrEl = CurrEl->Next;
                                  }
                    // Новый элемент - в конец списка
                          CurrEl->Next = NewEl;
                     // Ввод значения нового элемента
                           NewEl->Value = Val;
                    // Новый элемент ссылается на NULL
                          NewEl->Next = KeyWord;
                                  }

//-------------------------------------
// Функция создания списка (ввод первого элемента списка, созд. конструктором без параметров)
//-------------------------------------

                       void List::CreateList(){
                              List *CurrEl;
                                 char ch;
                                int Ok = 0;
                        // Текущий элемент - Объект
                              CurrEl = this;
                                    do
                          if ((Value == 0)||(Ok == 1)){
                              // Запрос на ввод значения
                     cout << "Lead the value of the first ";
                      cout << "element of new list "<<'\n';
                              cin >> CurrEl->Value;
                                     break;
                                    }else{
                        cout << "This List already exists.";
                    cout << "Do you want to delete it?(Y/N)";
                                 cin >> ch;
                        if ((ch == 'N')||(ch == 'n')) break;
                    else if ((ch == 'Y')||(ch == 'y')) Ok = 1;
```

```
                                else cout << "Input Error";
                                          }
                                    while (1);
                                       }

                          //-------------------------------------
                          // Производный класс: двусвязный список
                          //-------------------------------------

                                 class DLList: public List{
                              // Адрес предыдущий элемента списка
                                          List *Prev;
                                            public:
                                   DLList(): List(){
                                        Prev = NULL;
                                           }
                                   void AddElList();
                                  void DelElList(int);
                                 void AddElList(float);
                                         };

                          //-------------------------------------
                  // Функция добавления элемента в конец двусвязного списка
                          //-------------------------------------
                                 void DLList::AddElList(){
                                   // Текущий элемент списка
                                        DLList *CurrEl,
                                    // Новый элемент списка
                               // Выделение памяти под новый элемент
                                        *NewEl = new DLList;
                                  // Текущий элемент - Объект
                                           CurrEl = this;
                                  // Переход в конец списка
                  while (CurrEl->Next!= NULL){ CurrEl = (DLList*) CurrEl->Next; }
                         cout << "Lead the value of new element of list ";
                                  // Ввод знач-я нового элемента
                                        cin >> NewEl->Value;
                                // Новый элемент ссылается на NULL
                                        NewEl->Next = NULL;
                                 // Новый элемент - в конец списка
                                       CurrEl->Next = NewEl;
                              // Новый элемент ссылается на предыдущий
                                        NewEl->Prev = CurrEl;
                                           }

                          //-------------------------------------
                     // Функция удаления i-го элемента двусвязного списка
                          //-------------------------------------

                                    void DLList::DelElList(int i){
                                       // Счетчик элементов списка
                                             int Count = 1;
                                        // Текущий элемент списка
                                           DLList *CurrEl,
                                      // Предыдущий элемент
                                             *PrevEl;
                                    // Текущий элемент - Объект
                                             CurrEl = this;
                                     // Преход к i-му элементу
                                         while (Count < i){
                                  // Сохранение предыдущий элемента
                                           PrevEl = CurrEl;
                                  CurrEl = (DLList*) CurrEl->Next;
                                              Count++;
                                            }
                              // Предыдущий элемент ссылается на следующий
                                 PrevEl->Next = (DLList*) CurrEl->Next;
                                    PrevEl = (DLList*) PrevEl->Next;
                              // Следующий элемент ссылается на предыдущий
                                      PrevEl->Prev = CurrEl->Prev;
                                             free(CurrEl);
```

```
                                    }

                    //-------------------------------------
                  // Функция добавления элемента в конец списка
                    //-------------------------------------

                       void DLList::AddElList(float Val){
                                  DLList *CurrEl,
                                    *NewEl = new DLList;
                                 // Текущий элемент - Объект
                                        CurrEl = this;
                                  // Переход в конец списка
                                 while (CurrEl->Next!= NULL){
                                 CurrEl = (DLList*) CurrEl->Next;
                                              }
                                // Новый элемент - в конец списка
                                      CurrEl->Next = NewEl;
                                 // Ввод значения нового элемента
                                      NewEl->Value = Val;
                                // Новый элемент ссылается на NULL
                                      NewEl->Next = NULL;
                                              }

                    //-------------------------------------
                   // Производный класс: закольцованный список
                    //-------------------------------------

                           class RLList: public List{
                                      public:
                                       RLList(){
                                         Value = 0;
                                        Next = this;
                                            }
                                       };

                    //-------------------------------------
          // Главная функция программы тестирует работоспособность Класса «Список»
                    //-------------------------------------

                        int main(int argc, char **argv){
                                    List TestL;
                                    int Number;
                                 // Вспомогательная переменная
                                        char ch = 'Y';
                                   char Key = ' ', *PKey;

                                        // Приветствие
                              cout << "Hellow! You have ran the program of";
                              cout << " processing Lists just now." << '\n';
                                    cout << "First part:" << '\n';
                               cout << "Please, enter you choose:" << '\n';
                                        PKey = &Key;

                                       // Главное меню
                                            do{
                                    cout << " 1 - New List" << '\n';
                             cout << " 2 - Adding Element to List" << '\n';
                              cout << " 3 - Deleting Element of List" << '\n';
                               cout << " 4 - Output List to screen" << '\n';
                                      cout << " 5 - Exit" << '\n';
                                             Key = getch();
                                             switch (Key){
                                      case '1': TestL.CreateList();
                                                      break;
                                      case '2': TestL.AddElList(0);
                                                      break;
                             case '3': cout << "Enter the number of element";
                                     cout << " you want to delete" << '\n';
                                                cin >> Number;
                                           TestL.DelElList(Number);
                                                     break;
```

```
                    case '4': TestL.OutpList(0);
                                  break;
                           case '5': break;
                       default: cout << "Input Error";
                                  }
                      fread(PKey,1,1,stdin);
                      if (Key == '5') break;
                            clrscr();
                              }

                          while (1);
                           clrscr();
                 cout << "Second part:" << '\n';
                       // Объект - список
                         List L1("L1");
                              do{
                    if ((ch == 'Y')||(ch == 'y'))
                          // Добавление элемента
                             L1.AddElList(0);
                                else
                         // Нажата не та клавиша
                       cout << "Input error" << '\n';
                     // Запрос на ввод след. элемента
                     cout << "Do you want to add one";
                cout << " more element?(Y/N)" << '\n';
                             cin >> ch;
                           // Выход из цикла
                    if ((ch == 'N')||(ch == 'n'))
                                break;
                              }
                       // Бесконечный цикл
                            while (1);
                        L1.AddElList(12, 0);
                       // Вывод сп-ка на экран
                           L1.OutpList(0);
                              getch();
                             clrscr();
                  cout << "Third part:" << '\n';
                             List L2;
                              int i;
                          L2.CreateList();
                    for(i = 0; i <= IMAX; i++){
                     L2.AddElList((float) i+1, 0);
                              }
                     // Вывод сп-ка на экран
                           L2.OutpList(0);
                              getch();
                             return 0;
                               }

                //-------------------------------------
```

В приведенном примере определяется базовый класс односвязного списка List в соответствии с концепцией полны класса содержащий все основные функции работы со списками:

void AddElList(char) - добавление элемента,
void OutpList(char) - вывод содержимого мписка на экран,
void DelElList(int) - удаление элемента,
void CreateList() - первоначальное создание списка.

Как следует из теста программы, класс двусвязного списка **DLList** является производным классом от **List**. Такая реализация целесообразна вследствие того, что большинство методов односвязного списка полностью по программному коду совпадают с методами двусвязного, хотя обработка двусвязного списка должна быть дополнена рассмотрением обратных ссылок на элементы. Аналогичным образом реализуется и класс закольцованного списка **RLList**, по тем же соображениям являющийся производным от класса **List**.

Спецификатор доступа protected позволяет расширить область видимости полей-данных класса **List** так, чтобы их можно было использовать в классах, производных от класса **List** - **DLList** и **RLList**. Функции-члены класса **List** описаны с использованием спецификатора **protected**, что позволяет

использовать их имена как при определении производных классов, так и во всех остальных функциях программы. Ссылка на предыдущий элемент списка в описании класса **DLList** по умолчанию описана как **privat**, поскольку данный класс не является базовым для других классов.

Для всех классов, описанных в данном примере, используется один и тот же деструктор. Он наследуется производными классами как обычная функция-член, и следовательно будет работать правильно с любыми их объектами, несмотря на различия по числу и размеру полей.

Функция **main()** в примере предназначена лишь для проверки работоспособности определенных в программе классов и поэтому содержит описание необходимого объекта и меню для его тестирования. При желании число объектов в основной части программы можно пополнить.

Заключение

Объекты в программе работают как слаженный коллектив вполне самостоятельных программ, которые сами знают, когда им в зависимости от текущей ситуации нужно начать работу, когда ее временно приостановить, и наконец, когда совсем покинуть коллектив программ, не оставив о себе никакого воспоминания кроме необходимых полезных результатов работы. Как правило, каждый объект, начиная свою работу, заказывает у операционной системы оперативную память под свои данные, а заканчивая работу, возвращает эту память назад системе. Тем самым оптимизируется объем памяти, занимаемый всей программой в процессе ее работы.

Для того, чтобы объекты четко знали свое место и полномочия в едином коллективе, и не выполняли одну и ту же работу, они подвергаются специальной классификации, результатом которой является выделение классов объектов. Если два класса обладают общими свойствами, значит для них должен существовать более общий класс, который имеет только те свойства, которые для этих двух объектов являются общими. В этом случае объектам классов с общими свойствами нужно заботиться только о выполнении своих функций, связанных с их различающимися свойствами. Общую же часть может выполнить объект более общего класса.

Современные языки программирования, такие как C++, предоставляют в распоряжение программиста обширный арсенал инструментальных средств, позволяющий проектировать мощные и гибкие программы, но для того, что бы приступить к их составлению, необходимо владеть системой элементов объектно-ориентированной технологии. Этой системе элементов для языка C++ и посвящен настоящий раздел книги.

Индивидуальное задание

Пересмотреть проект, разработанный при выполнении лабораторной работы №1 (№8):

1. Разработать и программно реализовать иерархию классов.
2. Обосновать целесообразность или нецелесообразность использования объемлющих классов.

[В начало]