Программирование и алгоритмизация

Лекция 3

Введение в объектно-ориентированное программирование на C++

Идея

- Структурировать данные:
  - дать возможность независимо разрабатывать их фрагменты и программы, связанные с их обработкой
  - Легко создавать иерархические описания понятий предметной области с наследованием свойств в соответствии с иерархией
  - Идея взята из концепции фреймов в ИИ

Основные понятия

- Объект -- структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии
- Класс - описание категории таких объектов, их структуры и действий, которые могут выполняться над ними
- В классе описываются свойства и методы (программы) данной категории объектов

- В C++ класс обладает синтаксическими свойствами базового типа данных:
  - класс определяется как структурированный тип данных **(struct**);
  - объекты определяются как переменные класса;
  - возможно переопределение и использование стандартных операций языка, имеющих в качестве операндов объекты класса, в виде особых методов в этом классе.

```
Struct        matrix
{  //  определение структурированного типа
   // matrix и методов,
    //  реализующих операции  над matrix * matrix,
    // matrix * double
};
matrix        a,b;            // Определение
                              //        переменных -
double        dd;             // объектов класса matrix
a = a * b;                 // Использование
                              // переопределенных операций
b = b * dd * 5.0;
```

- Проблема "Что первично -курица или яйцо?" применительно к программированию звучит как "Что первично: алгоритм (процедура, функция) или обрабатываемые им данные".
- В традиционной технологии программирования взаимоотношения процедуры -данные имеют более-менее свободный характер, причем процедуры (функции) являются ведущими в этой связке: как правило, функция вызывает функцию, передавая данные друг другу по цепочке. Соответственно, технология структурного проектирования программ прежде всего уделяет внимание разработке алгоритма
- В технологии ООП взаимоотношения данных и алгоритма имеют более регулярный характер:
  - класс объединяет в себе данные (структурированная переменная) и методы (функции).
  - схема взаимодействия функций и данных принципиально иная. Метод (функция), вызываемый для одного объекта, как правило, не вызывает другую функцию непосредственно. Для начала он должен иметь доступ к другому объекту (создать, получить указатель, использовать внутренний объект в текущем и т.д.), после чего он уже может вызвать для него один из известных методов.

  Таким образом, структура программы определяется взаимодействием объектов различных классов между собой. Как правило, имеет место иерархия классов, а технология ООП иначе может быть названа как программирование "от класса к классу".

- Эпизодическое объектно-ориентированное программирование
  - Эпизодическое использование технологии ООП заключается в разработке отдельных, не связанных между собой классов и использовании их как необходимых программисту базовых типов данных, отсутствующих в языке. При этом общая структура программы остается традиционной. ("от функции к функции"). Например, для работы с матрицами программист может определить класс матриц, переопределить для него стандартные арифметические операции и использовать переменные типа "матрица" в обычной Си-программе.
- Тотальное программирование "от класса к классу"
  - Строгое следование технологии ООП предполагает, что любая функция в программе представляет собой метод для объекта некоторого класса. Это не означает, что нужно вводить в программу какие попало классы ради того, чтобы написать необходимые для работы функции. Наоборот, класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых физических предметов или абстрактных понятий (объектов программирования). С другой стороны, каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих. В конце концов вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить). Именно этот переход (а не понятия класса и объекта, как таковые) создает психологический барьер перед программистом, осваивающим технологию ООП.

- Программирование "от класса к классу" включает в себя ряд новых понятий. Прежде всего, это - **НАСЛЕДОВАНИЕ**. Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки

- Вторым по значимости понятием является **ПОЛИМОРФИЗМ**. Он основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится "самодостаточным". Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о его типе, он всегда может корректно вызвать свойственные ему методы. Полиморфной называется функция, определенная в нескольких производных классах и имеющая в них общее имя. Точнее сказать, что полиморфная функция, это группа функций, которая выступает под одним и тем же именем, но в разных классах. Полиморфная функция обладает тем свойством, что при отсутствии полной информации о том, объект какого из производных классов в данный момент обрабатывается, она тем не менее корректно вызывается в том виде, который соответствует именно объекту этого класса (Здесь уместен образный термин “многоликая функция”). Практический смысл полиморфизма заключается в том, что программист может сделать регулярным процесс обработки несовместимых объектов различных типов при наличии у них такого полиморфного метода (в Си++ **-виртуальной функции).**

Отличия C++ от классического C

- структура (**struct**) получила ряд свойств базового типа данных;
- введено понятие элемента-функции. Элементы-функции играют роль своеобразного интерфейса для использования структурированной переменной;
- расширены возможности транслятора по контролю и преобразованию параметров при вызове функции (ссылка, перегрузка функций, параметры по умолчанию). Введен обязательный контроль параметров функций с использованием прототипов.

Операции над структурированными переменными

- операция ПРИСВАИВАНИЯ СТРУКТУРИРОВАННЫХ ПЕРЕМЕННЫХ производит побайтное копирование одной структуры в другую. Присваивание возможно также при косвенном обращении по указателю на структуру;
- ФОРМАЛЬНЫЙ ПАРАМЕТР – СТРУКТУРИРОВАННАЯ ПЕРЕМЕННАЯ: имеет место способ передачи параметра по значению: в стеке резервируется место для размещения структуры -формального параметра и производится присваивание ей значения фактического параметра (копирование);
- РЕЗУЛЬТАТ ФУНКЦИИ – СТРУКТУРИРОВАННАЯ ПЕРЕМЕННАЯ: при выполнении оператора **return** в такой функции значение операнда - структуры присваивается структурированной переменной, использующей результат функции. При отсутствии непосредственного присваивания результата транслятор создает неявную автоматическую структурированную переменную , в которой временно его сохраняет. Функция, возвращающая структуру в качестве результата, может иметь неявный параметр - адрес размещения результата (указатель);
- в обозначении типа данных -структуры служебное слово **struct** можно опускать.

Функции - элементы структуры

```
struct dat
        {
        int day,month,year;
        int     TestData();                 // Проверка даты
        void    NextData();                 // Добавить 1 день
        void    PlusData(int n)             // Добавить n дней
                {
                while(n-- !=0) NextData();
                }
        };
static  int mm[] =
       {31,28,31,30,31,30,31,31,30,31,30,31};
//----------- Проверка  на корректность -----------------
int     dat::TestData()
{
if (month ==2 && day==29 && year %4 ==0) return(1);
if (month ==0 || month >12 || day ==0 || day
       >mm[month])
        return(0);
return(1);
}
```

```
//----------- Следующая дата ----------------------
void    dat::NextData()
{
day++;
if (day <= mm[month]) return;
if (month ==2 && day==29 && year %4 ==0)
      return;
day=1;
month++;
if (month !=13) return;
month=1;
year++;
}

//--------- Основная программа -----------------
void    main()
{
dat a;
do cin << a.day << a.month << a.year;
while(a.TestData() ==0);
a.PlusData(17);
}
```

Функции -элементы структуры (2)

Как видно из примера, в качестве элементов структуры могут выступать функции. Такие элементы-функции имеют следующие особенности:

- тело функции может быть определено в самой структуре (функция **PlusData**). В этом случае функция имеет стандартный вид;
- в определении структуры дается только прототип функции (заголовок с перечислением типов формальных параметров). Определение самой функции дается отдельно, при этом полное имя функции имеет вид

  имя_структуры::имя_функции
- в теле функции неявно определен один формальный параметр с именем **this** - указатель на структуру, для которой вызывается функция (В нашем примере это будет **struct dat *this**). Элементы этой структуры доступны через явное использование этого указателя или неявно:

```
this->month = 5;
        this->day++;
        month = 5;
        day++;
```

- для структурированной переменной вызов функции - элемента этой структуры имеет вид:

  . имя_переменной.имя_функции (список_параметров)

Перегрузка функций

В Си++ возможно определение нескольких функций с одинаковым именем, но с разными типами формальных параметров и результата. При этом транслятор выбирает соответствующую функцию по типу фактических параметров. Переопределяемую функцию необходимо объявить с ключевым словом overload:

```
overload       SetDat;
void   SetDat(int dd,int mm,int yy,dat *p)
       {                              // Дата вводится в виде трех целых
       p->day=dd;
       p->month=mm;
       p->year=yy;
       }
void   SetDat(char *s,dat *p)
       {                                        // Дата вводится в виде строки
       sscanf(s,"%d%d%d", &p->day, &p->month, &p->year);
       }
void   main()
       {
       dat    a,b;
       SetDat(12, 12, 1990, &a);          // Вызов первой функции
       SetDat("12,12,1990", &b);          // Вызов второй функции
       }
```

Перегрузка функций (2)

Функции-элементы структуры также могут быть переопределены, при этом явного объявления не требуется:

```
#include <stdio.h>
struct  dat
        {
        int     day,month,year;
        void    SetDat(int,int,int);
        void    SetDat(char *);
        };
void    dat::SetDat(int dd,int mm,int yy)
        {
        day=dd; month=mm; year=yy;
        }
void    dat::SetDat(char *s)
        {
        sscanf(s,"%d%d%d",&day,&month,&year);
        }
void    main()
        {
        dat     a,b;
        a.SetDat(12,12,1990);
        b.SetDat("12,12,1990");
        }
```

Определения класса и объекта через структуру

- В самом простом виде класс определяется в C++ как структура, работа с элементами данных которой возможна только через функции-элементы
- В отличие от структуры класс имеет “приватную” (личную) часть, элементы которой доступны только в функциях-элементах класса, и “публичную” (общую) часть, на элементы которой ограничения доступа не накладываются
- Объектом называется переменная, типом которой является класс (структура)

```
struct  dat                                                    class   dat
//-------------------------------------------------------------------------------------------------------
      ---
// Определение структуры                                       Определение класса
//-------------------------------------------------------------------------------------------------------
      ----
{                                                             {
                                                               // Личная часть класса
int   day,month,year;                                          int     day,month,year;
                                                                 public:
                                                                  // Общая часть класса
void  SetDat(int,int,int);                                    void SetDat(int,int,int);
void  SetDat(char *);                                         void SetDat(char *);
}                                                             }
void  main()                                                   void    main()
{                                                             {
// Переменные a,b типа dat                                    // Объекты a,b класса dat
dat   a,b;                                                     dat a,b;
a.day = 5;                                                       // Непосредственное обращение
a.month = 12;                                                    // к личной части объекта запрещено
b.SetDat("12,12,1990");                                        b.Setdat("12,12,1990");
}                                                                       }
```

Private и public в описании класса

- Личная часть класса не обязательно должна следовать в начале определения класса. Для обозначения отношения элементов структуры к личной части в произвольном месте определения класса перед ними можно использовать служебное слово **private**
- Стандартным является размещение элементов данных в личной части, а функций-элементов - в общей части класса. Тогда закрытая личная часть определяет данные объекта, а функции-элементы общей части образуют интерфейс объекта "к внешнему миру" (методы).

Private и public в описании класса (2)

- Другие варианты размещения элементов данных и функций-элементов в личной и общей части класса встречаются реже:
  - элемент данных в общей части класса открыт для внешнего использования как любой элемент обычной структуры;
  - функция-элемент личной части класса может быть вызвана только функциями-элементами самого класса и закрыта для внешнего использования

Объекты

- Таким образом, в первом приближении класс отличается от структуры четко определенным интерфейсом доступа к его элементам. И наоборот, структура - это класс без личной части
- Объекты класса обладают всеми свойствами переменных, в том числе такими, как область действия и класс памяти (время жизни). Напомним, что по классам памяти в Си различают следующие виды переменных:
  - Статические и внешние, создаваемые в статической памяти программы и существующие в течение всего времени работы программы;
  - автоматические, создаваемые в стеке в момент вызова функции и уничтожаемые при ее завершении;
  - динамические, создаваемые и уничтожаемые в свободной памяти задачи в моменты вызова функций **malloc и free** или выполнения операторов **new и delete**.

Объекты (2). Пример

```
class dat
        { ..... }
dat     a,b;                // Статические объекты
dat     *p;                 // Указатель на объект
void    main()
{
dat     c,d;                // Автоматические объекты
p = new dat;                // Динамический объект
delete p;                   // Уничтожение динамического объекта
}                   // Уничтожение автоматических объектов
```

Конструкторы и деструкторы

- Процесс создания и уничтожения объектов класса обычно сопровождается некоторыми действиями (инициализация данных, резервирование памяти, ресурсов и т.д.), которые производятся функциями-элементами специального вида. Элементы-функции, неявно вызываемые при создании и уничтожении объектов класса называются **КОНСТРУКТОРАМИ** и **ДЕСТРУКТОРАМИ**
- Они определяются как элементы-функции с именами, совпадающими с именем класса. Конструкторов для данного класса может быть сколь угодно много, если они отличаются формальными параметрами, деструктор же всегда один и имеет имя, предваренное символом **"~".**

Конструкторы и деструкторы (2)

- С процессом создания объектов связано понятие их инициализации. Инициализировать объекты обычным способом нельзя. Их инициализация осуществляется либо явным присваиванием (копированием) другого объекта, либо неявным вызовом конструктора. Если конструктор имеет формальные параметры, то в определении переменной после ее имени должны присутствовать в скобках значения фактических параметров.
- Момент вызова конструктора и деструктора определяется временем создания и уничтожения объектов:
  - для статических и внешних объектов конструктор вызывается перед входом в **main**, деструктор - после выхода из **main()**. Конструкторы вызываются в порядке определения объектов, деструкторы -в обратном порядке;
  - для автоматических объектов конструктор вызывается при входе в функцию (блок), деструктор - при выходе из него;
  - для динамических объектов конструктор вызывается при выполнении оператора **new**, деструктор - при выполнении оператора **delete**.

Перегрузка операторов

- Переопределение в классе стандартных операторов на специфические для данного класса
- Например + для матриц

Пример

```
class complex {
        double re,im;
public:
    complex(double r=0.,double i=0.) { re=r; im=i; }
    operator double() { return sqrt(re*re+im*im); }
    complex operator +(double b){ return complex(re+b,im); }
    complex operator +(complex a)
          { return complex(a.re+re,a.im+im); }
    void print() const { cout << '(' << re << ',' << im << ')'; } };
void main(void) {complex a(3.,4.),b=a+5,c=a+b;
          cout << "c="; c.print(); cout << endl; }
```

Результат: c=(11,8)

```
void main(void) {complex a(3.,4.),b=5+a,c=a+b;
          cout << "c="; c.print(); cout << endl; }
```

Результат: c=(13,4)

Дружественные функции и классы

- В соответствии с принципом инкапсуляции все поля объекта должны быть объявлены в разделе **private**, а доступ к ним должен осуществляться с помощью специальных функций.
- Но в некоторых случаях требуется сделать исключение из правил и предоставить некоторым функциям или методам класса возможность напрямую обращаться к полям.
- В языке C++ имеется механизм *дружественных функций* и *дружественных классов*, который позволяет ограниченно предоставить прямой доступ к защищенным полям класса, не нарушая при этом принципов инкапсуляции.
- Функция, дружественная некоторому классу, имеет полный доступ ко всем его полям и методам.
- Любой метод класса, дружественного некоторому классу, также имеет полный доступ к его полям и методам.
- При описании класса требуется специальным образом перечислить все дружественные функции и все дружественные классы.
- Для указания дружественных функций и классов используется ключевое слово **friend**.

Пример

```
class complex {
        double re,im;
public:
    complex(double r=0.,double i=0.) { re=r; im=i; }
    operator double() { return sqrt(re*re+im*im); }
    friend complex operator +(double,complex);
    complex operator +(double b){ return complex(re+b,im); }
    complex operator+(complex a)
          { return complex(a.re+re,a.im+im); }
    void print() const { cout << '(' << re << ',' << im << ')'; } };
complex operator+(double a, complex b)
          { return complex(a+b.re,b.im); }
main() { complex a(3.,4.), b=a+5, c=5+a; b.print(); c.print(); }
```

Результат: (8,4)(8,4)

Пример в среде CBuilder

```
#include <iostream.h>
#include <math.h>
#pragma hdrstop
class complex {
    double re,im;
public:
complex(double r=0.,double i=0.)
     { re=r; im=i; }
operator double()
     { return sqrt(re*re+im*im); }
friend complex operator
  +(double,complex);
complex operator
  +(double b){ return complex(re+b,im); }
complex operator  +(complex a)
    { return complex(a.re+re,a.im+im); }
void print() const
 { cout << '(' << re << ',' << im << ')'; } };

complex operator+(double a, complex b)
             { return complex(a+b.re,b.im); }

//---------------------------------------------------
------------------

#pragma argsused
int main(int argc, char* argv[])
{
complex a(3.,4.),
 b=a+5,
 c=5+a;
 b.print();
 c.print();
 return 0;
}
```

Шаблоны классов

- **Шабло́ны** (англ. *template*) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).
- В C++ возможно создание шаблонов функций и классов.
- Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем). Например, нам нужен какой-то класс:

```
class SomeClass{ int SomeValue; int SomeArray[20]; ... }
```

- Для одной конкретной цели мы можем использовать этот класс. Но, вдруг, цель немного изменилась, и нужен еще один класс. Теперь нужно 30 элементов массива SomeArray и вещественный тип SomeValue и элементов SomeArray. Тогда мы можем абстрагироваться от конкретных типов и использовать шаблоны с параметрами. Синтаксис: в начале перед объявлением класса напишем слово template и укажем параметры в угловых скобках. В нашем примере:

```
template < int ArrayLength, typename SomeValueType > class SomeClass{
    SomeValueType SomeValue; SomeValueType SomeArray[ ArrayLength ]; ... }
```

- Тогда для первой модели пишем:

```
SomeClass < 20, int > SomeVariable;
```

- для второй:

```
SomeClass < 30, double > SomeVariable2;
```

Шаблон функции

- Шаблон функции начинается с ключевого слова template, за которым в угловых скобках следует список параметров. Затем следует объявление функции:

```
template< typename T >
void sort( T array[], int size ); // прототип: шаблон sort объявлен, но не определён
template< typename T >
void sort( T array[], int size ) // объявление и определение
 { T t;
 for (int i = 0; i < size - 1; i++)
    for (int j = size - 1; j > i; j--)
       if (array[j] < array[j-1])
          { t = array[j]; array[j] = array[j-1]; array[j-1] = t; } }
 template< int BufferSize > // целочисленный параметр
 char* read()
 { char *Buffer = new char[ BufferSize ]; /* считывание данных */
 return Buffer; }
```

- Можно использовать class вместо typename: template< class T >