Проектиране и Тестиране на Софтуер
ТУ, кат. КС, летен семестър 2012

Лекция 4а

Тема:

ООП - Полиморфизъм

ООП – Конвертиране на данни

Част 1

ООП
Полиморфизъм

Съдържание:

- Ранно свързване и късно свързване
  - Нормални член функции и виртуални член функции
  - Методи, достъпни с указатели
- Предефинирани операции
- Приятелски функции

3-те главни ООП черти

- Капсулиране/Скриване на Данни
- Наследяване
- **Полиморфизъм**
  - Общо: способност за проява в различни форми
  - Конкретно ООП: способност за предефиниране на методи за породени класове (redefine *methods* for *derived classes)*
  - *Polymorphism* се отнася до способността в ПЕ да се третират обекти по различен начин в зависимост от техния тип или клас (process objects differently depending on their data type or class)

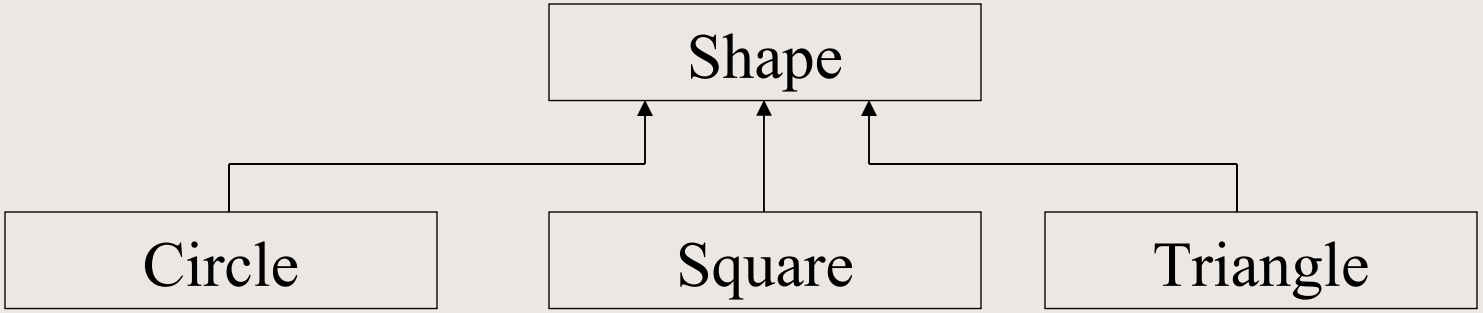
Що е полиморфизъм?

- **Polymorphism** – придава различен смисъл (значение) на една същност
- giving different meanings to the same thing
- Ранно свързване и късно свързване нормални/виртуални методи, достъпни с указатели. Примери:

Base **Derived1, Derived2** класове	**Person** **Professor, Student** класове

Полиморфизъм – 3та ООП х-ка

Пример: базов клас Shape, 3 породени класа



Задача: Да се нарисува сложен образ от 23 кръга, 19 квадрата и 58 триъгълника (общо 100 примитивни фигури) с минимален брой оператори (в идеалния случай с един единствен оператор)

Всички класове имат метод *void Draw()*.

Решението:

```
Shape *ptr[100];
for (I=0; I<=99; I++) ptr[I]->Draw();
```

или

```
I=0; while (I<=99) {  (*ptr[I]).Draw(); I++;  }
```

Полиморфизъм - Python пример1

Полиморфизъм демонстриран в/у базови/основни типове. Например, низове(immutable sequences of characters) и списъци (mutable sequences of elements of any type) имат еднакъв интерфейс за индексиране (indexing interface) и еднакъв интерфейс за търсене (lookup interface).

```
myString = 'Hello world'
myList = [0, 'one', 1, 'two', 3, 'five', 8]

print myString[:4]      # prints Hell
print myList[:4]        # prints [0, 'one', 1, 'two']

print 'e' in myString   # prints True
print 5 in myList       # prints False
```

Полиморфизъм - Python пример2

От областта на ООП – потребителски дефинирани типове

Два подкласа (*Cat* и *Dog*) са породени от супер клас *Animal*. Създават се два обекта от клас *Cat* и един обект от клас *Dog*. Трите обекта се събират в списък ([a, b, c]) и с цикличен обход по елементите на списъка се активира съответен метод *talk*.

Полиморфизъм - Python пример2

```
class Animal:
        def __init__(self, name):
            self.name = name

    class Cat(Animal):
        def talk(self):
            return 'Myau!'

    class Dog(Animal):
        def talk(self):
            return 'Au! Au!'
```

Полиморфизъм - Python пример2

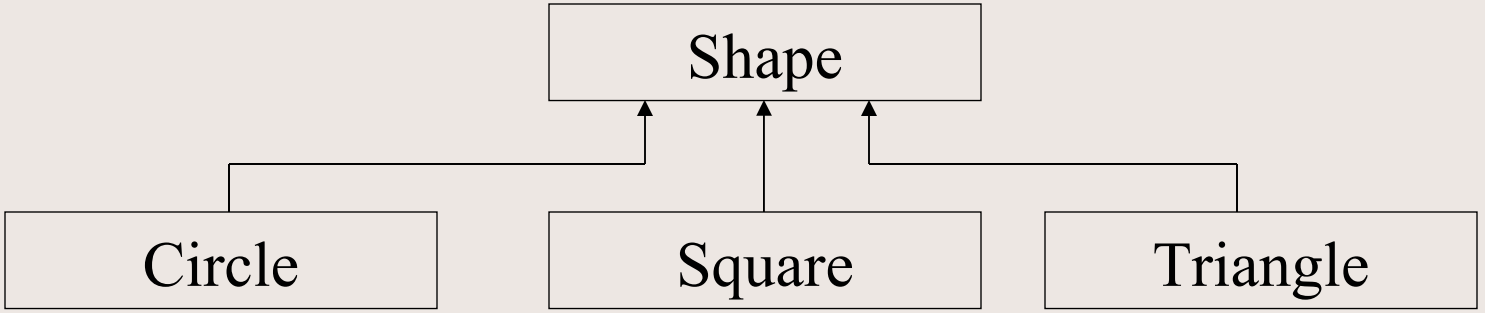
```
a = Cat('Missy')
b = Cat(‘Pissy')
c = Dog(‘Petercho')

for animal in [a, b, c]:
    print animal.name + ': ' + animal.talk()
```

```
# prints the following:
#
# Missy: Myau!
# Pissy: Myau!
# Petercho: Au! Au!
```

Полиморфизъм – 3та ООП х-ка

Пример: базов клас Shape, 3 породени класа



Задача: Да се нарисува сложен образ от 23 кръга, 19 квадрата и 58 триъгълника (общо 100 примитивни фигури) с минимален брой оператори (в идеалния случай с един единствен оператор)

Всички класове имат метод *void Draw()*.

Решението: *Shape *ptr[100];*

for (I=0; I<=99; I++) ptr[I]->Draw();

или

*I=0; while (I<=99) { (*ptr[I]).Draw(); I++; }*

Припомняне: що е Полиморфизъм?

Различен смисъл (значение) на една същност

Giving different meanings to the same thing

Същност (The same thing): операторът за активиране на метод `Draw()`

Различен смисъл (Different meaning): ефектът от изпълнение на метод `Draw()`: рисуване на фигура - различен примитив: кръг, квадрат, триъгълник

Полиморфизъм: условия

Следните две условия:

- Първо, всички видове форми /shapes, such as circles, squares, and triangles/, следва да са породени от един базов клас (like *Shape*).
- Второ, функцията `Draw()` трябва да се декларира като виртуална /`virtual` е запазена дума/ в базовия клас.

Функции достъпни с указатели

Въведение във виртуални функции:

Virtual значи ***existing in effect but not in reality***.

Ранно свързване се проявява при нормални член функции (методи), достъпни с указатели.

Късно свързване се проявява при виртуални член функции (методи), достъпни с указатели.

Ранно свързване / Късно свързване

виртуални с/у нормални методи

Пример 1 за Полиморфизъм: метод *Show()*

```
class Base { . . .  };
class Derived1 : public Base { . . . };
class Derived2 : public Base { . . . };

Derived1 drv1;   Derived2 drv2;    Base *ptr;
ptr = &drv1;       ptr->Show();

ptr = &drv2;       (*ptr).Show();
```

OOP3a.cpp OOP3aEarly.exe OOP3aLate.exe

Ранно свързване (при компилация)

Редовни, не-виртуални методи

```
class Base { public: void Show() {   cout << "\n Base:" ;       } };
class Derived1 : public Base { public: void Show() {
                                        cout << "\n Derived1:" ; } };
class Derived2 : public Base { public: void Show() {
                                        cout << "\n Derived2:" ; } };
```

Мажорен фактор: **типът** на указателя *ptr* – обект от клас Base

```
Derived1 drv1;    Derived2 drv2;    Base *ptr;
ptr = &drv1;      ptr->Show();
ptr = &drv2;      (*ptr).Show();
```

OOP3a.cpp OOP3aEarly.exe

Ранно свързване (при компилация)

Компилаторът игнорира *съдържанието* на указателя *ptr* и избира метод /член функция/, която съответства на /matches/ ***типа*** на указателя.

Виж сл. Фигура >>

Ранно свързване (при компилация)

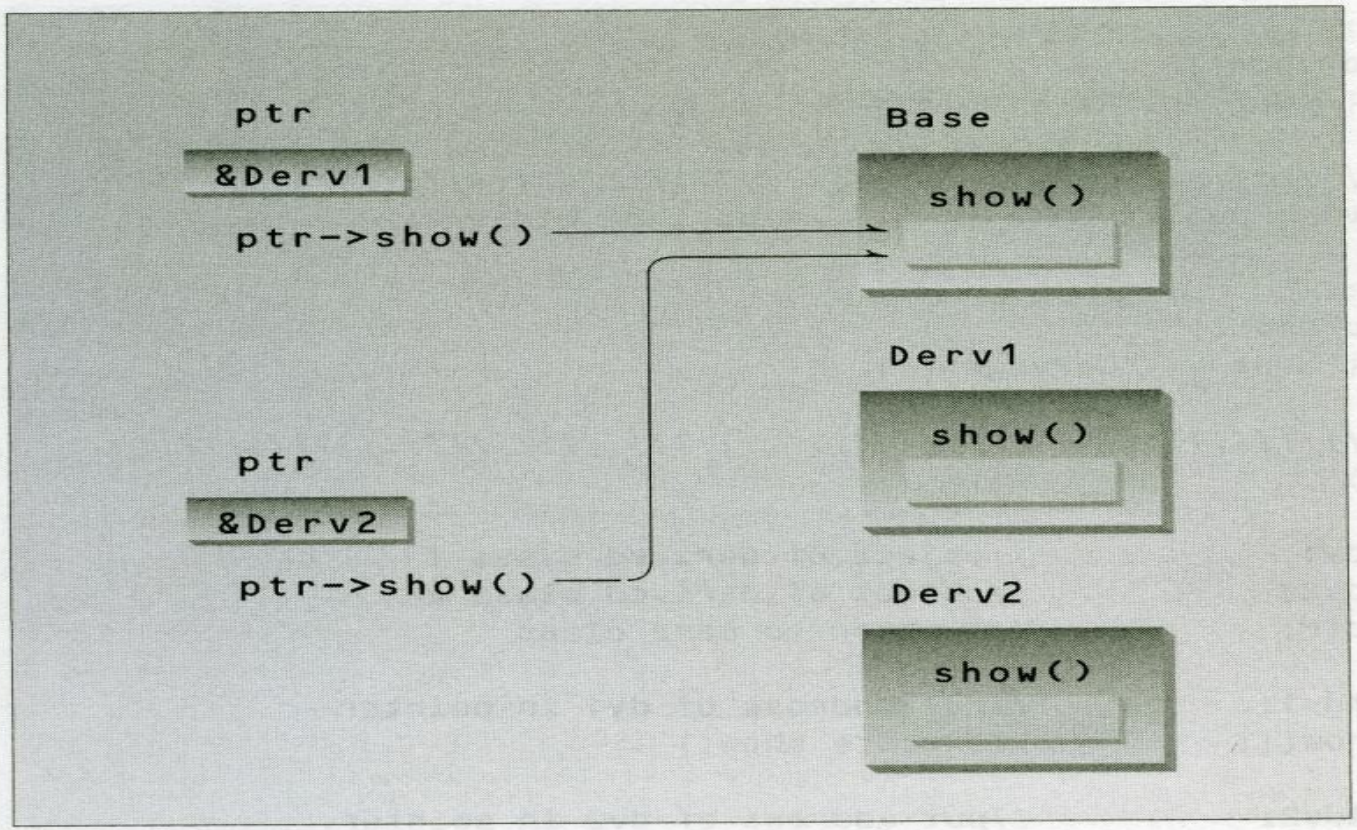


FIGURE 11.1
Nonvirtual pointer access.

Късно свързване (при изпълнение)

Виртуални методи

```
class Base {public: virtual void Show(){ cout << "\n Base:" ;  } };
class Derived1 : public Base {public: void Show(){
                                    cout << "\n Derived1:" ; } };
class Derived2 : public Base {public: void Show(){
                                    cout << "\n Derived2:" ; } };
```

Мажорен фактор: **съдържание** на у-теля *ptr* – обект от породен клас

```
Derived1 drv1;    Derived2 drv2;    Base *ptr;
ptr = &drv1;      ptr->Show();
ptr = &drv2;      (*ptr).Show();
```

OOP3a.cpp OOP3aLate.exe

Късно свързване (при изпълнение)

Компилаторът избира метод /член функция/ според *съдържанието* на указателя *ptr*, а не според ***типа*** на указателя.

Виж сл. Фигура >>

Късно свързване (при изпълнение)

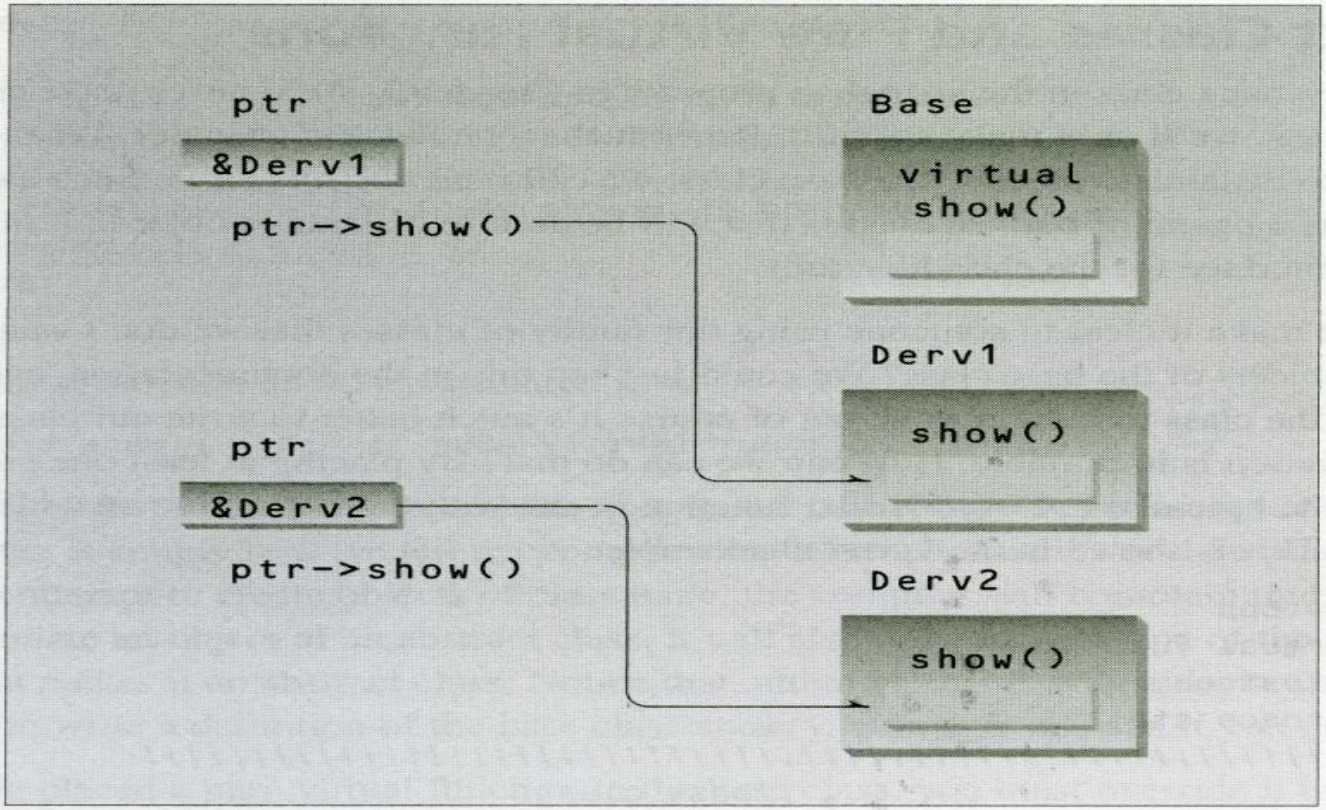


FIGURE 11.2
Virtual pointer access.

Късно свързване (при изпълнение)

Pure virtual methods

Виртуална функция без тяло никога не се изпълнява.

```
class Base { public: virtual void Show()  = 0  ;};

class Derived1 : public Base {
        public: void Show(){ cout << "\n Derived1:" ;}   };

class Derived2 : public Base {
        public: void Show(){ cout << "\n Derived2:" ;}   };
```

Ранно свързване / Късно свързване

виртуални с/у нормални методи

Пример 2 за Полиморфизъм:

метод *isOutstanding()*

```
class Person { . . .  };
class Professor: public Person { . . . };
class Student: public Person { . . . };
```

OOP3b.cpp OOP3b.exe

Виртуален метод *IsOutstanding()*

```
class Person {
 protected:  char name[20];
 public:
            void GetName()
             {
                cout << "\nEnter name:";  cin >> name;
             }
             void ShowName()
             {
               cout << "\n Name is:" << name << "     ";
              }
             bool virtual isOutStanding() = 0;
};
```

Виртуален метод *IsOutstanding()*

```
class Student : public Person
{
 private:   float score;
 public:
            void GetScore()
            {
                    cout << "\n Enter student's score:";  cin >> score;
            }
            bool isOutStanding()
            {
                    return  (score > 98.0) ? true: false;
            }
};
```

Виртуален метод *IsOutstanding()*

```
class Professor : public Person
{
 private:  int NumPubs;
 public:
            void GetNumPubs()
            {
               cout << "\n Enter number of professor's publications:";
               cin >> NumPubs;
            }
            bool isOutStanding()
            {
                return  (NumPubs > 100) ? true: false;
            }
};
```

Виртуален метод *IsOutstanding()*

```
void main ()
{
 Person *PersPtr[100];   Student *StuPtr;   Professor *ProPtr;  int n=0;   char choice;
 do  {
       cout << "\n Enter student or professor (s/p)?:";  cin >> choice;
       if (choice == 's')   {
                StuPtr = new Student;  StuPtr->GetName();  StuPtr->GetScore();
                PersPtr[n++] = StuPtr;
                          }
       else    {
                ProPtr = new Professor;  ProPtr->GetName();  ProPtr->GetNumPubs();
                PersPtr[n++] = ProPtr;
               }
       cout << "\n\n    Enter another  (y/n)?:";  cin >> choice;
     } while (choice == 'y');  // end of do
 for (int j=0; j<n; j++) {
     PersPtr[j]->ShowName();
     if (PersPtr[j]->isOutStanding() == true)              cout << " --outstanding person";
                          } // end of for
 }// end of main()
```

Ранно свързване / Късно свързване

виртуални с/у нормални методи

Пример 2 за Полиморфизъм:

метод *isOutstanding()*

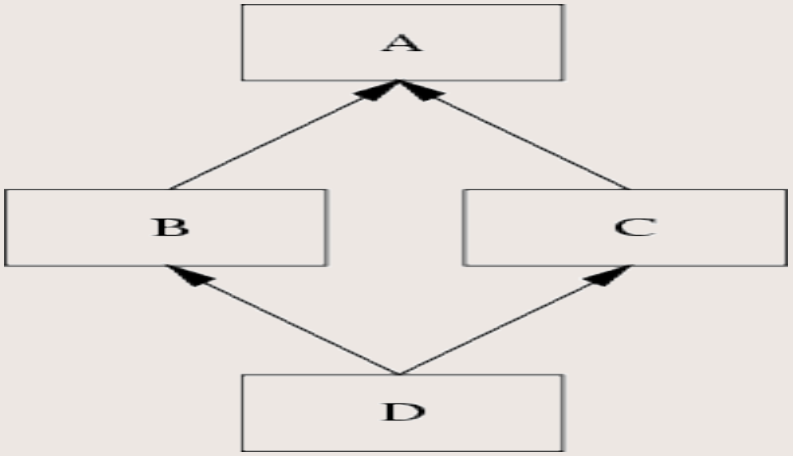
```
class Person { . . .  };
class Professor: public Person { . . . };
class Student: public Person { . . . };
```

OOP3b.cpp OOP3b.exe

Виртуални класове

- Припомняне: Diamond наследяване

Неоднозначност, предизвикана от конфликт на имена: клас *D* множествено наследява от базови класове *B,C,* които са породени от супер базовия клас *A*. Илюстрацията е по-долу.



Виртуални класове

A – клас *Parent*

/ \

B – клас *Child1* C – клас *Child2*

\ /

D – клас *Grandchild*

Проблем: Когато метод от клас D търси достъп до данна(и)/метод(и) от клас A

Виртуални класове

```
class Parent {   protected: int basedata; };
class Child1 : public Parent { } ;
class Child2 : public Parent { } ;
class Grandchild:public Child1,public Child2 {
    public: int getdata() { return basedata; }  // грешка, нееднозн.
};
```

Child1 наследява *basedata,* принадлежащ на *Parent.*
Child2 наследява *basedata,* принадлежащ на *Parent.*

Кой екземпляр *basedata* се наследява от клас *Grandchild*?

Виртуални класове – разрешение на неоднозначност

```
class Parent {   protected: int basedata; };
class Child1 : virtual public Parent { } ;
class Child2 : virtual public Parent { } ;
class Grandchild:public Child1,public Child2 {
    public: int getdata() { return basedata; }    //  няма грешка
};
```

Запазената дума **`virtual`** означава, че двата класа *Child1, Child2* споделят единствено общо копие на под-обект на техния базов клас *Parent*. След като има само един екземпляр на *basedata*, няма неоднозначност и то е достъпно в под-под-класа.

Предефинирани Операции
Overloaded Operators

Въведение

Каква е ползата от предефиниране на операциите?

- По-добра четливост

Примери:

- Клас *Counter*
- Клас *Distance*

Въведение

Пример: Клас **Counter**

```
...
Counter c(100);

  c.IncCount();      ИЛИ      c++;
```

Въведение

Пример: Клас **Distance**

```
...
  Distance d1(5,3.6),d2(6,4.5),d3;
  d3.AddDist1(d1,d2);
  d3 = d1.AddDist2(d2);
  d3 = d1 + d2;          // OK
```

Предефинирани операции като член функции

- Основно правило: Методът, който реализира предефинираната операция, се дефинира с формални параметри, които са с 1 по-малко от броя на операндите на предефинираната операция.
  - Методът за предефиниране на унарна (едноместна) операция няма параметри.
  - Методът за предефиниране на бинарна (двуместна) операция има един формален параметър.
- Правилото не важи за приятелски функции.

Предефиниране на едноместни операции

Предефинирана операция ++

```
class Counter
{
   void IncCount() { count++; }
   void operator++() { count ++; }
};
```

```
Counter c(100);  c.IncCount();  ++c;  c.operator++(); //OK
```

```
Counter d(300), e;          ++d; //OK
                            e = ++d; // NOT OK
```

OOP3c.cpp OOP3c.exe

Counter Предеф. операция ++

```
class Counter
{
 private: unsigned count;
 public:    Counter() { count = 0; }
            Counter(int val ) { count = val; }
            void IncCount() { count++; }
            void GetData() { cout <<"\nEnter data:"; cin >> count;}
            void ShowData() { cout <<"\nData count is:" << count; }
            unsigned GetCount() { return count; }
            // overloaded unary operator ++, first version
            Counter operator++() {      count++; return Counter(count);  }
//   overloaded unary operator ++, second version
//          Counter operator++() {      Counter temp;
//                                      count++;
//                                      temp.count = count;
//                                     return temp; }
};
```

Counter Предеф. операция ++

```
void main()
{
 Counter c1;
 c1.GetData(); c1.ShowData(); c1.IncCount(); c1.ShowData();
 cout << "\n\n" << c1.GetCount();
 ++c1;
 c1.ShowData();

 Counter c2;
 c2 = ++c1;   cout << "\nCounter c2 =" << c2.GetCount();
 c2 = ++c1;   cout << "\nCounter c2 =" << c2.GetCount();

 cout << '\n';
 }
```

Counter предеф. Операция ++

Ограничения върху *Inc/Dec* операции

За да се различават случаите

c2 = c1++;

c2 = ++c1;

са необходими две предефинирани ++ операции

operator++(); // does prefix (++var)

operator++(int) // does postfix (var++)

Предефиниране на двуместни операции

Предефинирана операция +

```
Distance operator+(Distance d2)
{
        int f = feet + d2.feet;  float in = inches + d2.inches;
        if (in >=12.) { f++; in-=12.;      }
        return Distance(f, in);
}

Distance d1(6, 5.18), d2=3.5, d3, d4, d5;
d3 = d1 + d2;          d4 = d1 + 10.0;          d5 = 10.0 + d1;
d3 = d1.operator+(d2);
```

OOP3d.cpp OOP3d.exe

Distance предеф. операции + , +=

```
class Distance { private: int feet; float inches;
 public: Distance() {  feet = 0; inches = 0.0; }
 Distance (int ft, float in) { feet = ft; inches = in; }
 void ShowDist() {  cout <<"\nDistObject= " << feet <<" "<< inches; }
 Distance operator+(Distance d1) {       int ft; float in;
                                         ft = feet + d1.feet;
                                         in = inches + d1.inches;
                                         if (in >= 12.)  { in -= 12.;   ft++;  }
                                         return Distance(ft, in);
                                 }
 Distance operator+=(Distance d1)
                         {               feet = feet + d1.feet;
                                         inches = inches + d1.inches;
                                         if (inches >= 12.) { inches -= 12.;   feet++;   }
                                         return Distance(feet, inches);
                         }
```

Distance предеф. операции + , +=

```
void main ()
{
 Distance d1(5, 6.8), d2(3, 4.5), d5,  d9(1, 1.0), d10;

 d1.ShowDist(); cout << "    Distance d1  ";
 d2.ShowDist(); cout << "    Distance d2  ";

 d5 = d1 + d2;   d5.ShowDist();  cout << "    Distance d5  ";

 d1.ShowDist(); cout << "    Distance d1  ";
 d1 += d9;
 d1.ShowDist(); cout << "    Distance d1  ";

 d10.ShowDist(); cout << "    Distance d10  ";
 d10 = d1 += d9;
 d10.ShowDist(); cout << "    Distance d10  ";
 }
```

Distance предеф. операция за сравнение

```
class Distance { private: int feet; float inches;
 public:    Distance() {  feet = 0; inches = 0.0; }
            Distance (int ft, float in) { feet = ft; inches = in; }
 void ShowDist() {  cout <<"\nDistObject= " << feet <<" "<< inches; }
 bool operator < (Distance) const;
};

 bool Distance::operator < (Distance d2) const
     {
      float bf1 = feet + inches/12;  float bf2 = d2.feet + d2.inches/12;
      return (bf1 < bf2) ? true : false;
     }
};
void  main() { Distance dist1(5, 6.8), dist2(3, 4.5);
          if (dist1<dist2) cout << "\n dist1 object is less than dist2 object";
               }
```

Предефинирани Операции - заключение

- Ползвайте сходни означения и синтаксис
  - Предефинирани операции да се прилагат по начин еднакъв за основни и за потребителски типове.
- Не всички операции могат се предефинират
  - Операция за достъп до елемент - Member access or dot (.) operator
  - Операция за принадлежност - Scope resolution operator (::)
  - Операция условен израз Conditional operator (? :)

Въпрос

Колко операнда /колко местна/ е операцията скоби [ ] ?

Въпрос

Колко операнда /колко местна/ е операцията индексиране ?

Предефинирана операция []

Операция скоби [] или индексиране служи за адресиране на елементи на масив и може да се предефинира.

В C++ няма защита с/у addressing exception.

Ще предефинираме операция скоби, за да създадем “safe” array: автоматично проверява дали стойността на индекса за достъп до елементи на масив е в допустимия диапазон.

Предефинирана операция []

Три примерни програми, всяка с различен подход за запис и четене на елемент от масив:

- Отделни функции *put()* и *get()*
- Единна функция *access()*, която връща стойност чрез позоваване by reference
- Предефинирана операция скоби *[]*, която връща стойност чрез позоваване by reference

Отделни функции *put()* и *get()*

```
class SafeArray { private: int arr[100];
  public:   void putel(int n, int elvalue) {
                        if (n<0 || n>=100) {cout<<“Index out of bounds”; exit(1); }
                        arr[n] = elvalue; }
            int getel(int n) const {
                        if (n<0 || n>=100) {cout<<“Index out of bounds”; exit(1); }
                        return arr[n];      }
};
void main()
{    SafeArray sa1;
     for (int j=0; j<100; j++)  sa1.putel(j, j*10);     // insert array elements
     for (int j=0; j<100; j++)    // display elements
            cout << “Element “<<j<<“ is =“<<sa1.getel(j)<< endl;
}
```

Единна функция *access(),* която връща стойност by reference

Вместо работа с две функции *get* и *put,* възможно е да се въведе единна функция за запис и четене елементи в “safe” масив.

Решението е тази функция да връща псевдоним чрез позоваване by reference. Това означава, че такава функция може да се записва от ляво и от дясно на операция/оператор/ за присвояване.

Припомням: лекция ПП, функции връщат псевдоним /by reference/

Единна функция *access()*, която връща стойност by reference

```
class SafeArray { private: int arr[100];
  public:   int& access(int n) {
                        if (n<0 || n>=100) {cout<<“Index out of bounds”; exit(1); }
                        return arr[n];      }
};
void main()
{    SafeArray sa1;
     for (int j=0; j<100; j++)  sa1.access(j) = j*10; // insert array elements

     for (int j=0; j<100; j++)    // display elements
              { int temp = sa1.access(j);
                 cout << “Element “<<j<<“ is =“<<temp<< endl;
              }
}
```

Предефинирана операция [], която връща стойност by reference

Същият подход се прилага при предефиниране на операция скоби /индексиране/.

Операция скоби/индексиране/ се среща от лявата и дясната страна на операция/оператор/ за присвояване.

Предеф. операция скоби [], която връща стойност by reference

```
class SafeArray { private: int arr[100];
  public:   int& operator[](int n) {
                        if (n<0 || n>=100) {cout<<“Index out of bounds”; exit(1); }
                        return arr[n];      }
};
void main()
{    SafeArray sa1;
     for (int j=0; j<100; j++)  sa1[j] = j*10;          // insert array elements

     for (int j=0; j<100; j++)    // display elements
              { int temp = sa1[j];
                 cout << “Element “<<j<<“ is =“<<temp<< endl;
              }
}
```

Приятелски Функции
Friend Functions

Приятелски функции

- Идея: Да се осигури достъп на функции не-членове на клас до данни на обекти, които са с квалификатор *private* или *protected*.
- Припомняне: Горната идея противоречи на принципа Капсулиране/Скриване на данни
- Има ситуации, когато скриването на данни води до редица неудобства.

Friend функции, приложение

- Правила за работа с приятелски ф-ии:
  - **`friend`** ф-ия се дефинира вън от клас(ове), т.е става дума за глобална/външна деф.
  - **`friend`** ф-ия се декларира must be declared като такава в класа, чийто данни тя ще обработва.
  - Ползва се запазена дума **`friend`**.

Приятелски функции - приложение

- Приятелски функции – мост между класове
- Метод на клас за предефиниране на операция събиране (+) и приятелска функция за предефиниране на операция събиране (+)
- Функционален запис (означение) чрез приятелски функции

Приятелски функции – мост между класове

Да се реализира функция, която борави с обекти от два различни класа.

Решение: Да се състави приятелска функция и за двата класа.

```
class beta;
class alpha { int data; . . .   friend int func(alpha, beta);  };
class beta   { int data; . . .   friend int func(alpha, beta);  };
// friend function
int func( alpha a, beta b) { return a.data + b.data; }

void main() {    alpha aa;   beta bb;
                 cout << func (aa, bb);   }
```

Приятелски функции за предефиниране на операция +

Reminder: binary overloaded operator + as member function

```
Distance operator+(Distance);

Distance Distance::operator+(Distance d)     {
int f = feet + d.feet;  float in = inches + d.inches;
        if (in >=12.) { f++; in-=12.;      }
        return Distance(f,in);
}

Distance d1(6, 5.18), d2=3.5, d3, d4, d5;
d3 = d1 + d2;          d4 = d1 + 10.0;          d5 = 10.0 + d1;
d3 = d1.operator+(d2);
```

OK! **OK!** **NOT OK!**

Приятелски функции за предефиниране на операция +

Comments:

The additions *d3 = d1 + d2;* *d4 = d1 + 10.0;*

d3 = d1.operator+(d2);

will compile. **OK!** **OK!**

Левият контест на операция + е обект от клас Distance и компилаторът знае да активира overloaded + operator

The addition *d5 = 10.0 + d1;* will not compile **NOT OK!**

Левият контест на операция + е основен тип (double) и компилаторът не знае да активира overloaded + operator

Приятелски функции за предефиниране на операция +

Как да разрешим **NOT OK** проблема?

1. Чрез явно създаване на обект от клас Distance. Тогава събирането *d5 = 10.0 + d1;* се променя в >>> d5=Distance(10, 0) + d1; Нужен е 2-арг конструктор и това е неинтуитивно и неелегантно решение.
2. По-доброто елегантно решение се постига с концепцията **`friend`** function.

Приятелски функции за предефиниране на операция +

```
friend Distance operator+(Distance, Distance);

Distance operator+(Distance d1, Distance d2)          {
int f = d1.feet + d2.feet;  float in = d1.inches + d2.inches;
        if (in >=12.) { f++; in-=12.;      }
        return Distance(f,in);
}

Distance d1(6, 5.18), d2=3.5, d3, d4, d5;
d3 = d1 + d2;          d4 = d1 + 10.0;          d5 = 10.0 + d1;
d3 = operator+(d1, d2);
```

OOP3e.cpp OOP3e.exe

Приятелски функции за предефиниране на операция +

```
const float MTF = 3.2808;
class Distance { private: int feet; float inches;
 public:    Distance() { feet = 0; inches = 0.0; }
            Distance (int ft, float in) { feet = ft; inches = in; }
            Distance (float meters) {
      float fltfeet = meters * MTF;  feet = int(fltfeet); inches = 12 *(fltfeet - feet);
                                    }
            void ShowDist()  { cout <<"\nDistObject= " << feet <<" "<< inches;

            friend Distance operator+( Distance, Distance);
};
```

Приятелски функции за предефиниране на операция +

```
Distance operator+( Distance d1, Distance d2)
{
        // first version of source text
        //          int ft; float in;
        //          ft = d1.feet + d2.feet;
        //          in = d1.inches + d2.inches;
        //          if (in >= 12.)  { in -= 12.;   ft++;  }
        //          return Distance(ft, in);

        //  second  version of source text
        Distance temp;
        temp.feet = d1.feet + d2.feet;
        temp.inches = d1.inches + d2.inches;
        if (temp.inches >= 12.)  { temp.inches -= 12.;   temp.feet++;  }
        return temp;
}// end of friend function operator+(Distance d1, Distance d2)
```

Приятелски функции за предефиниране на операция +

```
void main ()
{
 Distance d1(5, 6.8), d2(3, 4.5), d5, d6, d7;

 d1.ShowDist(); cout << "    Distance d1   ";
 d2.ShowDist(); cout << "    Distance d2   ";

 d5 = d1 + d2; d5.ShowDist();  cout << "    Distance d5 = d1 + d2 ";

 d6 = d1 + 10.0;  d6.ShowDist();  cout << "    Distance d6 = d1 + 10.0   ";

 d7 = 10.0 + d1; d7.ShowDist();  cout << "    Distance d7 = 10.0 + d1 ";
}
```

Приятелски функции за предефиниране на операция +

```
friend Distance operator+(Distance, Distance);

Distance operator+(Distance d1, Distance d2)          {
int f = d1.feet + d2.feet;  float in = d1.inches + d2.inches;
        if (in >=12.) { f++; in-=12.;      }
        return Distance(f,in);
}

Distance d1(6, 5.18), d2=3.5, d3, d4, d5;
d3 = d1 + d2;          d4 = d1 + 10.0;          d5 = 10.0 + d1;
d3 = operator+(d1, d2);
```

OOP3e.cpp OOP3e.exe

Приятелски функции – функционален запис

Distance d1(5, 7.8), d2, d3;

d2 = d1.square();

d3 = square(d1);

OOP3f.cpp
OOP3f.exe

OOP3g.cpp
OOP3g.exe

OOP3f.cpp – *square()* като метод

```
class Distance { private: int feet; float inches;
 public: Distance() {              feet = 0; inches = 0.0; }
       Distance (int ft, float in) { feet = ft; inches = in; }
       void ShowDist() {    cout <<"\nDistObject= " << feet <<" "<< inches;  }
 //
 Distance square()  {      float fltfeet = feet + inches/12.;
                           float fltsqrd = fltfeet * fltfeet;
                           int ft = int(fltsqrd);        float in = 12. * (fltsqrd - ft);
                           return Distance(ft, in);
                     } // end of function square( )
};

void  main()
{
 Distance d1(5, 6.8), d2(3, 4.5), d6, d7;

 d1.ShowDist(); cout << "   Distance d1  ";
 d2.ShowDist(); cout << "   Distance d2  ";

 d6 = d1.square();
 d6.ShowDist();  cout << "\t  Distance d6 = d1.square()  ";

 d7 = d2.square();
 d7.ShowDist();  cout << "\t\t  Distance d7 = d2.square()  ";
}
```

OOP3g.cpp–*square()* като приятелска ф-ия

```
class Distance { private: int feet; float inches;
 public: Distance() {              feet = 0; inches = 0.0; }
 Distance (int ft, float in) { feet = ft; inches = in; }
 void ShowDist() {          cout <<"\nDistObject= " << feet <<" "<< inches; }

 friend Distance square(Distance);
};

 Distance square(Distance d) {          float fltfeet = d.feet + d.inches/12.;
                                        float fltsqrd = fltfeet * fltfeet;
                                        int ft = int(fltsqrd);         float in = 12. * (fltsqrd - ft);
                                        return Distance(ft, in);
                                } // end of friend function square(Distance d)
void main ()
{
 Distance d1(5, 6.8), d2(3, 4.5), d6, d7;

 d1.ShowDist(); cout << "    Distance d1  ";     d2.ShowDist(); cout << "    Distance d2  ";

 d6 = square(d1);  d6.ShowDist();  cout << "\t  Distance d6 = square(d1)  ";

 d7 = square(d2);   d7.ShowDist();  cout << "\t  Distance d7 = square(d2)  ";
}
```

Приятелски класове

Всички член функции на един клас се третират като приятелски, ако самият клас се обяви като приятелски.

```
class alpha { private: int data1;
   public: alpha()  { data1 = 99; }
         friend class beta; };
class beta {
   public: void f1(alpha a){ cout << a.data1;};
void main() { alpha a; beta b; b.f1(a); }
```

Част 2

ООП
Конвертиране на Данни

Съдържание:

- Присвояване и конвертиране на данни:
  - Присвояване с еднакви данни
  - Конвертиране м/у основни типове данни
  - Конвертиране м/у обекти и основни типове
  - Конвертиране между обекти от различни класове
- Конструктори:
  - Подразбирани
  - Дефинирани от потребителя
  - Конструктор за копиране и конструктор за присвояване

Благодаря
За
Вниманието