Tillämpad programmering

C++ polymorfism: overload, cast, ärvning, template

Johan Montelius

polymorfism

- Att se och använda en variabel, ett objekt eller en funktion i olika former:
  - overload: olika versioner av samma funktion
  - cast: ändra en variabels typ
  - dispatch: hur hanterar vi objekthierarkier

print


```
void print(int x) {
      cout << “int x = “ << x << endl;
};
```

```
void print(float x) {
      cout << “float x = “ << x << endl;
}
```

overload

- Funktioner med olika typer av argument är olika funktioner:
  - kompilatorn försöker hitta den rätta
  - om flera alternativ finns genereras ett kompileringsfel

Student

```
class Student {
      string name;
      string program;
      int year;
  public:'
      Student(string &nn, string &prgm, int year);
      string getName() const;
      void setName(string &nn);
            :
            :
      void print(ostream &os) const;
}
```

Student

```
void Student::print(ostream &os) {
      os << “Student : “ << name
         << “ “ << program << “/” << year;
}
```

Student

```
Student joe = Student(“joe”, “ICT”, 2011);
    :
joe.print(cout);
    :
```

operator<<

```
ostream &operator<<(ostream &os, Student &ss) {
      ss.print(os);
      return os;
}
```

```
   cout << joe << endl;
```

operatorer

- Är operatorer metoder i en klass eller är de funktioner som är fristående från klasser?
- operator<<
- operator=

Rational

```
class Rational {
      int num;    // numeral - täljare
      int den;    // denominator - nämnare
  public:
      Rational( ..., ...) ... ;

      ... get_num() .... ;

      ... get_den() ... ;

      ... print(...) ... ;
};
```

Rational: +, -, *

```
Rational operator+(
      const Rational lhs,
      const Rational rhs) {
          :
          :
      return Rational(num, den);
}
```

ibland lite konstigt


```
Student joe("Joe", "ICT", 2011);
Student sue("Sue", "ICT", 2012);
      :
      :
cout << joe + sue << endl;
```

cast - ändra en variabels typ

- **static_cast**
  - kontroll vid kompilering
- **dynamic_cast**
  - för pekare eller referenser till objekt, kontroll vid körning
- **const_cast**
  - ta bort const från typen
- **reinterpret_cast**
  - helt utan säkerhetsbälte

cast

- I C var det inte syntaktiskt klart hur långt ut på isen man var.
- Man bör tänka två gånger innan man tar till cast, det kan vara ett tecken på att man tänkt fel från början.

Objekt och ärvning

```
class Person {
        int pnr;
        string name;
      public:
        Person(int pp, const string &nn)
            : pnr(pp), name(nn) {};
        const string &getName() const {
            return name;
        };
        void print(ostream &os) const {
            os << pnr << "  " << name;
        };
};
```

Objekt och ärvning

```
class Student : public Person {
        int year;
      public:
        Student(int pp, const string &nn, int yy)
            : Person(pp, nn), year(yy) {};
        int &getYear() const {
            return year;
        };
        void print(ostream &os) const {
            Person::print(os);
            os << "  " << year;
        };
};
```

fungerar


```
int main() {
  Person joe(123, “Joe”);
  Student sue(456, “Sue”, 2011);

  joe.print(cout);

  sue.print(cout);

  cout << sue.getName() << “ “ << sue.getYear();

}
```

lite annorlunda


```
void print_person(const Person &p) {
     p.print(cout);
}


int main() {
  Student sue(456, “Sue”, 2011);

  print_person(sue);

}
```

C++

- static dispatch
  - Metod bestäms vid kompilering och utgår från variabelns typ.
- dynamic dispatch (Java)
  - Metod bestäms vid körning och utgår från objektets typ.

C++

- Om det finns två sätt att göra det på så gör C++ det på:
  - båda sätten

virtual

```
class Person {
        int pnr;
        string name;
      public:
        Person(int pp, const string &nn)
            : pnr(pp), name(nn) {};
        const string &getName() const {
            return name;
        };
        virtual void print(ostream &os) const {
            os << pnr << "  " << name;
        };
};
```

abstrakta klasser

```
class Triangle {
      pos x, y, z;
  public:
      double area();
      int edges();
};

class Rectangle {
      pos x11,x12, x21, x22;
  public:
      double area();
      int edges();
};
```

abstrakta klasser

```
class Shape {
  public:
     virtual double area();
     virtual int edges();
};

class Triangle : public Shape {
     pos x, y, z;
  public:
     double area();
     int edges();
};
```

abstrakta klasser

```
void print(const Shape &shp) {
      cout << Edges : ” << shp.edges() << endl;
      cout << Area : ” << shp.area() << endl;
}
```

abstrakta klasser

```
class Shape {
  public:
     virtual double area() = 0;
     virtual int edges() = 0;
};
```

list_char

```
class list_char  {
  public:
     char car;
     list_char *cdr;

     list_char(char c, list_char *cd)
       : car(c), cdr(cd) {};

     void print(ostream &os) const {
       os << car;
       if( cdr != NULL) {
          os << " -> ";
                :
                :
};
```

list_int

```
class list_int  {
  public:
     int car;
     list_int *cdr;

     list_int(char c, list_int *cd)
       : car(c), cdr(cd) {};

     void print(ostream &os) const {
       os << car;
       if( cdr != NULL) {
          os << " -> ";
                :
                :
};
```

template

```
template <typename Type> class List {
  public:
     Type car;
     List *cdr;

     List(Type c, List *cd)
       : car(c), cdr(cd) {};

     void print(ostream &os) const {
       os << car;
       if( cdr != NULL) {
          os << " -> ";
                :
                :
};
```

template

```
   :
List<int> *Íi = new List<int>(42, NULL);
   :


List<double> *ld = new List<double>(4.2, NULL);
```