Polymorfi

Mångskiftande beteende

Arv

```
class Animal
{
public:
  Animal(bool petable)
   : m_petable(petable) {}

  bool dangerous();

protected:
  bool m_hungry;
private:
  bool m_petable;
};

// Repetition!
```

```
class Cat : public Animal
{
public:
  Cat() : Animal(true) {}

  void purr();
};

class Dog : public Animal
{
public:
  Dog() : Animal(true) {}
};
```

Polymorfi

- Gör att ”rätt” medlemsfunktion anropas när man har en pekare eller en referens till en basklass som egentligen går till någon av basklassens subklasser... basklassen ser ut bete sig olika beroende vilken klass pekaren eller referensen verkligen refererar till
- Kräver att en pekare eller referens till basklassen används eftersom en vanlig variabel inte kan vara ”mer” än basklassen, subklasser som är ”mer” får helt enkelt inte plats i en vanlig basklassvariabel. Däremot kan de pekas ut eller refereras!

Arv vs. Polymorfi

- Arv – statisk bindning
  - Default
  - Kompilatorn bestämmer vid kompilering vilken implementation (av medlemssfunktion) som anropas vid körning
  - Avgörs utifrån hårdkodad datatyp
- Polymorfi – dynamisk bindning
  - Nyckelordet virtual
  - Programmet avgör under körning vilken implementation (av medlemssfunktion) som anropas
  - Avgörs utifrån verklig datatyp
- Rita hierarki – förklara!

Polymorfi

```
class Animal
{
public:
  Animal(bool petable)
   : m_petable(petable) {}
  virtual ~Animal() {}

  bool dangerous();
  virtual string walk() = 0;
  virtual string talk() {
    return "Hear hear!";
  }
protected:
  bool m_hungry;
private:
  bool m_petable;
};

// Förklara! Vad är fel?
```

```
class Cat : public Animal
{
public:
  Cat() : Animal(true) {}
  string walk() {
    return "Catwalk";
  }
  void purr();
};

class Dog : public Animal
{
public:
  Dog() : Animal(true) {}
  string talk() {
    return "Woff";
  }
};
```

Användning

```
int main()
{
  vector<Animal*> v = { ... };

  random_shuffle(v);

  for (auto it = v.begin(); it != v.end(); ++i)
    cout << it->talk() << endl;

  for (auto it = v.begin(); it != v.end(); ++i)
    delete *it;

  return 0;
}

// Förklara!
```

Anrop av basklassmetod

```
string Cat::talk()
{
  // explicit anrop av talk() i Animal
  cout << Animal::talk() ": Miow" << endl;
}
```

Polymorf destruktor

- Alla polymorfa klasser skall ha polymorf (virtual) destrutor.
- Destruktorn för subklassen anropas då först, sedan destruktorn för basklassen. Mest specialiserade destruktorn bör ju köras först.
- Utan polymorf destruktor skulle inte subklassens destruktor köras när en pekare eller referens till den (av basklasstyp) tas bort.

Dynamisk typomvandling

dynamic_cast<TILLPEKARE>(BASPEKARE)

```
Animal* a = new Cat();
a->purr(); // ERROR
Cat* cat = dynamic_cast<Cat*>(a); // OK
if (cat != nullptr)
  cat->purr(); //OK
Dog* dog = dynamic_cast<Dog*>(a); //nullptr

Dog* dog = static_cast<Dog*>(a); // ??
Dog* dog = reinterpret_cast<Dog*>(a); // ??
```

Typinformation

```
#include <typeinfo>
Animal* a = new Animal();
Animal* ac = new Cat();
Animal* ad = new Dog();

// Kompilatorberoende !!
// (och alltså inte så användbart)
cout << typeid(ac).name() << endl;
cout << typeid(*ac).name() << endl;
cout << typeid(ad).name() << endl;
cout << typeid(*ad).name() << endl;

cout << typeid(a) == typeid(ac) << endl;
cout << typeid(*a) == typeid(*ac) << endl;
cout << typeid(ad) == typeid(ac) << endl;
cout << typeid(*ad) == typeid(*ac) << endl;
```

Mer exempel

- Rita! Diskutera!
- Person
  - Employee
    - Consultant
    - Boss
    - Hourly
- getSalary(…)

Föredra privat polymorfi!

```
class Bas
{
public:
  // publikt gränssnitt (stabilt, ändras ej)
  double calculate() { return calc(); }
private:
  // anpassningsbart beteende
  // frihet att ändra och anpassa senare!
  virtual double calc();
};
```

C++11 delete och default

```
class Base
{
public:
  void foo(int);
};

class Sub : public Base
{
public:
  Sub(int); // kompilatorgenererade defaultkonstruktorn ersätts
  Sub() = default; // återskapa, mer effektiv än egen tom konstruktor

  // tar bort funktioner för tilldelning och kopiering
  Sub(Sub const&) = delete;
  Sub& opertor=(Sub const&) = delete;

  // tar bort inärvda funktionen foo
  void foo(int) = delete;
};
```

Dugga

- Specifikation
  - Okomplicerad
  - Skilj på specifikation av klass (programmeringsgänssnitt) och huvudprogram (anv. gränssnitt)
- Klass
  - Beskriven "rakt på", ingen egentlig analys
  - Inga krav på "const"
- Slump
  - Initiera EN gång!
  - Inkludera inte onödiga filer! (endast kommentar)
- Beräkningar
  - "Rakt på", ingen problemlösning
- Felhantering (borttaget ur bedömning)
  - Kontrollera inläsningen omedelbart
  - Variabel orörd vid fel (kanske oinitierad!)
  - Först "clear" sedan "ignore"
- Filuppdelning (ingick ej)
  - Inkluderingsfil med gard
  - Implementationsfil kompileras

Fö / Fö

Undantag

- Fel kan kastas
  - throw
- Kastade fel kan fångas av funktionen som anropade eller funktionen som anropade den eller funktionen som anropade den osv.
  - try {} catch(exception& e) {}
- main är sista chansen att fånga ett kastat fel. Ej fångade fel landar i OS som då terminerar programmet
- För fel som detekteras av (i) programmet (dig) – fel som detekteras i OS (segmentation fault) ger inte exception
- Förklara!

Undantag med strömmar

```
#include <limits> // numeric_limits<DATATYP>

cin.exceptions(ios::failbit | ios::eofbit | ios::badbit);
bool stupid_user = true;

while ( stupid_user )
{
  try
  {
    int i;
    cin >> i;
    stupid_user = false;
  }
  catch ( exception& e)
  {
    cerr << e.what() << endl;
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
}
```

Basundantag

```
#include <exeption>

// Ger tillgång till följande basklass:
class exception
{
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
};
```

My_Error

```
#include <exeption>
#include <string>

class My_Error : public exception
{
public:
  My_Error(std::string const& err) throw()
   : exception(), mErr(err);

  const char* what() const throw()
  {
    return mErr.c_str();
  }
private:
  std::string mErr;
};
```

Standardundantag

```
#include <stdexcept>
```