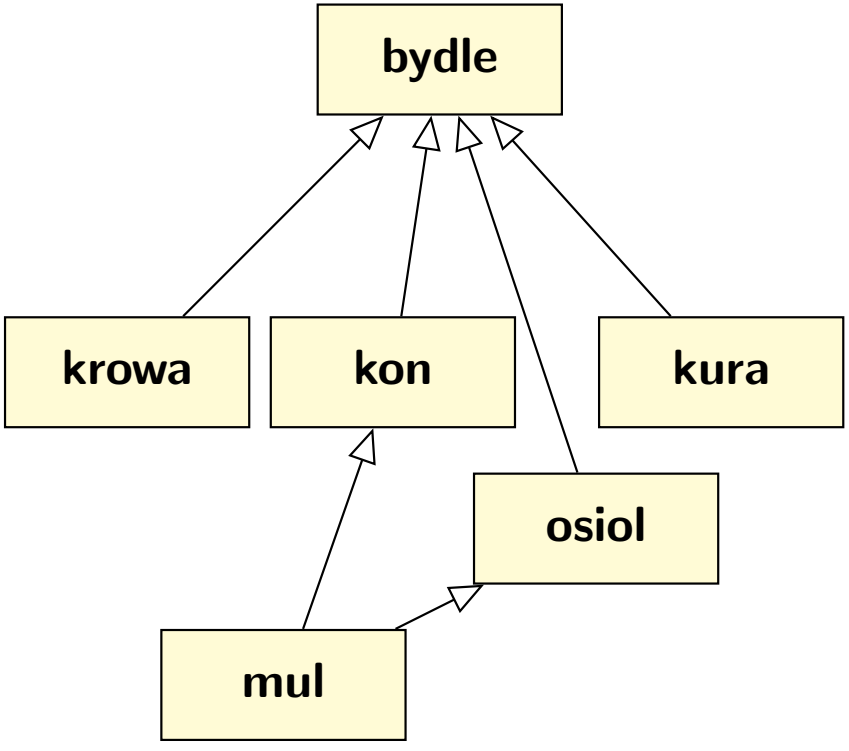
Programowanie obiektowe w języku C++

Stanisław Gepner

sgepner@meil.pw.edu.pl

Polimorfizm - dokończenie

Lepsza farma



- Polimorfizm - czyli wielopostaciowość
- Zdolność kodu do różnego zachowania w trakcie wykonania
- Wskaźnik polimorficzny *bydle *p = ...*
- ... tzn. taki, który możemy róznie interpretować
- To funkcje składowe klasy mogą być polimorficzne,
- a nie obiekt klasy
- Określenie, która funkcja jest wykonana dla danego obiektu jest przesunięte do czasu wykonania

Szablony

Jak pisać mniej a dostać więcej

- Szablon, *templejt**, generyk ...
- Umożliwia tworzenie kodu niezależnego od typów, większa elastyczność
- Pozwala na nieduplikowanie kodu
- Wykonanie w czasie kompilacji, jak makra preprocesora C
- Szablony funkcji, klas, metod
- Szablon może być oparty o typ (klasę) lub wartość całkowitą
- Metaprogramowanie: w oparciu o mechanizm szablonu. Tak język szablonów jest kompletny w sensie Turinga.

```
template <class T>
typ_funkcji fun(argumenty)
{
  Cialo funkcji
}

template <class T>
class List
{
    ...
};
```

*takiego słowa nie ma ...

Bez szablonów funkcji

Po co? przykład z przeładowaniem

Tworzymy funkcję *square()* liczącej kwadrat zmiennej każdego typu

```
int square(int x){  return x*x;}
short int square(short int x){  return x*x;}
unsigned short int square(unsigned short int x){  return x*x;}
long square(long x){  return x*x;}
unsigned long square(unsigned long x){  return x*x;}
long long square(long long x){  return x*x;}
unsigned long long square(unsigned long long x){  return x*x;}
float square(float x){  return x*x;}
double square(double x){  return x*x;}
// i tak dalej, i jeszcze dla naszych typow ...
```

Bez szablonów funkcji

Po co? przykład z przeładowaniem

A potem dodamy jeszcze dla wszystkich klas jakie tworzymy...

```
class complex{//liczba zespolona na int
  public:
  complex(int r, int i) : real(r), imag(i) {}
  const int& c_real(){return real;}
  const int& c_imag(){return imag;}
  private:
  int real;
  int imag;
};

complex square_elements(complex& a){
  return complex(a.c_real()*a.c_real(),
                 a.c_imag()*a.c_imag());
}
```

A może by tak napisać program do pisania programu?!

Szablon funkcji

Składnia

Zamiast pisać dużo kodu możemy napisać 'ogólną receptę':

Tak to język w języku ...

```
template <class T> // l typename
typ_funkcji fun(argumenty)
{
  Cialo funkcji
}
```

W naszym przypadku:

```
template <class T>
T square(T x)
{
  return x*x; // T musi wiedziec co to '*'
}
```

Szablon funkcji

Składnia

```
template <class T>
T square(T x)
{
  return x*x; // T musi wiedziec co to '*'
}

int a=9;
square<int>(a);
```

nasza klasa *complex* nie wie co to * ...

Szablon funkcji
Domyślny kompilator

```
template <class T>
T square(T x)
{
  return x*x; // T musi wiedziec co to '*'
}

double b=9;//kompilator 'domysli sie typu
square(b);
```

W przypadku szablonów funkcji można pominąć <> kompilator postara się zdecydować za nas. No chyba, że nie może ... na przykład:

```
template<typename T>
T my_type_caster_no_sense(double x)
{
  T a = x; //rzutowanie na T
  return a;
}
```

Szablon funkcji

Argumenty szablonu

- Typy, *class* lub *typename*
- Wartości całkowite, *int*, *enum* ...

```
template <class T> //typename
T square(T x)
{
  return x*x; // T musi wiedziec co to '*'
}

template<int V>
int templated_with_value(double x)
{
  return a*V;
}
```

Szablon funkcji

Argumenty szablonu - może być kilka

```
template <class T, int N> //typename
T pow(T x)
{
  T res=1;
  for(int i=0; i<N; ++i)
    res *= x;
  return res;
}
```

```
template <int N, class T> //typename
T pow(T x)
{
  T res=1;
  for(int i=0; i<N; ++i)
    res *= x;
  return res;
}
```

Specjalizacja

Czyli co robić dla określonych argumentów szablonu

```
template<typename T>
T fun(T x)
{
  return 2*x;
}

template<>
string fun<string>(string x)
{
  return x + "␣" +x;
}


int main()
{
  double a=9;
  cout << fun(a) << endl;

  string b="Ala␣ma␣kota";
  cout << fun(b) << endl;
}
```

Szablon klasy
Przykład

```
class complex_int{//liczba zespolona na int
  public:
  complex_int(int r, int i) : real(r), imag(i) {}
  const int& c_real(){return real;}
  const int& c_imag(){return imag;}
  private:
  int real;
  int imag;
};
```

```
class complex_double{//liczba zespolona na int
  public:
  complex_double(double r, double i) : real(r), imag(i) {}
  const double& c_real(){return real;}
  const double& c_imag(){return imag;}
  private:
  double real;
  double imag;
};
```

A inne typy? A kontenery?

Szablon klasy

Składnia

```
template <class T>
class class_name{//cialo oparte o T
  pyblic:
  class_name(){}
  void fun();
  private:
  T a;
  T b;
};

template<class T>
void class_name<T>::fun()
{
...
}
```

Szablon klasy

Różne!

```
template <class T>
class class_name{//cialo oparte o T
  pyblic:
  class_name(){}
  void fun();
  private:
  T a;
  T b;
};

...
class_name<int> != class_name<double>!!!
```

Szablon klasy

Argumenty

- Typy, *class* lub *typename*
- Wartości całkowite, *int*, *enum* ...

```
template <class T>
class class_name{//cialo oparte o T
  pyblic:
  class_name(){}
  void fun();
  private:
  T a;
  T b;
};

template<int V>
class class_name2{
  pyblic:
  class_name(){}
  void fun();
  private:
  int a[V];
  double b[V];

};
```

Szablon klasy

Argumenty szablonu - może być kilka

```
template <class T, int N> //typename
class class_name{//cialo oparte o T
  pyblic:
  class_name(){}
  void fun();
  private:
  T tab[N];
};
```

```
template <int N, class T> //typename
class class_name{//cialo oparte o T
  pyblic:
  class_name(){}
  void fun();
  private:
  T tab[N];
};
```

Szablon klasy

Specjalizacja, czyli co robić dla określonych argumentów szablonu

```
template<int N>
class foo{
  public:
  foo();
  void fun();
  double tab[N];
};

template<int N>
foo<N>::foo()
{}

template<>
foo<0>::foo()
{
  cout << "This␣will␣be␣empty!!!" << endl;
}
```

ZŁO!

Język w języku

- Okazuje się, że mechanizm szablonów w C++ jest osobnym językiem
- Tak, wewnątrz C++ zaszyto inny język
- Turing Kompletny ...
- Odkryto to przypadkiem
- I tak, ma zastosowania
- Pozwala na wykonanie "programu"w czasie kompilacji - metaprogramowanie!

Przykład

Silnia - klasycznie

```
unsigned int factorial(unsigned int n)
{
  cout << "Called␣with␣n=" << n << endl;
  return n == 0 ? 1 : n * factorial(n - 1);
}

int main()
{
  cout <<"␣Finally␣the␣factorial␣is:" << factorial(4) << endl;
}
```

Przykład

Silnia - inaczej

```
template <unsigned int n>
class factorial {
public:
  static const unsigned long long value = n * factorial<n-1>::value;
};

template <>
class factorial<0> {
public:
  static const unsigned long long value = 1;
};


int main()
{
  std::cout << factorial<5>::value << "\n";
}
```

Częściowa Specjalizacja

```
template<int N, class T>
class foo{
  public:
  foo();
  void fun();
  T tab[N];
};
template<class T> //specialized for N=0
class foo<0,T>{
  public:
  foo();
  void fun();
  T tab[0];
};
template<int N, class T>
foo<N,T>::foo()
{}
template<class T>
foo<0,T>::foo()
{
  cout << "This will be empty!!!" << endl;
}
```