Jacek Matulewski
http://www.fizyka.umk.pl/~jacek/

# Szablony 1: Podstawy

Wersja α

Toruń, 2 listopada 2006

Najnowsza wersja dokumentu dostępna pod adresem
http://www.fizyka.umk.pl/~jacek/dydaktyka/cpp/cpp-szablony1.pdf

Źródła programów z tego skryptu (C++Builder 6 i Visual C++ 2005):
http://www.fizyka.umk.pl/~jacek/dydaktyka/cpp/cpp-szablony1.zip

# Szablony

**Szablony = wzorce = typy parametryczne = typy ogólne (generyczne)**

Kody dostępne w wersjach C++Builder 6 i Visual C++ 2005 Express. W tekście wersja C++Builder.

Metaprogramowanie (tu: programowanie funkcji i klas, które mogą być wykorzystane dla różnych typów danych):

1) makra (brak kontroli typow i kontroli zakresow)
2) programowanie dla void* (ewentualnie TObject) (wady j.w.)
3) szablony

Biblioteki standardowe C++ oparte są na szablonach.

# Szablony funkcji i szablon `vector` z biblioteki STL

0. Dołączamy nagłówek:

```
#include <vector.h>
```

1. Tworzymy obiekty tego typu (np. w metodzie zdarzeniowej przycisku)

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    typedef int typ;
    //typedef float typ;

    int rozmiar=5; //nie musi byc const
    vector<typ> v(rozmiar);
    for(int i=0;i<rozmiar;i++) v[i]=(typ)(i/2.0);
}
```

**parametr szablonu** (typ podany w nawiasach ostrych), może być typem prostym, aliasem (`typedef`) lub klasą

**konkretyzacja szablonu** (czasownik, ang. instantation) = proces zastepowania parametru konkretnym typem, to robi kompilator dla wykorzystywanych w kodzie przypadkow szablonu

**specjalizacja** generowana (rzeczownik) = ukonkretniony przez kompilator typ (prawdziwa klasa)

2. Tworzymy szablon funkcji (musi znaleźć się nad metodą zdarzeniową):

```
template<typename T> void Drukuj(int ile,vector<T> v)
{
    AnsiString s="vector: ";
    for(int i=0;i<ile;i++) s+=(AnsiString)v[i]+"  ";
    ShowMessage(s);
}
```

I dodajemy jego wywołanie w metodzie:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    typedef int typ;
```

```
    //typedef float typ;

    int rozmiar=5;    vector<typ> v(rozmiar);
    for(int i=0;i<rozmiar;i++) v[i]=(typ)(i/2.0);
    Drukuj<typ>(rozmiar,v);
}
```

3. Możemy nie wskazywać jawnie typu parametru, a wówczas kompilator wydedukuje go po użytych argumentach:

```
Drukuj(rozmiar,v);
```

4. Typ `vector` to szablon. Natomiast `vector<int>` (konkretyzacja szablonu) jest już dla kompilatora (w drugim przebiegu kompilacji) zwykłą klasą. Możemy użyć jej metod:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    typedef int typ;
    //typedef float typ;

    int rozmiar=5;
    vector<typ> v(rozmiar);
    for(int i=0;i<rozmiar;i++) v[i]=(typ)(i/2.0);
    Drukuj<typ>(rozmiar,v);

    //Uzycie metod skladowych
    typ element=(typ)1;
    v.push_back(element); //dodawanie elementow na koncu wektora
    v.push_back(0);
    v.push_back(0);
    v.erase(v.begin()+1); //usuniecie drugiego (arytm. wskaznikow), v.begin - iterator
do pierwszego, v.end - do ostatniego, iterator=uogulniony wskaznik
    v.pop_back(); //usuniecie ostatniego elementu
    Drukuj<typ>(v.size(),v); //v.size
}
```

5. Typ `vector` jest znacznie inteligentniejszy od zwykłej tablicy, którą ma zastępować. Obiekt `v` przechowuje np. wszelkie informacje o sobie, w tym rozmiar. Przy przekazywaniu jako argument funkcji pamięta swój rozmiar. Dla porównania tablica konwertowana jest na wskaźnik (patrz skrypt *tablice*) i „zapomina" swój rozmiar.
   Możemy wobec tego pominąć pierwszy argument w funkcji `Drukuj`. Stwórzmy jej przeciążoną wersję:

```
template<typename T> void Drukuj(vector<T> v)
{
    AnsiString s="vector: ";
    typedef unsigned int uint;
    for(uint i=0;i<v.size();i++) s+=(AnsiString)v[i]+"  ";
    ShowMessage(s);
}
```

Możemy teraz zmienić wywołanie funkcji Drukuj na:

```
Drukuj(v);
```

Szablon `vector` pomimo całej swej mądrości nie zapobiega zapisywaniu „za" tablicą np. `v[10]=2;`!

6. ZADANIE: Wyczyścić zawartość wektora `v` (metoda `clear`). Umieścić w nim dwa elementy: `0` i `1`. Przygotować funkcję `PushFibonacci`, która przyjmuje argument `vector<T>` i dodaje do niego element, który jest sumą dwóch wcześniejszych elementów. Następnie wywołać tę metodę 30 razy i wynik pokazać za pomocą metody `Drukuj`.

Funkcja:

```
template<typename T> void PushFibonacci(vector<T>& v)
{
    v.push_back(v.back()+v[v.size()-2]);
}
```

Wykorzystanie:

```
v.clear();
v.push_back(0);
v.push_back(1);
for(int i=0;i<30;i++) PushFibonacci<typ>(v);
Drukuj<typ>(v);
```

# Szablon klasy

1. Zdefiniujmy szablon klasy `TKlasa1` zawierającej pole o nazwie `pole1` typu parametru `T`:

```
template<typename T> class TKlasa1
{
    private:
        T pole1;
    public:
        TKlasa1():pole1(T()){} //T() zwraca wartosc domyslna dla danego typu
        TKlasa1(T pole1):pole1(pole1){}
        AnsiString ToString(){return (AnsiString)""+pole1;};
}
```

Możemy przetestować szablon na wielu typach tworząc obiekty na stosie i na stercie:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    TKlasa1<int> obiekt1a;
    ShowMessage("1a: "+obiekt1a.ToString());
    TKlasa1<int> obiekt1b(1);
    ShowMessage("1b: "+obiekt1b.ToString());
    TKlasa1<int>* pobiekt1c=new TKlasa1<int>(2);
    ShowMessage("1c: "+pobiekt1c->ToString());
    delete pobiekt1c;
}
```

2. Defiujemy klasę potomną dodającą do klasy bazowej pole o nazwie `pole2` typu `S`:

```
template<typename T,typename S> class TKlasa2 : public TKlasa1<T>
{
    private:
        S pole2;
    public:
```

```
        TKlasa2():TKlasa1<T>(),pole2(S()){}
        TKlasa2(T pole1,S pole2):TKlasa1<T>(pole1),pole2(pole2){}
        AnsiString ToString(){return (AnsiString)"("+TKlasa1<T>::ToString()+",
"+pole2+")";};
}
```

Testujemy nową klasę:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    TKlasa1<int> obiekt1a;
    ShowMessage("1a: "+obiekt1a.ToString());
    TKlasa1<int> obiekt1b(1);
    ShowMessage("1b: "+obiekt1b.ToString());
    TKlasa1<int>* pobiekt1c=new TKlasa1<int>(2);
    ShowMessage("1c: "+pobiekt1c->ToString());
    delete pobiekt1c;

    TKlasa2<int,float> obiekt2a;
    ShowMessage("2a: "+obiekt2a.ToString());
    TKlasa2<int,float> obiekt2b(1,2.0);
    ShowMessage("2b: "+obiekt2b.ToString());
    TKlasa2<int,float>* pobiekt2c=new TKlasa2<int,float>(3,4.0);
    ShowMessage("2b: "+pobiekt2c->ToString());
    delete pobiekt2c;
}
```

3. Do obu klas dodajemy szablony metod:

```
template<typename T> class TKlasa1
{
    private:
        T pole1;
    public:
        TKlasa1():pole1(T()){} //T() zwraca wartosc domyslna dla danego typu
        TKlasa1(T pole1):pole1(pole1){}
        AnsiString ToString(){return (AnsiString)""+pole1;};

    public:
        T GetPole1() const {return pole1;}
        void SetPole1(T pole1){this->pole1=pole1;}
};

template<typename T,typename S> class TKlasa2 : public TKlasa1<T>
{
    private:
        S pole2;
    public:
        TKlasa2():TKlasa1<T>(),pole2(S()){}
        TKlasa2(T pole1,S pole2):TKlasa1<T>(pole1),pole2(pole2){}
```

```
        AnsiString ToString(){return (AnsiString)"("+TKlasa1<T>::ToString()+",
"+pole2+")";};


    public:
        S GetPole2() const {return pole2;}
        void SetPole2(S pole2){this->pole2=pole2;}
};
```

Aby je przetestować możemy do metody zdarzeniowej przycisku dodać następujące instrukcje:

```
obiekt2a.SetPole1(5);
obiekt2a.SetPole2(6);
ShowMessage("2aSet: "+obiekt2a.ToString());
```

4. Definiujemy operator + przyjmujący jako argumenty obiekty typu TKlasa2:

```
template<typename T,typename S> TKlasa2<T,S> operator +(const TKlasa2<T,S>&
skladnikA,const TKlasa2<T,S>& skladnikB)
    {
    return
TKlasa2<T,S>(skladnikA.GetPole1()+skladnikB.GetPole1(),skladnikA.GetPole2()+skladnikB.Ge
tPole2());
    };
```

Ponownie do metody zdarzeniowej przycisku dodajemy polecenia testujące poprawność kodu:

```
TKlasa2<int,float> wynik=obiekt2a+obiekt2b;
ShowMessage("2a+2b: "+wynik.ToString());
```

5. Operatora można użyć dla dowolnych parametrów ze zdefiniowanym operatorem + nie tylko dla typów arytmetycznych np. dla łańcucha:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
//uzycie do typu innego niz arytmetyczny (musi miec operator +)
    TKlasa2<int,AnsiString> a(1,"Malleus");
    ShowMessage("a: "+a.ToString());
    TKlasa2<int,AnsiString> b(2,"Maleficarum");
    ShowMessage("b: "+b.ToString());
    ShowMessage("a+b: "+(a+b).ToString());
}
```

# Typy domyślne i specjalizacje

6. Można określić typ domyślny jednego z parametrów. Tu stworzymy klasę potomną TKlasa3, ale można go dodać także do TKlasa2.

```
//wartosc domyslna parametrow szablonu
template<typename T,typename S=int> class TKlasa3 : public TKlasa2<T,S>
{
    public:
        TKlasa3(T pole1,S pole2):TKlasa2<T,S>(pole1,pole2){ShowMessage("Klasa z
parametrem domyslnym");}
};
```

7. Specjalizacje generowane są przez kompilator. Można je jednak również zdefiniować samemu, co daje okazję aby do klasy dodać kod dla tych konkretnych typów użytych jako parametrów. Można

wyspecyfikować wszystkie parametry (**specjalizacja całkowita**) lub tylko niektóre (**specjalizacja częściowa**).

Specjalizacja częściowa z własnym konstruktorem

```
template<typename S> class TKlasa3<float,S> : public TKlasa2<float,S>
{
    public:
        TKlasa3(float pole1,S pole2):TKlasa2<float,S>(pole1,pole2){ShowMessage("Klasa z
czesciowa specjalizacja utworzona przez programiste");}
};
```

Specjalizacja całkowita z własnym konstruktorem

```
template<> class TKlasa3<float,int> : public TKlasa2<float,int>
{
    public:
        TKlasa3(float pole1,int pole2):TKlasa2<float,int>(pole1,pole2)
{ShowMessage("Klasa z calkowita specjalizacja utworzona przez programiste");}
};
```

Testowanie wszystkich wersji klasy `TKlasa3`:

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
    //par. domyslny
    TKlasa3<int> a(1,2); //dla <float>=<float,int> uzyta zostalaby specjalizacja
czesciowa
    ShowMessage("a: "+a.ToString());

    //specjalizacja częściowa
    TKlasa3<float,AnsiString> b(1.0f,"Yotz!");
    ShowMessage("b: "+b.ToString());

    //specjalizacja całkowita
    TKlasa3<float,int> c(1.0f,2);
    ShowMessage("c: "+c.ToString());
}
```

Tu specjalizacje tworzone przez programistę były klasami potomnymi – w przypadku klas nie dziedziczących ich deklaracje się upraszczają.

# Parametry szablonów nie będące typami

8. Jako parametrów szablonu można użyć wartości całkowitych lub wskaźników, a nie typów

```
//parametry niebedace typami
template<typename T,int STALA> class TNowaKlasa
{
    private:
        T tablica[STALA];
    public:
        TNowaKlasa(T wartosc)
```

```
        {
            for(int i=0;i<STALA;i++) tablica[i]=wartosc;
            ShowMessage("STALA="+IntToStr(STALA));
        };
        TNowaKlasa(T tablica[STALA]):tablica(tablica){};
        AnsiString ToString()
        {
            AnsiString s="";
            for(int i=0;i<STALA;i++) s+=(AnsiString)tablica[i]+" ";
            return s;
        }
};

//czesciowa specjalizacja
template<typename T> class TNowaKlasa<T,0>
{
    private:
        T zmienna;
    public:
        TNowaKlasa(T wartosc):zmienna(wartosc){ShowMessage("Brak tablicy");};
        AnsiString ToString()
        {
            return (AnsiString)zmienna;
        }
};

void __fastcall TForm1::Button5Click(TObject *Sender)
{
    /** 8 **/
    TNowaKlasa<double,10> nk1(0.0);
    ShowMessage((AnsiString)"nk1: "+nk1.ToString());

    TNowaKlasa<double,0> nk2(0.0);
    ShowMessage((AnsiString)"nk2: "+nk2.ToString());
}
```

# Zadania (szablony vs. makra)

1. Napisać dwuargumentowe makro `MIN` zwracające mniejszy z dwóch podanych argumentów

```
#define MIN(x,y) ((x)<(y)?(x):(y))
```

2. Napisać dwuargumentowa funkcje `Min` zwracającą mniejszy z dwóch podanych argumentów

```
template <class T> inline const T& Min(const T& t1,const T& t2)
{
    return (t1<t2)?t1:t2;
}
```

Testy obu wersji:

```
void __fastcall TForm1::Button5Click(TObject *Sender)
{
    //makro
    ShowMessage(IntToStr(MIN(5,1)));
    ShowMessage(FloatToStr(MIN(0.5,0.1)));

    //szablon
    ShowMessage(IntToStr(Min<int>(5,1)));
    ShowMessage(FloatToStr(Min<double>(0.5,0.1)));
}
```

3. Zanalizować dokumentacje szablonu klasy `complex` (Standard C++ Library = std).
   Zdefiniować typ `dcomplex` - konkretyzacje szablonu `complex` dla `double`.
   Zdefiniować stałą i (jedn. urojona).

```
void __fastcall TForm1::Button7Click(TObject *Sender)
{
    typedef std::complex<double> dcomplex;
    const dcomplex ic=dcomplex(0,1);
    const dcomplex uc=exp(ic*M_PI);
    ShowMessage((AnsiString)"complex uc="+real(uc)+"+"+imag(uc)+"i");
}
```

# Ciekawostka: metaprogramowanie

Źródło: D. Vandervoorde, N.M. Joutis „C++. Szablony. Vademecum profesjonalisty” s. 286

Sztuczka z obliczaniem n-tej potęgi liczby 3 w szablonie korzystająca z konkretyzacji rekurencyjnej:

```
//szablon bazowy = regula rekurencji
template<int N> class Pow3
{
    public:
        enum{result=3*Pow3<N-1>::result};
};

//specjalizacja calkowita = przerwanie rekurencji (tu rekurencja w dol)
template<> class Pow3<0>
{
    public:
        enum{result=1}; //dla N=0
};

#include <math.h>
void __fastcall TForm1::Button7Click(TObject *Sender)
{
    ShowMessage((AnsiString)"Pow3<7>::result = "+Pow3<7>::result);
    ShowMessage((AnsiString)"pow(3,7) = "+pow(3.0,7));
}
```