

Programowanie obiektowe

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

<http://home.agh.edu.pl/~pszwed/>

Aktualizacja: 25.10.2019

7. Szablony

Szablony

- Szablony (*ang. templates*) są mechanizmem, który w języku C++ zyskał bardzo szerokie uznanie, zwłaszcza po pojawieniu się około 1995 roku standardowej biblioteki C++, znanej także jako STL (*Standard Template Library*).
- Szablony wyparły z języka C++ kontenery obiektowe:
 - Konieczność ograniczenia się do jednej hierarchii klas
 - Kontenery obiektowe są niewydajne dla typów wbudowanych (int, char, double)

Szablony

- Idea użycia szablonów jest nieco podobna do wykorzystania makr preprocesora. W pierwszym wydaniu książki *C++ Programming Language*, Stroustrup zaproponował implementację kontenerów właśnie za pomocą makr preprocesora.
- W przypadku szablonów - zamiast mechanizmu dziedziczenia - następuje wielokrotne użycie *kodu źródłowego*.
- Kontenery zdefiniowane w postaci szablonów nie zawierają wskaźników do obiektów potomnych klasy `Object`, ale **nieokreślone parametry**, które w momencie użycia szablonu są zastępowane przez kompilator odpowiednimi wartościami typów lub wartościami liczbowymi.

Makra preprocesora

- Makra preprocesora pozwalają na wielokrotnie wstawienie do kodu instrukcji, których działanie może przypominać wywołanie funkcji lub procedury.
- Podczas działania preprocesora następuje substytucja tekstu bez sprawdzania typów. Pozwala to na zdefiniowanie wzorców kodu, który może działać dla różnych typów danych.

Makra preprocesora - przykład

```
#define min(a,b)      (a<b?a:b)

#define sum(table,size,result)      \
{                                     \
    for(int i=0;i<size;i++)result+=table[i]; \
}

#define duplicate(table,size,T,result) \
{                                       \
    result=new T[size]; \
    for(int i=0;i<size;i++)result[i]=table[i]; \
}
```

- Znak \ to kontynuacja wiersza (wstępne przetwarzanie tekstu przed fazą preprocesora)
- Makro preprocesora musi być zdefiniowane w jednej linii

Makra preprocesora - przykład

```
int x=1,y=2;
int m= min(x,y) ; // (x<y?x:y)

float t[3]={1.1F,200,-45};
float result=0;
sum(t,3,result);
float *tcopy;
duplicate(t,3,float,tcopy);

char s[]="Ala ma kota";
char*copy;
duplicate(s,strlen(s)+1,char,copy);
```

- Makro `min` będzie działało poprawnie dla dowolnego typu, dla którego zdefiniowany jest operator `<`.
- Makro `sum` dla typu, który definiuje operator `+=`
- Makro `duplicate`, dla typu z poprawnie działającym operatorem przypisania.

Składnia szablonów

Szablony oferują znacznie większe możliwości, niż makra preprocesora. Pozwalają na zdefiniowanie parametryzowanych wzorców funkcji i klas.

Składnia

template < [typelist] [, [arglist]] > declaration,

1. **typelist** – jest listą typów użytych w szablonie; definicje typów oddzielone są przecinkami i mają postać:
 - **class** identifier lub
 - **typename** identifier
2. **arglist** – jest listą argumentów (analogiczną do argumentów funkcji)
3. **declaration** – jest deklaracją funkcji lub klasy, w której zazwyczaj używane są identyfikatory zdefiniowane w liście typów i argumentów.

Przykład

```
template<class T, int size>
class Array
{
public:
    T buffer[size];
    operator const T*()const{return buffer;}
    operator T*(){return buffer;}
};
```

Instancjacja szablonów

- Wywołanie funkcji lub utworzenie obiektu klasy zdefiniowanej w szablonie wymaga podania parametrów szablonu: nazw typów lub wartości argumentów (obiektów lub wartości liczbowych).
- Kod funkcji i definicje klas nie są generowane w trakcie wykonania programu, lecz w trakcie kompilacji, na podstawie statycznej analizy typów parametrów użytych w wywołaniu.
- Proces przypisywania parametrów szablonowi nazywany *instancją* szablonu. Dla każdego zestawu użytych parametrów generowana jest pojedyncza *instancja* kodu funkcji lub klasy.

```
void main()
{
    Array<char,256> string;
    strcpy(string,"Text");
    cout<<(const char*)string<<endl;
}
```

Szablony funkcji

- Szablon funkcji definiuje nieskończony zbiór funkcji, którego elementy mogą zostać utworzone w wyniku instancjacji.
- W definicji szablonu funkcji składowa *deClaration* jest deklaracją funkcji parametryzowaną listą typów i argumentów szablonu.
- Wygenerowane funkcje są dołączane do kodu wynikowego programu.

Szablony funkcji - przykład

```
template<class T>
T sum(const T*table, int size) {
    T result;
    for(int i=0;i<size;i++)result+=table[i];
    return result;
}
```

```
template<class T>
T*duplicate(const T*table, int size) {
    T*result=new T[size];
    for(int i=0;i<size;i++)result[i]=table[i];
    return result;
}
```

```
template<class T>
T min(T a, T b){
    return a<b?a:b;
}
```

Instancjacja

Proces instancjacji (czyli generowania kodu funkcji) może następować automatycznie w wyniku znalezienia przez kompilator jej wywołania. Typ funkcji jest określony na podstawie typu użytych argumentów.

```
int main()
{
    float t[3]={1.1F,200,-45};
    float result= sum(t,3);
    // sum<float>(t,3)
    float*fc=duplicate(t,3);
    // duplicate<float> (t,3)
    char s[]="Ala ma kota";
    char*copy= duplicate(s, strlen(s)+1);
    // duplicate<char>(s, strlen(s)+1)
}
```

Instancjacja

Brak jawnej specyfikacji parametrów szablonu jest wygodny – typ funkcji do wygenerowania pozostawiany jest kompilatorowi, ale równocześnie może prowadzić do niejednoznaczności. W takim przypadku raportowane są błędy.

Przykład `float a = min(1.0f, 2.1);`

Pierwszy z parametrów jest typu `float`, a drugi typu `double`. Kompilator nie jest w stanie dopasować argumentów, do zdefiniowanego wcześniej szablonu funkcji

```
template<class T> T min(T a, T b)
```

i tym samym automatycznie wygenerować jej instancji.

Jawna instancjacja

Instancja funkcji może być zdefiniowana jawnie przez podanie parametrów, dla których powinien być wygenerowany kod.

Składnia: `identyfikator<arglist>`

`identyfikator` – identyfikator funkcji zdefiniowanej w szablonie

`arglist` – lista argumentów (nazw typów i obiektów) oddzielonych przecinkami.

Przykład `float a = min<float>(1.0f, 2.1);`

Wygenerowana zostanie funkcja

```
float min(float, float);
```

Wołanie funkcji zostanie prawidłowo skompilowane (z ostrzeżeniem o możliwym obcięciu parametru 2.1 typu `double`).

Szablony klas

- Szablony klas definiują nieskończony zbiór klas, które mogą powstać w procesie instancjacji przez przypisanie parametrom szablonu identyfikatorów typów i argumentów będących obiektami.
- Szablony klas są mechanizmem ogólnego przeznaczenia, jednak najczęściej ich rolą jest zdefiniowanie kodu kontenera, który może być wielokrotnie wykorzystany przy generacji kontenerów zdolnych przechowywać obiekty określonego typu.

Szablon klasy Vector

```
template<class T>
class Vector
{
    friend class Iterator<T>;
    T*start;
    T*end;
    int capacity;
    void copy(const Vector&other);
    void free();
public:
    Vector(){start=end=0;capacity=0;}
    Vector (const Vector &other){copy(other);}
    ~Vector(){free();}
    Vector &operator=(const Vector &other) {
        if(&other!=this){free();copy(other);}
        return *this;
    }
    // kontynuacja na następnym slajdzie...
```

Szablon klasy Vector

```
// template<class T>
// class Vector
// {
// kontynuacja...
int getSize()const{return end-start;}
int getCapacity()const{return capacity;}
bool reserve(int newCapacity);
bool pushBack(const T&t);
T&operator[](int index){
    if(index<0||index>=(end-start)) throw -1;
    return start[index];
}
};
```

Szablon klasy Vector

```
template<class T>
void Vector<T>::copy(const Vector &other){
    start=end=0;capacity=0;
    if(other.getSize()){
        start = new T[other.capacity];
        capacity=other.capacity;
        for(int i=0;i<other.getSize();i++){
            start[i]=other.start[i];
        }
        end= start+other.getSize();
    }
}
```

```
template<class T>
void Vector<T>::free(){
    if(start)delete []start;
    start=0;capacity=0;
}
```

Szablon klasy Vector

```
template<class T>
bool Vector<T>::reserve(int newCapacity)
{
    if(newCapacity<capacity)return false;
    T*tmp=new T[newCapacity];
    if(capacity){
        for(int i=0;i<getSize();i++)tmp[i]=start[i];
        delete []start;
    }
    end=tmp+(end-start);
    capacity=newCapacity;
    start=tmp;
    return true;
}
```

Szablon klasy Vector

```
template<class T>
bool Vector<T>::pushBack(const T&t)
{
    if(capacity==getSize() && !reserve(capacity+16))
        return false;
    *end=t;
    end++;
    return true;
}
```

Szablon klasy Iterator

```
template<class T>
class Iterator
{
    const Vector<T>&vector;
    T*current;
public:
    Iterator(const Vector<T>&v)
        :vector(v),current(v.start){}
    Iterator&operator++(){current++;return *this;}
    Iterator operator++(int){
        Iterator tmp = *this;
        ++*this;
        return tmp;
    }
    bool good()const{return current<vector.end;}
    T&get()const{return *current;}
};
```

Instancjacja int

```
void test_int()
{
    Vector<int> v;
    for(int i=0;i<10;i++){
        v.pushBack(i);
    }
    Vector<int> v2=v;
    v2.pushBack(1000000);

    for(int i=0;i<v.getSize();i++){
        cout<<v[i]<<" ";
    }
    cout<<endl;
    for(Iterator<int> it(v2);it.good();++it)
        cout<<it.get()<<" ";
}
```

Wynik

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 1000000
```

Instancjacja typem użytkownika

```
typedef struct {double x;double y;}point;

void test_point(){
    Vector<point> plot;
    for(double t=0;t<10;t+=.0001){
        point p;
        double d = exp(cos(t))-2*cos(4*t)-pow(sin(t/12),5);
        p.x=sin(t)*d;
        p.y=cos(t)*d;
        plot.pushBack(p);
    }
    for(int i=0;i<10;i++){
        cout<<"("<<plot[i].x<<" "<<plot[i].y<<")"<<endl;
    }
}
```

```
(0 0.718282)
(7.18282e-05 0.718282)
(0.000143656 0.718282)
(0.000215485 0.718283)
...
```


Tablica 2D

```
void test_2d(){
    Vector<Vector<int>> tab;
    int cols = 8;
    int rows = 16;
    for(int i=0;i<rows;i++){
        Vector<int> row;
        for(int j=0;j<cols;j++){
            row.pushBack(i*cols+j);
        }
        tab.pushBack(row);
    }
    //...
```

- Do tab dodawanych jest 16 obiektów typu Vector. Każdy zawiera 8 elementów.
- W efekcie powstaje struktura danych o interfejsie tablicy dwuwymiarowej. Jej elementy to `tab[i][j]`, gdzie `i` to numer wiersza, a `j` to numer kolumny.

Tablica 2D

```
//...
for(int i=4;i<rows;i++) {
    for (int j = 0; j < 8; j++) {
        cout << (char) tab[i][j];
    }
    cout<<" ";
    for (int j = 0; j < 8; j++) {
        cout << setw(3)<<tab[i][j]<<" ";
    }
    cout << endl;
}
}
```

!"#\$%&'	32	33	34	35	36	37	38	39
()*+,-./	40	41	42	43	44	45	46	47
01234567	48	49	50	51	52	53	54	55
89:;<=>?	56	57	58	59	60	61	62	63
@ABCDEFGFG	64	65	66	67	68	69	70	71
HIJKLMNO	72	73	74	75	76	77	78	79
PQRSTUVWXYZ	80	81	82	83	84	85	86	87
XYZ[\]^_	88	89	90	91	92	93	94	95
`abcdefg	96	97	98	99	100	101	102	103
hijklmno	104	105	106	107	108	109	110	111
pqrstuvw	112	113	114	115	116	117	118	119
xyz{ }~□	120	121	122	123	124	125	126	127

Struktura kodu szblonów

- **Proces instancjacji wymaga, aby wszystkie definicje klas i funkcji były widoczne w danej jednostce kompilacji.** Dotyczy to zarówno metod inline i standardowych metod klas. Wzorzec funkcji może być zdefiniowany po wystąpieniu odwołanie jednakże wcześniej musi być widoczna deklaracja prototypu funkcji.
- **Równocześnie, w danej jednostce kompilacji nie mogą być widoczne dwie definicje szablonu klasy lub definicje funkcji.** (Prototypy funkcji i deklaracje klas bez podania ich definicji mogą być podane wielokrotnie.)
- Kompilator i linker generują zawsze **unikalny kod instancji klas i funkcji**, nawet, jeżeli proces instancjacji zachodzi w różnych jednostkach kompilacji. Oznacza to, że zostanie wygenerowana dokładnie jedna instancja klasy `Vector<int>`, nawet, jeżeli była ona wykorzystywana w dwóch różnych modułach.

Struktura kodu szblonów

Typowym rozwiązaniem jest umieszczenie jednej lub kilku definicji szablonek w pliku nagłówkowym oraz zabezpieczenie ich przed wielokrotnym włączeniem do danej jednostki kompilacji za pomocą instrukcji preprocesora:

```
#if !defined _Vector_h_
#define _Vector_h_
//deklaracja parametryzowanej klasy Vector<T>
//definicje parametryzowanych metod klasy Vector<T>
//definicja parametryzowanej klasy Iterator<T>
// inne definicje funkcji
#endif
```

Błędy instancjacji szablonów

- Kłopotliwą cechą szablonów jest to, że błędy raportowane są dopiero w momencie instancjacji (czyli generacji funkcji lub klas dla konkretnych wartości parametrów).
- Proces skanowania definicji szablonów przed ich instancjacją ogranicza się w zasadzie do sprawdzenia poprawnej liczby nawiasów otwierających i zamykających, stąd możliwe jest skompilowanie kodu, w którym pojawi się błędna definicja klasy lub funkcji:

```
template<class T>
void foerr()
{
    if;
}
```

Błędy instancjacji szablonów

- Kompilator raportuje także błędy instancjacji, jeżeli typ/klasa będąca parametrem nie realizuje pewnego oczekiwanego interfejsu.

```
template <class T>
int compare(const T&a,const T&b){
    if(a<b)return -1;
    if(a==b)return 0;
    return 1;
}

class C{};

int main()
{
    cout<<compare<int>(1,2)<<endl; //ok
    cout<<compare<C>(C(),C()); // error
}
```

Szablony i dziedziczenie

Klasy wygenerowane przez instancjację szblonów mogą stać się klasami bazowymi dla klas zdefiniowanych przez użytkownika.

```
template <class T,  
         int size=100>  
class A {  
public:  
    virtual void foo()=0;  
    T buf[size];  
};
```

```
class B:public A<int>  
{  
public:  
    void foo(){  
        cout<<"B::foo()"<<endl;};  
};  
  
int main(){  
    A<int>*ptr=new B();  
    ptr->foo();  
}
```

Relacja dziedziczenia pomiędzy parametrami szablónów klas nie przenosi się jednak na dziedziczenie instancji wygenerowanych klas:

- `Vector<Linestring>` nie dziedziczy po `Vector<GeoObject>`;
- `Vector<Linestring*>` nie dziedziczy po `Vector<GeoObject*>`

Kontener zawierający wskaźniki

- Parametrem instancjacji szablonów może być typ wskaźnikowy

```
void test_ptrs(){  
    Vector<const char*> txts;  
    txts.pushBack("Stoi na stacji");  
    txts.pushBack("Lokomotywa");  
    txts.pushBack("I pot z niej spływa");  
    txts.pushBack("Tłusta oliwa");  
    for(Iterator<const char*> it(txts);it.good();++it){  
        cout<<it.get()<<endl;  
    }  
}
```

Stoi na stacji
Lokomotywa
I pot z niej spływa
Tłusta oliwa

W tym przypadku parametrem są wskaźniki do tekstów umieszczonych w pamięci statycznej (bloku danych).

Kontener zawierający wskaźniki

Częściej będą to wskaźniki do obiektów, dla których pamięć przydzielana jest na stercie.

Funkcja `str_alloc` sporządza kopię tekstu na stercie:

- Oblicza długość tekstu i alokuje pamięć
- Kopiuje tekst

```
char*str_alloc(const char*t){
    char*ptr = new char[strlen(t)+1];
    strcpy(ptr,t);
    return ptr;
}
```

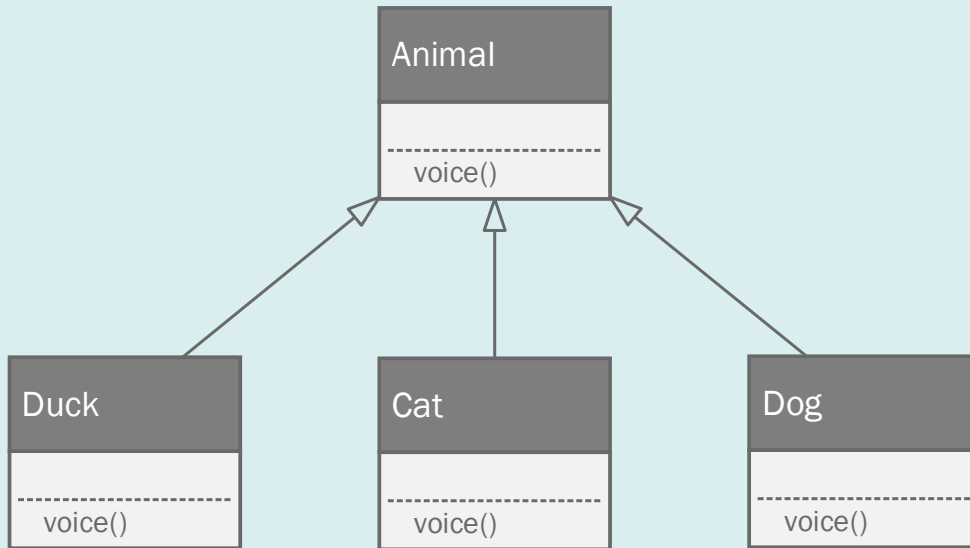
Kontener zawierający wskaźniki

```
void test_alloc_ptrs(){
    Vector<char*> txts;
    txts.pushBack(str_alloc("Stoi na stacji"));
    txts.pushBack(str_alloc("Lokomotywa"));
    txts.pushBack(str_alloc("I pot z niej spływa"));
    txts.pushBack(str_alloc("Tłusta oliwa"));
    for(Iterator<char*> it(txts);it.good();++it){
        cout<<it.get()<<endl;
    }

    // Konieczne zwolnienie pamięci

    for(Iterator<char*> it(txts);it.good();++it){
        delete it.get();
    }
}
```

Hierarchia klas



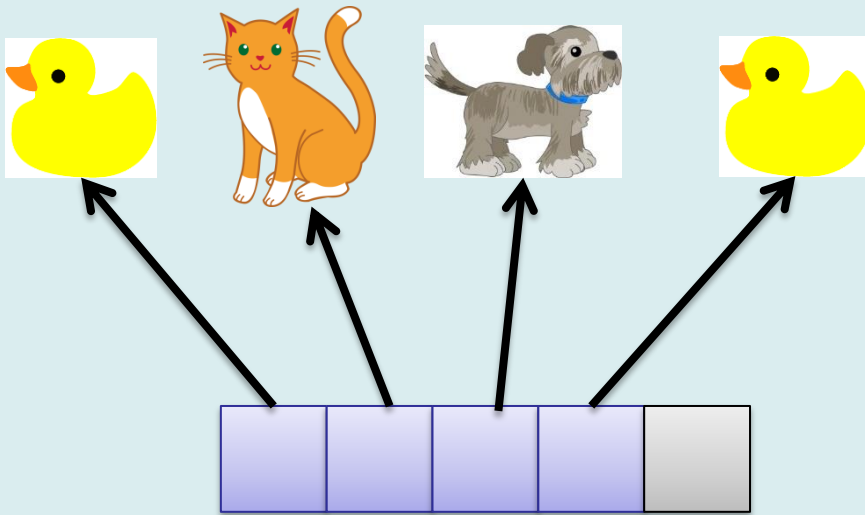
```
class Animal{
public:
    virtual void voice()=0;
};
```

```
class Dog: public Animal{
public:
    virtual void voice(){
        cout<<"hau-hau ";
    }
};
```

```
class Duck : public Animal{
public:
    virtual void voice(){
        cout<<"kwak-kwak ";
    }
};
```

```
class Cat: public Animal{
public:
    virtual void voice(){
        cout<<"miau-miau ";
    }
};
```

Kontener zawierający wskaźniki



- Zoo jest klasą potomną `Vector<Animal*>`. Przechowuje wskaźniki do obiektów, dla których pamięć jest przydzielana na stercie.
- Zoo jest właścicielem obiektów.
- **Minimalne wymagania to implementacja destruktora.**

```
class Zoo:public Vector<Animal*>{  
public:  
    ~Zoo(){  
        for(int i=0;i<getSize();i++)  
            delete (*this)[i];  
    }  
};
```



Kontener zawierający wskaźniki

Przeddefiniowując funkcje `free()` i `copy()` można również zapewnić poprawną implementację konstruktora kopiującego i operatora przypisania.



Wersja 2

```
class Zoo:public Vector<Animal*>{
protected:
    void free();
    void copy(const Zoo&other);
public:

};

void Zoo::free(){
    for(int i=0;i<getSize();i++)delete (*this)[i];
    Vector<Animal*>::free();
}
```

Kontener zawierający wskaźniki

```
void Zoo::copy(const Zoo&other){
    Vector<Animal*>::copy(other);
    for(int i=0;i<getSize();i++){
        Animal*a = (*this)[i];
        if(typeid(*a)==typeid(Duck)){
            a=new Duck(*(Duck*)a);
        }
        else if(typeid(*a)==typeid(Cat)){
            a=new Cat(*(Cat*)a);
        }
        else if(typeid(*a)==typeid(Dog)){
            a=new Dog(*(Dog*)a);
        }
        else throw -1;
        (*this)[i]=a;
    }
}
```

Operator typeid pozwala na sprawdzenie, czy obiekt należy do danej klasy.

Jeżeli a jest kaczką (Duck) to tworzona jest kopia obiektu przez wywołanie konstruktora kopiującego. To samo dla innych typów.

Test

```
void test_zoo(){
    Zoo zoo;
    zoo.pushBack(new Duck());
    zoo.pushBack(new Cat());
    zoo.pushBack(new Dog());
    zoo.pushBack(new Duck());
    for(Iterator<Animal*> it(zoo);it.good();++it){
        it.get()->voice();
    }
    cout<<endl;

    Zoo anotherZoo;
    anotherZoo = zoo;
    anotherZoo.pushBack(new Cat());
    for(Iterator<Animal*> it(anotherZoo);it.good();++it){
        it.get()->voice();
    }
    cout<<endl;
}
```

kwak-kwak miau-miau hau-hau kwak-kwak
kwak-kwak miau-miau hau-hau kwak-kwak miau-miau