Język C++

część 9
szablony klas

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- szablony funkcji
  - były omawiane na wcześniejszym wykładzie …
  - nazewnictwo: szablon funkcji <-> wzorzec funkcji

- szablony klas
  - szablony klas umożliwiają definiowanie ogólnej klasy czyli mogącej przetwarzać dane różnego typu (tzw. parametryzacja typów)
  - użyteczność takich klas jest szczególnie widoczna przy implementacji tzw. kontenerów. Intensywnie używa ich biblioteka STL
  - obecnie pokazane zostaną zasady tworzenia klas szablonowych
  - przykład:
    zmodyfikujemy wielokrotnie używaną wcześniej klasę `Stack`, aby możliwe było obsługiwanie stosów nie tylko elementów całkowitych `int`

- szablony klas a dziedziczenie
  - szablony klas to inne podejście do idei tzw. powtórnego użycia kodu (ang. *code reuse*). Zamiast dziedziczenia (powtórne użycie kodu obiektowego), wzorce powtórnie używają kodu źródłowego

- szablony klas
  - wersja **pierwotna** klasy Stack

```
#include <iostream.h>
enum stack_state {OK, FULL, EMPTY};

class StackNew {
public:
    StackNew( int _size = 10 );
    ~StackNew() { delete [] table; }
    void push( int );
    int pop();
    void print();
    int ile_elementow();
    int get_size() { return size; }
private:
    int size;
    int* table;
    int top;
    stack_state state; };

StackNew::StackNew( int _size )
{
    size = _size;
    table = new int[ size ];
    top = 0;
    state = EMPTY;
}                                   cdn.
```

- szablony klas

```
void StackNew::push( int _element )
{
        ...
}

int StackNew::pop()
{
        ...
}

void StackNew::print()
{
        ...
}

inline int StackNew::ile_elementow()
{
        ...
}                                   cdn.
```


- szablony klas

```
void main()
{
    StackNew s1( 5 );

    s1.push( 5 );
    s1.push( 10 );
    s1.push( 15 );

    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    s1.print();

    cout << "Pop 1 = " << s1.pop() << endl;
    cout << "Pop 2 = " << s1.pop() << endl;
    cout << "Pop 3 = " << s1.pop() << endl;
    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    cout << "Pop 4 = " << s1.pop() << endl;

    for(int i = 0; i < s1.get_size()+1; i++ )
    {
        cout << "Push : " << i << endl;
        s1.push( i );
    }

    s1.print();
    cout << "Liczba elementow = "<< s1.ile_elementow() << endl;
}
```

- szablony klas – klasa i obiekty
  - wersja **szablonowa** klasy `StackNew` (wprowadzono niewielkie zmiany w stosunku do oryginału)
  - składnia jest bardzo podobna do tej używanej w szablonach funkcji

```
enum stack_state {OK, FULL, EMPTY};

template<class T> class StackNew
{
public:
    StackNew( int _size = 10 );
    ~StackNew() { delete [] table; }
    bool push( T );
    bool pop( T& );
    void print();
    int ile_elementow();
    int get_size() { return size; }
private:
    int size;
    T* table;
    int top;
    stack_state state;
};
```

uwaga dotycząca nazewnictwa: `StackNew` to **kontener** do przechowywania danych różnych typów

Ogólny typ `T` (nazwa `T` oczywiście dowolna)

bool

puch, pop: inaczej niż w wersji "non-template"

```
// Przyjęło się formatować raczej tak:

template<class T>
class StackNew      // od nowej linii
{
...
```

  - obiekty klasy szablonowej

```
StackNew<int>    s1( 5 );          // wymamiaj "StosNew typu int"
StackNew<long>   s2( 7.4 );
StackNew<double> s3( 15.6 );
```


- szablony klas – metody składowe klasy szablonowej
  - składnia jest dosyć skomplikowana ale logiczna

```
template<class T> StackNew<T>::StackNew( int _size ) {
    size = _size;
    table = new T[ size ];
    top = 0;
    state = EMPTY;  }

template<class T> bool StackNew<T>::push( T _element )  {
    if( state != FULL )
         table[ top++ ] = _element; return true;
    else
         cout << "*** Pelny stos ! ***" << endl;
    if( top >= size ) state = FULL;  return false; else  state = OK;  return true;
}

template<class T> bool StackNew<T>::pop( T& _element)  {
    if( state != EMPTY )
         _element = table[ --top ]; return true;
    else
         cout << "*** Pusty stos ! ***" << endl;

    if( top <= 0 ) state = EMPTY;  return false; else  state = OK. return true ; }

template<class T> void StackNew<T>::print()  {
    for( int i = top-1; i >= 0; i-- )
          cout << "[ " << table[ i ] << " ]" << endl;  }

template<class T> int StackNew<T>::ile_elementow() {
    return top; }
```

typ wbudowany

oczywiście można je mieszać ze sobą

typ sparametryzowany

- szablony klas – konkretyzacja klasy szablonowej
  - test stosu dla różnych typów danych. **konkretyzacja** klasy `StackNew<T>`

```
void main() {

    StackNew<int> s1( 5 ); // ---------- Stos danych typu int.

    s1.push( 5 );
    s1.push( 10 );

    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    s1.print();

    cout << "Pop 1 = " << s1.pop() << endl;
    cout << "Pop 2 = " << s1.pop() << endl;
    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    cout << "Pop 3 = " << s1.pop() << endl;

    StackNew<long> s2( 5 ); // ---------- Stos danych typu long.

    s2.push( 50000L );
    s2.push( 1000000L );

    cout << "Liczba elementow = " << s2.ile_elementow() << endl;
    s2.print();

    cout << "Pop 1 = " << s2.pop() << endl;
    cout << "Pop 2 = " << s2.pop() << endl;
    cout << "Liczba elementow = " << s2.ile_elementow() << endl;
    cout << "Pop 3 = " << s2.pop() << endl;
```


- szablony klas – konkretyzacja klasy szablonowej
  - klasę szablonową można konkretyzować za pomocą innej klasy (tam gdzie ma to sens ze względu na operacje na danych obiektach)
  - podobnie postępowaliśmy z szablonami funkcji

```
void main() {
    StackNew<Zespolona> s1( 5 ); // ---------- Stos danych typu Zespolona.

    s1.push( Zespolona( 5, -1 ) );
    s1.push( Zespolona( 10, -2 ) );

    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    s1.print();

    cout << "Pop 1 = " << s1.pop() << endl;
    cout << "Pop 2 = " << s1.pop() << endl;
    cout << "Liczba elementow = " << s1.ile_elementow() << endl;
    cout << "Pop 3 = " << s1.pop() << endl;

    StackNew<String> s2( 5 ); // ---------- Stos danych typu String.

    s2.push( "Pierwszy test" );
    s2.push( "Drugi test" );

    cout << "Liczba elementow = " << s2.ile_elementow() << endl;
    s2.print();

    cout << "Pop 1 = " << s2.pop() << endl;
    cout << "Pop 2 = " << s2.pop() << endl;
    cout << "Liczba elementow = " << s2.ile_elementow() << endl;
    cout << "Pop 3 = " << s2.pop() << endl;   }
```

klasę `Zespolona` pokazano na wcześniejszych wykładach

klasę `String` pokazano na wcześniejszych wykładach

- szablony klas – zastosowanie szablonów funkcji
  - na dwóch poprzednich stronach kody testujące różne stosy były bardzo podobne
  - podobieństwo to "obsłużymy" szablonem funkcji (znamy już je z poprz. wykładów)

wzorzec funkcji

... funkcja testująca

```
template< class T > void testStack(
   StackNew< T > &theStack,          // reference to Stack< T >
   T value,                          // initial value to push
   T increment,                      // increment for subsequent values
   const char *stackName )           // name of the Stack < T > object

{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) {
      cout << value << ' ';
      value += increment;
   }

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";
}
                              c.d.n.
```

... dlatego `push` zwraca **bool**

- szablony klas – zastosowanie szablonów funkcji
  - ... i używamy szablonu funkcji testującej różne stosy

```
int main()
{
   StackNew< double > doubleStack( 5 );    // obiekt #1
   StackNew< int >    intStack;            // obiekt #2

   testStack( doubleStack, 1.0, 0.5, "doubleStack" );
   testStack( intStack, 1, 3, "intStack" );

   return 0;
}
```

- szablony klas – uwagi dotyczące notacji
  - wersja inline

```
#include <iostream>
using namespace std;

template<class T> class Array        // nasza "inteligentna" tablica
{
private:
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index)           // wersja inline operator[]
  {
    if (index >= 0 && index < size)
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
}
```


- szablony klas – uwagi dotyczące notacji
  - wersja inline nie zawsze będzie odpowiednia, więc …

```
#include <iostream>
using namespace std;

template<class T>class Array
{
private:
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>                    // wersja non-inline operator[]
T& Array<T>::operator[](int index)
{
  if (index >= 0 && index < size)
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
}
```

porównaj z poprzednią wersją

formatowanie ułatwiające czytanie

- szablony klas – iteratory
  - o iteratorach często mówi się jako o tzw. "bystrych wskaźnikach" (ang. *smart pointers*). Trudno przy ich użyciu na przykład "wyjść poza obiekt"
  - iterator to (prawie) zwykły obiekt, który "przesuwa się" wzdłuż danego kontenera obiektów wybierając jeden z nich. Co ważne, dostęp do implementacji danego kontenera nie jest wymagany (gdyż będziemy używali klas zaprzyjaźnionych)
  - terminologia obiektowa:
    iterator to obiekt do przemieszczania się po kontenerze obiektów

- szablony klas – iteratory
  - najprostszy przykład iteratora.
  - Zmiany w deklaracji klasy (na razie nie-szablonowej)

```
#include <iostream.h>

enum stack_state {OK, FULL, EMPTY};

class StackNew
{
public:
    StackNew( int _size = 10 );
    ~StackNew() { delete [] table; }
    void push( int );
    int pop();
    void print();
    int ile_elementow();
    int get_size() { return size; }
    friend class StackNewIter;
private:
    int size;
    int* table;
    int top;
    stack_state state;
};
```

`StackNewIter`, zaprojektowano aby działało tylko ze `StackNew`

klasa zaprzyjaźniona z klasą `StackNew`. To dodajemy.

- szablony klas – iteratory
  - najprostszy przykład iteratora
  - implementacja `StackNew`**`Iter`**. Ograniczamy się jedynie do "iterowania" w przód (przeciążamy tylko operator ++)

```
class StackNewIter
{
  StackNew& s;
  int index;
public:
  StackNewIter ( StackNew& is ) : s(is), index(0) { } // konstruktor

  int operator++()                  // Prefix
  {
    if (index < s.top) cout <<  "iterator moved out of range";
    else
    return s.table[ ++index ];
  }

  int operator++( int )             // Postfix
  {
    if (index < s.top) cout <<  "iterator moved out of range";
    else
    return s.table[ index++ ];
  }
};
```


- szablony klas – iteratory
  - Testowanie iteratora

```
int main()

{
  StackNew MyStack;                          // stos (kontener)

  for(int i = 0; i < 20; i++)
    MyStack.push( i );

  StackNewIter iter( MyStack );    // iterator (obiekt) iter
                                   // kontenera MyStack
                                   // przekazanego jako parametr
  for(int j = 0; j < 20; j++)
    cout << iter++ << endl;
}
```

przeciążanie operatora 1-argumentowego.
Patrz poprzedni wykład

- szablony klas – iteratory
  - Załóżmy, że dla każdego kontenera jaki utworzymy chcemy dodać **iterator**. Najwygodniej, aby każdy deklarowany w programie iterator był po prostu klasy **iterator**. Jak wyeliminować jednak konflikty nazw, które się pojawią?
  - Użyjemy zagłębionej klasy (ang. nested class). Tak też robi się w bibliotece STL

Uwaga! Dalszy materiał jest trudny !

```
class StackNew
{
public:
    StackNew( int _size = 10 );
    ~StackNew() { delete [] table; }
    void push( int );
    int pop();
    void print();
    int ile_elementow();
    int get_size() { return size; }
    class iterator;
    friend class iterator;                    // nested friend class
    class iterator
    {
    // kod iteratora (patrz następna strona)
    };
    iterator begin() { return iterator(*this); }
    iterator end()   { return iterator(*this, true); } // Create the "end sentinel":

private:
    int size;
    int* table;
    int top;
    stack_state state;
};
```

When making a nested friend class, you must go through the process of first declaring the name of the class, then declaring it as a friend, then defining the class. Otherwise, the compiler will get confused.

używamy mechanizmu zagłębiania w sobie klas

patrz następna strona

- szablony klas – iteratory
  - implementacja "zagłębionego" iteratora. Dodano kilka **nowych** elementów

```
private:                        // kod iteratora
  StackNew& s;
  int index;
public:
    iterator ( StackNew& is ) : s(is), index(0) {}            // jak poprzednio
    // To create the "end sentinel" iterator:
    iterator ( StackNew& is, bool)  : s(is), index(s.top) {} // konstruktor #2
    int current() const { return s.table[ index ]; }

    int operator++() {          // Prefix
      if (index < s.top) cout <<  "iterator moved out of range"; else
      return s.table[ ++index ];  }

    int operator++(int) {       // Postfix
      if (index < s.top) cout <<  "iterator moved out of range"; else
      return s.table[ index++ ];   }

    iterator& operator+= ( int amount ) { // iterator forward
      if ( index + amount < s.top ) {cout << "StackNew::iterator::operator+=()";
                                      cout << "tried to move out of bounds";} else
      index += amount;
      return *this;     }

     bool operator==(const iterator& r) const { // To see if you're at the end:
       return index == r.index;  }

     bool operator!=(const iterator& r) const {
       return index != r.index;   }

     friend ostream& operator<<(ostream& os, const iterator& it) {
       return os << it.current();  }
```

2 kwalifikatory gdyż "nested class" !

- szablony klas – iteratory
  - Testowanie "zagłębionego" iteratora

```
int main()
{
  StackNew MyStack;

  for(int i = 0; i < 20; i++)
    MyStack.push( i );

  cout << "Traverse the whole StackNew\n";
  StackNew::iterator iter = MyStack.begin();

  while(iter != MyStack.end())
    cout << iter++ << endl;

  cout << "Traverse a portion of the StackNew\n";
  StackNew::iterator start = MyStack.begin()
  StackNew::iterator end   = MyStack.begin();


  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;

  while(start != end)
    cout << start++ << endl;
}
```

2 kwalifikatory gdyż "nested class" !

dwa iteratory (obiekty). Oba ustawione na starcie na początku kontenera

- szablony klas – iteratory

komentarz do przykładu z poprzedniej strony:

  - Funkcja `current( )` zwraca aktualny element, na który "wskazuje" iterator
  - Dzięki operatorowi `+=` można "iteraować" co wybraną liczbę elementów
  - Operatory `==` oraz `!=` porównują dwa iteratory (obiekty klasy `StackNew::iterator`). W praktyce służą one do testowania osiągnięcia końca kontenera. Porównaj deklaracje iteratorów `start` i `end` (często nazywanego w literaturze "end sentinel").
  - Iteratory `start` i `end` tworzone są z użyciem funkcji `begin()` i `end()`
  - `begin()` korzysta z pierwszej wersji konstruktora klasy `iterator` (w którym pole `index` ustawiane jest na 0)
  - `end()` korzysta z drugiej wersji konstruktora klasy `iterator`. (w którym pole `index` ustawiane jest na wartość `top`). Drugi parametr tego konstruktora (`bool`) to więc tylko atrap, służąc odróżnieniu od siebie obu konstruktorów.

- szablony klas – iteratory
  - Kolejna modyfikacja mogła by polegać na napisaniu analogicznych wersji iteratorów ale zagłębionych w klasie **szablonowej** (gdyż omawiane na kilku poprzednich stronach zagadnienia ilustrowaliśmy na stosie int-ów)
  - pomijamy jednak to zagadnienie

- szablony klas – kontenery (standardowe)
  - Kontenery (pojemniki) są to klasy sparametryzowane (szablony) służące do przechowywania i udostępniania danych. Kontenery standardowe zdefiniowano w normie ANSI 1998:
  - poniżej tylko ogólne spojrzenie na kontenery standardowe (ANSI 1998):

```
vector              - tablica rozszerzalna, dostęp do elementów - przez indeks.
vector<bool>        - specjalna wersja kontenera, operująca na bitach pamięci.
list                - lista elementów; efektywne wstawianie i usuwanie.
stack               - stos; dostęp tylko do wierzchołka stosu
queue               - kolejka; wstawianie na końcu, usuwanie z początku.
deque               - kolejka dwustronna; wstawianie/usuwanie z obu końców.
priority queue      - kolejka do operacji na dużych elementach.
set                 - brak kolejności; operacje zawierania, dołączania i usuwania.
multiset            - j.w. z powtarzającymi się elementami.
bitset              - jak set; zoptymalizowany sposób przechowywania wartości binarnych
map                 - dostęp do wartości poprzez unikatowy klucz
multimap            - j.w. - ale klucz może się powtarzać.
string              - przechowywanie i operacje na łańcuchu.
```

- szablony klas – algorytmy (standardowe)
  - Algorytmy są sparametryzowanymi funkcjami realizującymi typowe operacje na kontenerach
  - Dostępne są przez plik nagłówkowy `<algorithm>`. Przykłady z biblioteki standardowej:

```
find()              - znajduje element odpowiadający argumentowi
find_if()           - znajduje element spełniający podany warunek
max_element()       - znajduje wartość maksymalną w podanym zakresie
min_element()       - znajduje wartość minimalną w podanym zakresie
reverse()           - odwraca kolejność elementów
replace()           - zastępuje podane wartości nową wartością
next_permutation() - generuje permutację elementów
random_shuffle()    - miesza (tasuje) elementy
count()             - zlicza elementy odpowiadające podanej wartości
for_each()          - stosuje podaną funkcję do każdego elementu.
```


- szablony klas – kontener vector
  - Klasa jest dostępna poprzez plik nagłówkowy `<vector>` .
  - Przykłady konkretyzacji klasy:

```
vector<double> x;         // pusta tablica
vector<int> a(10, 0);     // tab. 10-el. wypełniona zerami
```

  - Wybrane metody klasy `vector`:

```
size     - zwraca liczbę elementów w wektorze
insert   - wstawia nowy element przed podaną pozycję
push_back - wstawia nowy element na koniec tablicy
erase    - usuwa element z podanej pozycji
clear    - usuwa wszystkie elementy z wektora.
```

  - Do wektora można zastosować też funkcje realizujące różnego typu algorytmy (sortowanie, odwracanie kolejności, permutacje itp..)

- lista (zagadnień) nieobecnych
  - Na omówienie tych zagadnień nie starczyło już czasu :-(
  - **biblioteka `STL`**. Została jedynie zasygnalizowana
    + podano bardzo elementarne przykłady
    `http://www.sgi.com/tech/stl/`
    `(Standard Template Library Programmer's Guide)`
    `http://www.msoe.edu/eecs/cese/resources/stl/`
    `http://pegasus.rutgers.edu/~elflord/cpp/stdlib/`
    `http://yolinux.com/TUTORIALS/LinuxTutorialC++STL.html`
  - **strumienie wej/wyj, przetwarzanie plików**
    (w przykładach używaliśmy jedynie naprawdę podstawowych elementów związanych z wej / wyj i plikami)
  - **obsługa wyjątków**
    (słowa kluczowe `try, throw, catch`
    niespodziewane wyjątki
    hierarchia wyjatków)
  - **preprocesor**
  - **klasa string** i przetwarzanie napisów
  - …