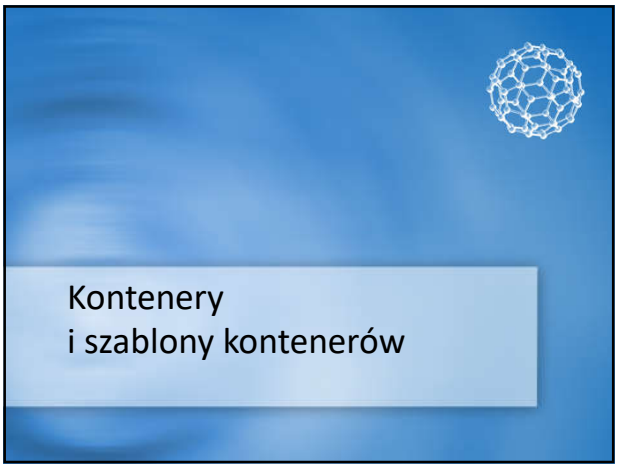

161

C++ - szablony kontenerów

Kontenery

- Kontener (lub inaczej pojemnik, ang. *container*, *collection*) to struktura danych, której zadaniem jest przechowywanie w zorganizowany sposób zbioru innych danych (obiektów). Kontener zapewnia wygodne narzędzia dostępu, w tym dodawanie, usuwanie i wyszukiwanie danej (obiektu) w kontenerze.
- Oczywistym kontenerem jest tablica (*ale prymitywnym*).
- Inne rodzaje kontenerów to: lista, drzewo, stos..

162

C++ - szablony kontenerów

Przykład – szablon kontenera typu „stos”

```
template<typename T, int rozm>
    class SzablonStosu {
      T stos[rozm];
      int top;
    public:
      SzablonStosu() : top(0) {}
      void push(const T& i) {
        stos[top++] = i;
      }
      T pop() {
        return stos[--top];
      }
    };
```


163

C++ - szablony kontenerów

Przykład zastosowania kontenera stosu:

```
int main(int argc, char *argv[])
{
      SzablonStosu<int, 100> is;
      // Dodanie kilku liczb:
      for(int i = 0; i < 20; i++)
        is.push(i);

       // Zdjęcie ich ze stosu i jednoczesne
       // wypisanie na ekranie:
      for(int k = 0; k < 20; k++)
        printf("%i\n", is.pop());
      …
```


164

C++ - szablony kontenerów

Efektywność kontenerów

Jeżeli chcielibyśmy przygotować się na znaczną liczbę obiektów:

1. z których każdy miałby mieć przypisany swój stos,
2. przy czym tworząc struktury do zarządzania tymi obiektami, musielibyśmy również od razu tworzyć komplet pustych stosów gotowych do wykorzystania,

to w momencie uruchomienia nastąpiłoby zajęcie dużej ilości pamięci.

A jeżeli z czasem okazałoby się, że obiektów do przechowywania jest niewiele, to duża część pamięci zajętej przez stosy pozostałaby niewykorzystana. ☹

165

C++ - szablony kontenerów

Efektywność kontenerów

- Cel: taki szablon stosu, który pozwalałby na tworzenie kontenerów zajmujących początkowo niewiele pamięci. Kontener osiągałby swoje właściwe rozmiary dopiero w momencie pierwszego umieszczenia danych na stosie.
- Najprościej – można opakować przykładowy szablon w inny szablon, działający tak, aby uniknąć tworzenia od razu całej struktury danych.

166

C++ - szablony kontenerów

```
template<typename T, int rozmiar>
    class SzablonStosu {
      T stos[rozmiar];
      int top;
    public:
      SzablonStosu() : top(0) {}
      void push(const T& i) {
        stos[top++] = i;
      }
      T pop() {
        return stos[--top];
      }
      template<typename U, int rozm>
    friend class SzablonSprytnegoStosu;
     };
```

```
template<typename T, int rozmiar>
   class SzablonSprytnegoStosu {
     SzablonStosu<T, rozmiar> *sp;
   public:
     SzablonSprytnegoStosu(): sp(0) { };
     void push(const T& i) {
       if (!sp)
          sp = new SzablonStosu<T,rozmiar>;
       sp->push(i);
     };
     T pop() {
       if (sp->top)
         return sp->pop();
       else
           return T();
     };
     ~SzablonSprytnegoStosu() {
       delete sp;
     };
   };
```


167

C++ - szablony kontenerów

Wykorzystanie przykładowego sprytnego stosu

```
SzablonSprytnegoStosu<int, 100> is;
// minimalna alokacja pamięci przy tworzeniu obiektu-kontenera

// Dodanie na stos kilku liczb:
for(int i = 0; i < 20; i++)
   is.push(i);  // w pierwszym wywołaniu - właściwa alokacja pamięci

// Zdjęcie liczb ze stosu i wypisanie na ekranie:
for(int k = 0; k < 20; k++)
   cout << is.pop() << endl;
```


168

C++ - szablony kontenerów

Szablony i wykorzystanie relacji przyjaźni

Przyjmijmy, że mamy funkcję lub szablon funkcji oraz klasę lub szablon klasy. Są cztery możliwe przypadki relacji między nimi:

- **Jeden-do-wielu**: Funkcja może być przyjacielem dla wszystkich instancji szablonu klasy (wszystkich klas, które zostały utworzone z szablonu)
- **Wielu-do-jednego**: Wszystkie instancje szablonu funkcji są przyjaciółmi regularnej klasy
- **Jeden-do-jednego**: Instancja szablonu funkcji dla określonego zbioru parametrów jest przyjacielem instancji klasy z takim samym zbiorem parametrów (identycznie jak w przypadku regularnych funkcji i klasy)
- **Wielu-do-wielu**: Wszystkie potencjalne instancje szablonu funkcji mogą być przyjaciółmi wszystkich potencjalnych instancji szablonu klasy

169

C++ - szablony kontenerów

Przykład relacji między szablonami i klasami lub funkcjami

```
template<typename R>          // 1. szablon funkcji
int g() { .. };
template<typename S>          // 2. szablon funkcji
int h() { .. };
class B{                      // 3. klasa
   template<typename V>
   friend int g();
};
template<typename T>          // 4. szablon klasy
class A {
   friend int e();
   friend int f(T);    // działa tylko gdy: T==double (!)
   friend int g<T>();
   template<typename U>
   friend int h();
};
int e() { .. };               // 5. funkcja
int f(double) { .. };         // 6. funkcja
```


170

C++ - szablony kontenerów

```
template<typename R>
int g() { .. };
template<typename S>
int h() { .. };
class B{
   template<typename V>
   friend int g();
};
template<typename T>
class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<typename U>
   friend int h();
};
int e() { .. };
int f(double) { .. };
```

jeden-do-wielu:
funkcja **e** i szablon klasy **A**

171

C++ - szablony kontenerów

```
template<typename R>
int g() { .. };
template<typename S>
int h() { .. };
class B{
   template<typename V>
   friend int g();
};
template<typename T>
class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<typename U>
   friend int h();
};
int e() { .. };
int f(double) { .. };
```

wielu-do-jednego:
szablon funkcji **g** i klasa **B**

172

C++ - szablony kontenerów

```
template<typename R>
int g() { .. };
template<typename S>
int h() { .. };
class B{
   template<typename V>
    friend int g();
};
template<typename T>
class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<typename U>
   friend int h();
};
int e() { .. };
int f(double) { .. };
```

jeden-do-jednego:
funkcja **f** i szablon klasy **A**, oraz
szablon funkcji **g** i szablon klasy **A**

173

C++ - szablony kontenerów

```
template<typename R>
int g() { .. };
template<typename S>
int h() { .. };
class B{
   template<typename V>
    friend int g();
};
template<typename T>
class A {
   friend int e();
   friend int f(T);
   friend int g<T>();
   template<typename U>
   friend int h();
};
int e() { .. };
int f(double) { .. };
```

wielu-do-wielu:
szablon funkcji **h** i szablon klasy **A**

174

C++ - szablony kontenerów

```
template<typename T, int rozmiar>
    class SzablonStosu {
      …

   // 1. relacja: wielu-do-wielu
      template<typename U, int rozm>
   friend class SzablonSprytnegoStosu;

   albo..

   // 2. relacja jeden-do-jednego:
   friend class
   SzablonSprytnegoStosu<T, rozmiar>;
};
```

```
template<typename T, int rozmiar>
   class SzablonSprytnegoStosu {
     public:
     SzablonStosu<T, rozmiar> *sp;
     SzablonSprytnegoStosu(): sp(0) { };
     void push(const T& i) {
       if (!sp)
          sp = new SzablonStosu<T,rozmiar>;
       sp->push(i);
     };
     T pop() {
       if (sp->top)
         return sp->pop();
       else
           return T();
     };
     ~SzablonSprytnegoStosu() {
       delete sp;
     };
};
```


175

C++ - szablony kontenerów

Kontenery i prawo własności obiektów

1. **Kontenery przechowujące obiekty (w postaci wartości)**
   są bezwzględnie właścicielami tych obiektów. Autorzy kodu kontenerów mają obowiązek napisać kod tak, aby przy usuwaniu instancji kontenerów usunąć uprzednio wszelkie obiekty przechowywane w kontenerze.
2. **Kontenery przechowujące tylko wskaźniki do obiektów**
   muszą mieć ustalone reguły własności obiektów wskazywanych przez te wskaźniki (po czyjej stronie spoczywa obowiązek usunięcia zawartości kontenera przy okazji usuwania całego kontenera – po stronie kontenera czy programisty).

Nie ma Garbage Collectora, wiec nie można udawać, że raz zajęta pamięć jakoś potem sama się zwolni.. ☹

176

C++ - szablony kontenerów

Kontenery i prawo własności obiektów

1. **Kontenery przechowujące obiekty (w postaci wartości)**
2. **Kontenery przechowujące wskaźniki do obiektów**

Najlepiej pisać kod wariantowo, tak aby programista sam np. przy wywołaniu konstruktora kontenera określał w argumencie wywołania reguły własności

Można też wprowadzić atrybuty własności dla obiektów i dodać usługi zmieniające stan tych atrybutów

177

Szablony kontenerów

Iteratory

178

C++ - szablony kontenerów

Iteratory

Iterator jest obiektem, który porusza się w obrębie kontenera po przechowywanych w nim obiektach (podobnie jak wskaźnik po elementach listy).

Udostępnia on za każdym razem pojedynczy, zawarty w kontenerze obiekt, nie zapewnia jednak bezpośredniego dostępu do implementacji tego kontenera (do metod i pól).

Iteratory stanowią standardowy sposób dostępu do elementów.

Klasy iteratorów najczęściej są tworzone wraz z klasami kontenerów.

C++ - szablony kontenerów

Iteratory

Mówiąc potocznie, iteratory są „sprytnymi wskaźnikami”, jednak w odróżnieniu od prawdziwych wskaźników są bezpieczne w użyciu – dużo trudniej jest doprowadzić do sytuacji odpowiadającej np. przekroczeniu zakresu indeksów w tablicy.

Tradycyjnie *obiekt-iterator* jest tworzony za pomocą takiego konstruktora, który przyłącza go do pojedynczego *obiektu-kontenera*.

W czasie swojego życia obiekt-iterator nie jest zazwyczaj przyłączany do żadnego innego kontenera.

C++ - szablony kontenerów

Przykład stosu i jego iteratora

```
class IntStos {
    enum { rozm = 100 };
    int Stos[rozm];
    int top;
  public:
    IntStos() : top(0) {}
    void push(int i) {
      assert(top < rozm);
      Stos[top++] = i;
    }
    int pop() {
      assert(top > 0);
      return Stos[--top];
    }
    friend class IntStosIter;
};
```

```
class IntStosIter { // klasa iteratora
    IntStos& s;
    int index;
  public:
    IntStosIter(IntStos& is):s(is),index(0)
        { };
    int operator++() {            // Prefix
      assert(index < s.top);
      return s.Stos[++index];
    };
    int operator++(int) {         // Postfix
      assert(index < s.top);
      return s.Stos[index++];
    };
};
```


C++ - szablony kontenerów

Korzystanie z kontenera i iteratora:

```
int main(int argc, char *argv[])
{
    IntStos is;        // obiekt lokalny typu stos
    for(int i = 0; i < 20; i++) // zapełniamy stos wartościami
       is.push(i);

    IntStosIter it(is); // obiekt lokalny typu iterator

    // przemieszczamy się po stosie za pomocą iteratora:
    for(int j = 0; j < 20; j++)
       cout << it++ << endl;

    …
```


C++ - szablony kontenerów

Korzystanie z kontenera i iteratora:

- Tak jak w przypadku wskaźnika, zadaniem obiektu typu **`IntStosIter`** jest przemieszczanie się po **`IntStos`** i zwracanie przechowywanych wartości.
- W tym prostym przykładzie **`IntStosIter`** może poruszać się tylko w przód (używając zarówno przed jak i przyrostkowej formy operatora ++). Jednak, oprócz narzuconych przez kontener, na którym pracuje, nie ma żadnych ograniczeń na to, jak iterator może być zdefiniowany.
- Możemy poruszać się po elementach stosu nie pobierając ich, tak jakby to była tablica, dzięki temu, że iterator zna strukturę stosu oraz ma dostęp do wszystkich składowych kontenera (jest **`friend`**)

C++ - szablony kontenerów

Korzystanie z kontenera i iteratora:

- Iterator ma za zadanie upraszczać złożony proces bezpiecznego przemieszczania się od jednego elementu kontenera do drugiego.
- Całkowicie akceptowalnym jest, aby iterator przemieszczał się w dowolnym kierunku "swojego" kontenera, a także by powodował zmiany wartości elementów (w ramach ograniczeń jakie narzuca dany kontener).
- Dąży się do tego, aby każdy iterator posiadał taki sam interfejs, niezależnie od tego, jakie elementy w jakim kontenerze wskazuje.

C++ - szablony kontenerów

Korzystanie z kontenera i iteratora:

- W bibliotekach STL każdy kontener zawsze ma związaną ze sobą klasę zwaną iteratorem.
- Najczęściej klasa iteratora jest zadeklarowana jako zagnieżdżona wewnątrz deklaracji kontenera i ma nazwę **`iterator`**.
- Służy do tworzenia obiektów typu **`iterator`** dopasowanych – co do zasady działania – do danej klasy kontenerów.

C++ - szablony

```
class IntStos {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
 public:
  IntStos() : top(0) {}
  void push(int i) {
   assert(top < ssize);  stack[top++] = i;
  }
  int pop() {
   assert(top > 0);  return stack[--top];
  }
```

```
  class iterator {
       IntStos& s;
       int index;
   public:
   iterator(IntStos& is) : s(is), index(0) {}
   int operator++() { // Prefix
    assert(index < s.top);
    return s.stack[++index];
   }
   int operator++(int) { // Postfix
    assert(index < s.top);
    return s.stack[index++];
   }
  };
  iterator begin() { return iterator(*this); } // zwraca iterator
  friend class iterator;
 };
```


C++ - szablony

```
class IntStos {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
 public:
  IntStos() : top(0) {}
  void push(int i) {
   assert(top < ssize);  stack[top++] = i;
  }
  int pop() {
   assert(top > 0);  return stack[--top];
  }
```

```
  class iterator {
       IntStos& s;
       int index;
   public:
   iterator(IntStos& is) : s(is), index(0) {}
   iterator(IntStos& is, bool) : s(is), index(s.top) {}
   int operator++() { // Prefix
    assert(index < s.top);
    return s.stack[++index];
   }
   int operator++(int) { // Postfix
    assert(index < s.top);
    return s.stack[index++];
   }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
  friend class iterator;
 };
```


C++ - szablony

```
class IntStos {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
 public:
  IntStos() : top(0) {}
  void push(int i) {
   assert(top < ssize);  stack[top++] = i;
  }
  int pop() {
   assert(top > 0);  return stack[--top];
  }
  class iterator {
       IntStos& s;
       int index;
  public:
  iterator(IntStos& is) : s(is), index(0) {}
  iterator(IntStos& is, bool) : s(is), index(s.top) {}
```

```
   int operator++() { // Prefix
    assert(index < s.top);
    return s.stack[++index];
   }
   int operator++(int) { // Postfix
    assert(index < s.top);
    return s.stack[index++];
   }
   bool operator==(const iterator& rv) const {
     return index == rv.index;
   }
   bool operator!=(const iterator& rv) const {
     return index != rv.index;
   }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
  friend class iterator;
 };
```


C++ - szablony

Korzystanie z rozszerzonych wersji klas `IntStos` i `iterator`:

```
int main(int argc, char *argv[])
{
    IntStos is;

    // zapełniamy stos wartościami
    for(int i = 0; i < 20; i++)
        is.push(i);

    // przemieszczamy się po stosie za pomocą iteratora:
    IntStos::iterator it = is.begin();
    while(it != is.end()) // nie musimy znać rozmiaru stosu..
        cout << it++;
```


C++ - szablony

Korzystanie z rozszerzonych wersji klas IntStos i iterator:

Przydałby się sposób pozyskiwania elementu wskazywanego przez iterator, bez jednoczesnej jego inkrementacji. W tym celu dodajmy do klasy **`IntStos::iterator`** przeciążony operator *:

```
int operator*() const { return s.stack[index];}
```

C++ - szablony

Korzystanie z rozszerzonych wersji klas IntStos i iterator:

Teraz chcielibyśmy móc napisać w programie tak:

```
IntStos::iterator it;
for (it = is.begin(); it!=is.end(); it++)
      printf("%i\n",*it);
```

To jednak wymaga utworzenia najpierw konstruktora domyślnego w klasie **IntStos::iterator**.

191

C++ - szablony

Dopisujemy konstruktor domyślny:

```
iterator() { };
```

ale..

```
1>------ Rebuild All started: Project: kontenerStosu, Configuration: Debug Win32 ------
1>  main.cpp
1>d:\zpodemo2014\kontenerstosu\intstos.h(25): error C2758:
'IntStos::iterator::s' : must be initialized in constructor base/member initializer list
1>d:\zpodemo2014\kontenerstosu\intstos.h(22) : see declaration of 'IntStos::iterator::s'
========== Rebuild All: 0 succeeded, 1 failed, 0 skipped ==========
```

.. dowiadujemy się, że składowej referencyjnej w obiekcie nie wolno zostawiać bez inicjalizacji.

Rezygnujemy więc z deklarowania iteratorów, które nie są przypisane do żadnego stosu.

192

C++ - szablony

Ostatecznie w zamian dodajemy do klasy **IntStos::iterator** przeciążony operator przypisania:

```
iterator& operator=(const iterator& rv) {
        index = rv.index;
        return *this;
}
```

Dlatego nasz kod korzystający ze stosu wygląda tak:

```
IntStos::iterator it(is);     // tworzymy iterator i ..
          // ..robimy pętlę for po elementach stosu:
for (it = is.begin(); it!=is.end(); it++)
      printf("%i\n",*it);
```


193

C++ - szablony

Suplement – jeżeli nadal..

..jest to dla nas niekomfortowe, że musimy deklarować iteratory dopiero po zadeklarowaniu obiektu reprezentującego stos, to:

aby uniknąć problemów z konstruktorem domyślnym zamiast zmiennej referencyjnej możemy używać w klasie 'iterator' wskaźnika.

```
class iterator {
        IntStos* s;
        int index;
```

Teraz już bez problemu dodajemy konstruktor domyślny oraz odpowiednio dostosowujemy kod metod składowych iteratora.

194

Dlaczego iteratory są takie ważne przy korzystaniu z kontenerów?

195

C++ - szablony

Jeżeli połączymy iteratory z szablonami, dziedziczeniem i polimorfizmem dostajemy zupełnie nowe możliwości programowania.

Przykład:

Przyjmijmy, że mamy kilka różnych typów kontenerów, każdy ze swoim typem iteratora. Na podstawie każdego z nich został zadeklarowany inny obiekt.

Przyjmijmy, że mamy kilka różnych klas dziedziczących po jednej wspólnej klasie bazowej, reprezentujących np. figury na płaszczyźnie. Każdy obiekt przechowuje współrzędne położenia figury na płaszczyźnie oraz inne jej atrybuty. Każdy posiada metodę **rysuj**, która jest metodą polimorficzną

Obiekty różnych typów są przechowywane w kontenerach różnych typów. Liczba typów kontenerów i typów obiektów może się zmieniać.

196

C++ - szablony

Chcielibyśmy dla dowolnego podzbioru tych obiektów wywołać ich metodę polimorficzną **`rysuj`**, jednak kontenery nie mają wspólnego typu bazowego, mają natomiast różne metody dostępu do danych, które przechowują. ☹

Korzystamy więc z ich iteratorów:

```
template<typename Iter>
void wypisz(Iter start, Iter end) {
      while (start!=end) {
            (*start)->rysuj();
            start++;
      }
};
```


C++ - szablony

Szablon ‘wypisz’ można nazywać ***algorytmem***

- ponieważ określa, jak zrobić coś z wykorzystaniem danych iteratorów, opisujących zakres elementów.
- Jest to możliwe przy założeniu, że iteratory te mogą być wyłuskiwane, porównywane i inkrementowane. Takie właściwości mają wszystkie iteratory biblioteki STL.
- Uzyskanie takiej ogólności *bez pośrednictwa iteratorów* nie byłoby możliwe – kontenery mogą mieć różne właściwości i ich bezpośrednie używanie nie pozwala na jednorodne ich stosowanie.
- **Tylko iteratory pozwalają na:**

  napisanie *kodu ogólnego przeznaczenia* bez znajomości budowy i zasady działania kontenerów

C++ - szablony

Lekko zmodyfikowany szablon ‘wypisz’ może się przydać również w przypadku klasy-kontenera: `IntStos`

```
template<typename Iter>
void wypisz(Iter start, Iter end)
   {
    while (start!=end) {
       cout << *start << endl;
           start++;
   }
};
```

```
int main(int argc, char *argv[])
{
 IntStos is;
  for(int i = 0; i < 20; i++)
   is.push(i);

 double tab[20];
 for(int i = 0; i < 20; i++)
  tab[i] = 20-i;

 wypisz(is.begin(), is.end());
 wypisz(tab, tab+20);
```


SZABLONY – ROZSZERZENIA I UZUPEŁNIENIA

C++ - szablony

Szablony – rozszerzenia i uzupełnienia:

- Szablon jako parametr szablonu
- Zagnieżdżanie
- Specjalizacja jawna
- Częściowe uporządkowanie szablonów klas
- Cechy charakterystyczne
- Rekurencyjny wzorzec szablonu

Szablony

szablon jako parametr szablonu

Szablony – rozszerzenie

Są dwie odmiany szablonów:

1. klas,
2. funkcji.

Każdy parametr szablonu może być:

1. typem (wbudowanym lub zdefiniowanym przez użytkownika),
   ```
   template<typename T>
   class ElementListy;
   ```
2. stałą w chwili kompilacji,
   ```
   template<typename T, int rozmiar>
   class SzablonStosu
   ```
3. **innym szablonem..**

Szablony – rozszerzenie

Przykład

Przyjmijmy, że mamy szablon reprezentujący kontener typu *stos*, o liczbie komórek tablicy dostosowującej się adaptacyjnie do potrzeb:

```
template<class T>
class DynamicznyStos {
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  DynamicznyStos();
  ~DynamicznyStos();
  void push_back(const T& t);
  void pop_back();
  T* begin() { return data; };
  T* end() { return data + count; };
};
```


Szablony – rozszerzenie

```
template<class T>
class DynamicznyStos {
  enum { INIT = 10 };
  T* data;
  size_t capacity;
  size_t count;
public:
  DynamicznyStos() {
    count = 0;
    data = new T[capacity = INIT];
  };
  ~DynamicznyStos() {
    delete [] data;
  };
```

```
void push_back(const T& t) {
    if(count == capacity) {
      size_t newCap = 2 * capacity;
      T* newData = new T[newCap];
      for(size_t i = 0; i < count; ++i)
        newData[i] = data[i];
      delete [] data;
      data = newData;
      capacity = newCap;
    };
    data[count++] = t;
  };
  void pop_back() {
    if(count > 0)
      --count;
  };
  T* begin() { return data; };
  T* end() { return data + count; };
};
```


Szablony – rozszerzenie

Przyjmijmy, że potrzebujemy teraz zadeklarować uniwersalny kontener, który będzie miał jednorodny zbiór metod i operatorów udostępniających zasoby, innych niż to, co posiada **`DynamicznyStos`**.

Zdefiniujmy teraz szablon reprezentujący kontener, którego parametrami są typ przechowywanych danych oraz **inny, dowolny kontener** (!)

Szablony – rozszerzenie

```
template<typename T,
   template<typename> class Seq>
class Kontener {
  Seq<T> seq;
public:
  …
};
```


Szablony – rozszerzenie

- **`DynamicznyStos`** jest zwykłym szablonem klasy
- **`Kontener`** jest szablonem klasy, który przechowuje dane w strukturze budowanej na podstawie innego szablonu kontenera
- Kluczowa linijka kodu w **`Kontener`** to:
  ```
  Seq<T> seq;
  ```
- W nagłówku **`Kontener`** nie jest konieczne podawanie symbolu dla typu używanego przez Seq:
  ```
  template<typename T, template<typename> class Seq>
  ```
- Tak też można (*wersja dla purystów*):
  ```
  template<typename T, template<typename U> class Seq>
  ```
  ale `U` jest to informacja nieistotna – ważniejsze jest to, że `Seq` jest jednoparametrowy.

Szablony – rozszerzenie

```
template<typename T,
   template<typename> class Seq>
class Kontener {
  Seq<T> seq;
public:
  void append(const T& t){
    seq.push_back(t);
  };
  T* begin() {
    return seq.begin();
  };
  T* end() {
    return seq.end();
  };
};
```

```
int main() {
  Kontener<int, DynamicznyStos> C;
  C.append(0);
  C.append(1);
  C.append(2);
  int* p = C.begin();
  while(p != C.end())
    cout << *p++ << endl;

  system("PAUSE");
  return EXIT_SUCCESS;
} ///:~
```


Szablony – rozszerzenie

Inny przykład: dany jest pewien szablon ze stałą..

```
template<typename T, size_t N> class Stosik {
  T data[N];
  size_t count;
public:
  Stosik() { count = 0; };
  void push_back(const T& t) {
    if(count < N)
      data[count++] = t;
  };
  void pop_back() {
    if(count > 0)
      --count;
  };
  T* begin() { return data; };
  T* end() { return data + count; };
};
```


Szablony – rozszerzenie

Inny przykład - zastosowanie

```
template<typename T,
   size_t N,
   template<typename ,size_t>
   class Seq >
class Kontenerek {
  Seq<T,N> seq;
public:
  void append(const T& t) {
    seq.push_back(t);
  };
  T* begin() {
   return seq.begin(); };
  T* end() {
   return seq.end(); };
};
```

```
int main() {
  const size_t N = 10;
  Kontenerek<int, N, Stosik> C;
  C.append(0);
  C.append(1);
  C.append(2);
  int* p = C.begin();
  while(p != C.end())
    cout << *p++ << endl;
} ///:~
```