ŠABLÓNY (TEMPLATES)

Generické programovanie

- Implementácia kontajnerov, t.j. dátových typov implementujúcich množinu prvkov rovnakého typu bez potreby reimplementácie pre každý možný typ týchto prvkov.
- Špecifikovať algoritmus bez väzby na konkrétny dátový typ
- Napríklad algoritmus pre triedenie poľa funguje rovnako pre triedenie poľa celých čísel aj reálnych čísel. Ale v klasických prísne typovaných programovacích jazykoch ho musím písať dva krát
- Zrušiť prísnu typovú kontrolu by ale bolo kontraproduktívne. Vďaka nej sa podarí vychytať množstvo chýb softvéru už počas kompilácie

Príklad: jednoduché triedenie poľa

```
void sort(int *a, int n) {
  int i,j;
  for(i=2; i<n; i++) {
    for(j=i; j>0; j--) {
      if (a[j-1] > a[j]) {
         int t;
         t=a[j]; a[j]=a[j-1]; a[j-1]=t;
      }
    }
  }
}

int main() {
  int i, a[] = {6,2,9,5,1};
  sort(a, 5);
  for(i=0; i<5; i++) printf("%d ", a[i]);
}
```

C preprocessor umožňuje "generické" konštrukcie

```
#define SORT(type, a, n) {                          \
   int i,j;                                         \
   for(i=2; i<n; i++) {                             \
     for(j=i; j>0; j--) {                           \
       if (a[j-1] > a[j]) {                         \
         type t;                                    \
         t=a[j]; a[j]=a[j-1]; a[j-1]=t;             \
       }                                            \
     }                                              \
   }                                                \
 }

int main() {
  int i, a[] = {6,2,9,5,1};
  SORT(int, a, 5);
  for(i=0; i<5; i++) printf("%d ", a[i]);
}
```

C preprocessor a jeho "generické" programovanie

- Dokonca sme urobili akúsi generickú knižnicu pre C (sglib).
- Ale, každý cíti, že to nie je "dobré" riešenie.
- C preprocessor nie je vôbec bezpečný.
- Z matematického hľadiska sa sémantika programov s preprocessorom prakticky nedá definovať.
- To znamená, že takéto konštrukcie by nemali byť súčasťou "normálneho" programovania

Riešenie OOP

- Algoritmus definovaný nad nejakou triedou funguje aj pre všetky odvodené triedy
- Ak si zvolím nejakú univerzálnu triedu (nazvem ju napríklad Object) z ktorej odvodím všetky ostatné, tak algoritmy pracujúce nad touto triedou budú generické.
- Ak algoritmy budú potrebovať dodačné operácie (napríklad usporiadanie), tak ich zabezpečím dodatočným dedením, buď cez viacnásobné dedenie (C++), alebo cez niečo podobné (Java interface, ...)

Nevýhoda riešenia OOP

- Návratové hodnoty generických funkcií budú mať generický typ (napr. Object), pri ich používaní musím neustále používať cast-y.
- Generické algoritmy nebudú pracovať nad built-in dátovými typmi (ktoré samozrejme nie sú odvodené od Object) napríklad, int, double, ...
- Volanie virtuálnej metódy je pomalšie ako priame volanie. Volanie virtuálnej metódy cez viacnásobné dedenie (alebo cez Java interface) je ešte pomalšie.

Generické typy v Java 1.5

- Zjednodušene povedané, Java 1.5 zavádza generické typy ako nadstavbu nad OOP generickým programovaním
- Pridáva syntaktický cukor, ktorý odstraňuje potrebu cast-ov.
- Ale ďalšie dve spomínané nevýhody zostali

Riešenie C++

- Inšpirované z použitia preprocessora na generické programovanie.
- C++ sa snažilo vyšpecifikovať a vložiť do jazyka konštrukciu umožňujúcu robiť, to čo v preprocessore, ale v rámci jazyka a s "dobre" definovanou sémantikou.
- Táto konštrukcia sa volá *template*, alebo náš equivalent *šablóna*.
- Táto konštrukcia netrpí problémami OOP generického programovania. Trpí ale množstvom vlastných problémov. Napriek svojej nespornej užitočnosti sú template-y považované za nie úplne dobre dotiahnutú časť C++.

Informatívna syntax definície šablóny

šablóna ::= **template** **<** *formálne_parametre* **>** *deklarácia_šablóny*

deklarácia_šablóny ::= *deklarácia_triedy*
| *deklarácia funkcie*

formálne_parametre ::= *formálny_parameter*
| *formálny_parameter* **,** *formálne_parametre*

formálny_parameter ::= **class** *meno_formálneho_parametra* *možno_default_argument*
| **typename** *meno_formálneho parametra* *možno_default_argu*
| *deklarácia_formálneho_parametra_ako_u_funkcie*
...

možno_default_argument ::= *typ*
|

Template, príklad1

```
template <typename X> class list {
private:
  X         data;
  list<X>   *next;
public:
  ...
  void append(list<X>);
  ...
};

template <typename X> void list<X>::append(list<X>) { .... }


int main() {
  list<int> a, b;
  ...
  a.append(b);
}
```

Template definuje generickú triedu **list** s parametrom X

Post-definícia metódy **append** z generickej triedy

Template, príklad2

Template definuje generickú funkciu **max**, pre akýkoľvek typ **T**

```
template <typename T> T max(T x, T y) {
   if (x > y) return(x); else return(y);
}
```

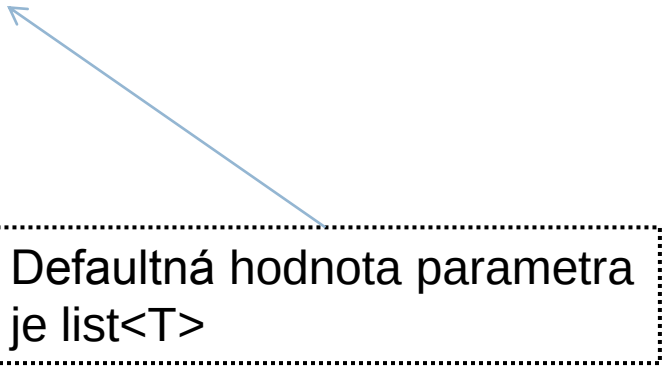
Template, príklad3

```
template <typename X, int N> class vector {
private:
  X[N]      data;
public:
  ...
  X &operator[](int i) {...}
  ...
};


int main() {
  vector<double, 1000> a;
  ...
  a[5] = 1.0;
  ...
}
```

Template definuje generickú triedu **vector** s parametrami X (typ prvov) a N (maximálna veľkosť vektora)

Template, príklad 4

```
template <typename T, typename CONT = list<T> >
class Stack {
  private:
    CONT elems;             // elements
  public:
    void push(T &x);
    void pop();
    T top();
    ...
};

int main() {
  Stack<int, vector<int, 100> >  s1;
  Stack<int>                     s2;
  ...
}
```



Vnorené šablóny, syntaktický problém1

```
int main() {
  list<int>         a;
  list<list<int>>   a;
}
```

Chyba! >> je v C++ lexikálny symbol pre operátor bitového posunu vpravo!

```
int main() {
  list<int>          a;
  list<list<int> >   a;
}
```

Správne!

Vnorené šablóny, syntaktický problém2

```
int main() {
  ...
  vector< int, 5 >                 a;
  vector< int, x>10?1000:100 >  b;
}
```

Chyba! > sa pochopí ako koniec templatu.
Nie ako logický operátor väčší.

```
int main() {
  ...
  vector< int, 5 >                   a;
  vector< int, (x>10)?1000:100 >  b;
}
```

Správne!

Template template parameter

```
template <typename T, template <typename ELEM> class CONT>
class Stack {
  private:
    CONT<T> elems;          // elements
  public:
    void push(T &x);
    void pop();
    T top();
};

int main() {
  Stack<int, std:vector> s;
  ...
}
```

Stack je template, ktorého druhý paremeter je meno nejakého iného template-u

Friend template

- V triede môžete špecifikovať, že nejaký template bude mať prístup k private členom.

```
class X {
   template <typename T> friend class Vector;
   ...
};
```

Miesto inštanciácie šablón

- Šablóna je čosi ako makro. V momente definície jej telo de-facto neexistuje. Až na mieste prvého použitia s nejakým konkrétnym parametrom sa vygeneruje skutočná definícia.
- Ak chcem aby na nejakom mieste v programe bola šablóna vygenerovaná, môžem použiť explicitnú inštanciáciu. Napr.

```
template class vector<int, 1000>;
```

Linkovanie metód definovaných v šablónach

- Štandardne, metódy zo šablón nemajú globálne meno a nie sú linkované medzi súbormi. Sú to akoby "inline" metódy.
- Nepríjemný dôsledok je, že takéto metódy budú v kóde skompilované toľko krát, koľko súborov ich používa
- Na zabránenie tohto efektu bolo do C++ pridané slovíčko "export", ktoré spôsobuje globálnu viditeľnosť metód.

```
export template <typename T> class MyClass {...}
```

Explicitná špecializácia

```
template <typename T> class A {
public:
  void fun() {printf("A fun ");}
};

template <> class A<int> {
public:
  void fun() {printf("A<int> fun ");}
};

int main() {
  A<int> a;
  a.fun();
}
```

V prípade, že niekto inštanciuje A s parametrom int, použije sa táto definícia triedy A namiesto všeobecnej z template-u

Čiastočná špecializácia

```
class <typename T1, typename T2, typename T3> class A {
   ...
};

class <typename T1, typename T3> class A<T1, T1 *, T3> {
   ...
};
```

V prípade, že niekto použije triedu A s 3 parametrami, pričom druhý parameter je smerník na prvý, tak sa použije táto definícia namiesto všeobecnej. Aj táto definícia je ale stále template, keďže závisí na dvoch generických parametroch.

Funkcionálny template, max

```
template <typename T> T max(T x, T y)   {
    return (x > y) ? x : y;
}

int main() {
  printf("max1 == %d\n", max(2, 3));
  printf("max2 == %f\n", max(1.5, 2.5));
  printf("max3 == %x\n", max(2, 2.5));
  max("jano", "juraj");
}
```

Definuje funkciu max ktorá sa dá aplikovať na akýkoľvek typ

Inštanciácia a vyvolanie pre funkciu max(int, int)

Inštanciácia a vyvolanie pre funkciu max(double, double)

Chyba! Pri inštanciácii funkcionálnych templatov sa nerobí žiadna konverzia typov. Keďže každý z parametrov max je iného typu, template sa neinštanciuje.

Chyba! aj tu majú parametre rôzny typ: char[5] a char[6].

Ako opraviť chybu z predchádzajúceho príkladu

- Cast:
  ```
  max( static_cast<double>(2), 2.5)
  ```
- Explicitná ištanciácia:
  ```
  max<double>(2, 2.5)
  ```
- Template s dvomi rôznymi typmi:

```
template <typename T1, typename T2> T1 max(T1 x, T2 y)  {
    return (x > y) ? x : y;
}
```

Ale tento template vracia hodnotu typu T1, ale čo ak výsledkom bude y, ktoré je typu T2?

Ako opraviť problém z predchádzajúceho príkladu

- Pridať ešte jeden parameter

```
template<typename T1,typename T2,typename T3> T3 max(T1 x,T2 y)   {
    return (x > y) ? x : y;
}
```

- Ale T3 nevystupuje v parametroch, nedá sa automaticky inferovať. T.j. pri použití treba použiť explicitnú ištanciáciu

```
int main() {
  max<int, double, double>(2, 2.5);
}
```

Ako opraviť problém z predchádzajúceho príkladu

- Typ výsledku bude prvým parametrom

```
template<typename T3,typename T1,typename T2> T3 max(T1 x,T2 y)   {
    return (x > y) ? x : y;
}


 int main() {
   max<double>(2, 2.5);
 }
```

Preťaženie (overloading) funkcionálnych templatov

- Viacero funkčných templatov môže mať to isté meno a to aj rovnaké ako normálna funkcia.

```
int max(int x, int y) {...}

template <typename T> T max(T x, T y) {...}

template <typename T> T max(T x, T y, T z) {...}

int main() {
  max(2, 3);
  max(1.1, 2.2);
  max(1,2,3);
  max(2, 2.5);
}
```

Preťaženie (overloading) funkcionálnych templatov

- Nájdenie volanej funkcie:

1. Hľadaj presnú korešpondenciu (exact match) parametrov a normálnej (nie template) funkcie.
2. Hľadaj template z ktorého po inštanciácii vznikne funkcia s presnou korešpondenciou typov (exact match).
3. Rob normálne rozhodnutie pre preťažené normálne (nie template) funkcie.

Preťaženie (overloading) funkcionálnych templatov

```
int max(int x, int y) {...}

template <typename T> T max(T x, T y) {...}

template <typename T> T max(T x, T y, T z) {...}

int main() {
  max(2, 3);
  max(1.1, 2.2);
  max(1, 2, 3);
  max(2, 2.5);
}
```


template max(int, int)


template max<double>(T,T)


template max<double>(T,T,T)


max(int, int)

Pokročilá inferencia typu

```
template <typename T, int I> class vector { };

template <typename T, int I> void tfun(vector<T, I> v) {
  printf("I == %d", I);
}

int main() {
  vector<int, 5> a;
  // ...
  tfun(a);
}
```

Vypíše: I == 5

Inferencia (matching) parametrov šablón

- T
- const T
- volatile T
- T *
- T &
- T[n]
- *a_type*[I]
- CT<T>
- CT<I>
- T (*)(*args*)
- *a_type*(*)(*args_with_T*)
- *a_type*(*)(*args_with_I*)
- T C::*
- C T::*

Pokročilá inferencia typu príklad

```
template <typename T> max(T x, T y) {...}

template <typename T> max(T *x, T *y) {...}

int main() {
   int a,b;
   max(&a, &b);
}
```

Vyvolá max(T*, T*), nie všeobecnejší max(T,T)

Standard Template Library

- Containers
  1. Sequence Containers
     a) vector
     b) deque
     c) list
  2. Container Adaptors
     a) stack
     b) queue
     c) priority_queue
  3. Associative Containers
     a) set
     b) multiset
     c) map
     d) multimap

Standard Template Library

- Iterators
  I. Random Access Iterators
  II. Bidirectional Iterators
  III. Forward Iterators

  IV. Input Iterators
  V. Output Iterators

  VI. Stream Iterators

  VII. Iterator Adaptors

Standard Template Library

- Allocators
- Generic Algorithms

STL Generic algorithms

I. Comparators: comparing (ranges of) elements: equal; includes; lexicographical_compare; max; min; mismatch;
II. Copiers: performing copy operations: copy; copy_backward; partial_sort_copy; remove_copy; remove_copy_if; replace_copy; replace_copy_if; reverse_copy; rotate_copy; unique_copy;
III. Counters: performing count operations: count; count_if;
IV. Heap operators: make_heap; pop_heap; push_heap; sort_heap;

STL Generic algorithms

I. Initializers: initializing data: fill; fill_n; generate; generate_n;
II. Operators: performing arithmetic operations of some sort: accumulate; adjacent_difference; inner_product; partial_sum;
III. Searchers: performing search (and find) operations: adjacent_find; binary_search; equal_range; find; find_end; find_first_of; find_if; lower_bound; max_element; min_element; search; search_n; set_difference; set_intersection; set_symmetric_difference; set_union; upper_bound;

STL Generic algorithms

I. Shufflers: performing reordering operations (sorting, merging, permuting, shuffling, swapping): inplace_merge; iter_swap; merge; next_permutation; nth_element; partial_sort; partial_sort_copy; partition; prev_permutation; random_shuffle; remove; remove_copy; remove_copy_if; remove_if; reverse; reverse_copy; rotate; rotate_copy; sort; stable_partition; stable_sort; swap; unique;Non-mutating

II. Visitors: visiting elements in a range: for_each; replace; replace_copy; replace_copy_if; replace_if; transform; unique_copy;