TRIEDNE ŠABLÓNY **2**

VNORENÉ ŠABLÓNOVÉ TRIEDY **5**

TRIEDNE ŠABLÓNY S VIACERÝMI GENERICKÝMI ARGUMENTMI **6**

TRIEDNE ŠABLÓNY AKO FRIEND **8**

FUNKČNÉ ŠABLÓNY **8**

Triedne šablóny

C++ trieda zvyčajne spracováva nejaký druh dát. Často funkčnosť triedy dáva koncepčne zmysel i pre ďalšie dátové typy. Zoberme napríklad triedu, ktorá vytvára nejaký rámec okolo nejakého dátového typu, podporujúci rozmanité operácie s typom, potom dáva zmysel izolovať dátový typ celkom od triedy, umožňujúci narábať s generickým dátovým typom T. Takáto trieda nie je v skutočnosti triedou, ale popisom triedy a nazýva sa ***triedna šablóna***. Triednu šablónu používa kompilátor na vytvorenie skutočnej triedy počas kompilácie, použijúc konkrétny dátový typ.

Zoberme napríklad tvorbu triedy na prácu so zreťazeným zoznamom. Operácie so zreťazenými zoznamami nie sú závislé od typu dát spracovávaného zreťazeným zoznamom. Jediná triedna šablóna sa dá použiť pre všeobecné zreťazené zoznamy, celé čísla, štruktúry alebo akýkoľvek iný definovaný typ dát. Ak chceme deklarovať triednu šablónu pre triedu zreťazeného zoznamu, mali by sme použiť notáciu z Výpis 1.

Výpis 1 Deklarácia triednej šablóny

```
template<class T> class List {
  public:
    List();
    void add(T&);
    void remove(T&);
    void detach(T& t) {remove(t);};
    ~List();
};

template<class T> List<T>::List()
{
//.....
}

template<class T> void List<T>::add(T&)
{
//...
}

template<class T> void List<T>::remove(T&)
{
//...
}


template<class T> List<T>::~List()
{
//....
}
```

Deklarácia triedy začína kľúčovým slovom template a potom výraz

```
<class T>
```

ktorý deklaruje generický dátový typ T, ktorý použijeme vo vnútri triedy. Ak deklarujeme triednu šablónu a potom zlyhá použitie ľubovoľného generického dátového typu T, C++ sa nebude sťažovať. Vhodná by bola varovná správa

```
Parameter 'xyz' is never used
```

ktorá nastane vo funkcii, ak ju deklarujeme ale nepoužijeme funkčný argument. Okrem malej zmeny syntaxe deklarácie, použitie triednej šablóny sa moc nelíši od iných tried. Keď vytvoríme inštanciu triednej šablóny, prenesením špecifického dátového typu vytvoríme ***šablónovú triedu***. Nepleťme si triednu šablónu so šablónovou triedou. Pojmy sú trochu mätúce, najmä pre ľudí, ktorých rodným jazykom je angličtina, ale je to tak. Ak použijeme triednu šablónu , kompilátor automaticky vytvorí všetky členské funkcie, ktoré budú narábať s dátovým typom, ktorý zadáme. Zoberme napríklad kód z Výpis 2,ktorý používa triednu šablónu List.

Výpis 2 Používanie šablónových tried

```
void main()
{
  List<long> phone_numbers;
  List<char *> club_members;

  static long number = 5551000;
  phone_numbers.add(number);

  static char* name = "Michael";
  club_members.add(name);
}
```

Kompilátor generuje kód, ktorý narába s dátovými typmi int a char* a generuje celú triedu pre každý dátový typ. Ak triedna šablóna deklaruje statické dátové členy alebo virtuálne funkcie, vygenerované šablónové triedy budú mať svoje vlastné kópie statických dátových členov a ich vlastné virtuálne funkcie.

Členské funkcie triednej šablóny vyžadujú špeciálnu notáciu na deklarovanie svojho návratového typu. Ak funkcia vracia int, deklarujeme to takto:

```
template<class T> int MyClass<T>::remove(T&)
{
//......
}
```

kde návratový typ sa deklaruje za výrazom template<class T>

Ak chceme deklarovať viac ako jeden argument, použijeme syntax z Výpis 3

Výpis 3 Deklarovanie triednej šablóny s viacerými argumentmi

```
#include <stlib.h>

template<class T,int i,long L>
class MyClass {
  T* object;
  lonh offset;
  int size,length;

public:
  MyClass();
  MyClass(int);
  int Foo(long);
};

template<class T, int i,long L>
```

```
MyClass<T,i,L>::MyClass()
{
//  pouzi premennu deklarovanu s triednou sablonou
offset = L;

// pouzi premennu deklarovanu s clenskou  funkciou
size = i;

// pouzi premennu deklarovanu s triednou sablonou
// a s clenskou funkciou
object = new T();
}

template<class T,int i,long L>
MyClass<T,i,L>::MyClass(int x)
{
  //prirad argumenty triednej sablony
// do privatnej premennej
length = i;
offset = L;

//pouzi funkcny argument
size = x;

//pouzi dalsi argument triednej sablony
object = new T();
}

template<class T,int a, long L>
int MyClass<T,a,L>::Foo(long value)
{
  return(value&0xFF);
}
```

Triedna šablóna MyClass je deklarovaná tak, aby akceptovala argumenty typu class T, int a char 8. Toto sú argumenty triedy ako celku, nie argumenty, nie argumenty hociktorej z členských funkcií. Tieto triedne argumenty môžeme použiť len vo vnútri členských funkcií triednej šablóny. Ak chceme použiť triednu šablónu MyClass, mali by sme napísať takýto kód (Výpis 4):

Výpis 4 Použitie šablónových tried s viacerými argumentmi

```
// vclen hlavickovy subor triednej sablony MyClass

class List{};

void main()
{
  MyClass<int,5,100> x(2);
  MyClass<float,100,200> y(3);
  MyClass<List,100,300> z();
}
```

Tiež môžeme deklarovať implicitné argumenty triednu šablónu.

Vnorené šablónové triedy

Dajú sa použiť šablónové triedy v deklarácií inej šablónovej triedy? Áno a niekedy sú takéto triedne šablóny perfektným riešením, obzvlášť keď narábame s tzv. kontajnerovými triedami. Kontajnerová trieda je trieda, ktorej úlohou je správa skupiny ďalších tried. Zoberme napríklad implementáciu všeobecného zreťazeného zoznamu. Najskôr musíme triednu šablónu definovať tak, aby spracovávala každý uzol v zozname. Tu je možný prístup:

```
// vytvor triednu sablonu pre uzly v zretazenom zozname

template<class R> class Node {
Node<R>* previous;
Node<R>* next;
R* data;

public:
  Node(Node<R>,Node<R>*,R*);
~Node();
};

template<class R>
Node<R>::Node(Node<R>* p,Node<R>* n,R* d)
{
  previous =p;
  next = n;
  data =d;
}

template<class R>
Node<R>::~Node()
{
  delete data;
}
```

Teraz môžeme generickú triedu Node<R> použiť kontajnerovou triedou na implementáciu vlastného zreťazeného zoznamu použitím nasledovného kódu:

```
// pouzi predchadzajucu triednu sablonu v dalsej
// triednej sablone

template<class R> class List {

  Node<R> *head;
  Node<R>* current;

public:
  List();
  void add(R*);
  void remove();
  ~List();
};

template<class R>
List<R>::List()
{
  head=current=0;
}

template<class R>
void List<R>::add(R* object)
```

```
{
  if(!head) {
    // vytvor prvy uzol zoznamu
    head=current=new Node<R>(0,0,object);
  } else {
    // pripoj novy uzol do zoznamu
    //....
  }
}

template<class R>
void List<R>::remove()
{
  // odpoj aktualny uzol zo zoznamu
//...

// vymaz uzol
delete current;
}

template<class R>
void List<R>::~List()
{
  //vymaz vsetky prvky zoznamu
current=head;
while(current)
  remove();
}
```

Keď vytvoríme triedu List<R>, kompilátor automaticky vytvorí triedu Node<R>. Generická trieda List<R> spravuje obidva objekty List<R> a <R>, i keď deklarácia triedy uvádza iba argument typu <class R>.

Triedne šablóny s viacerými generickými argumentmi

Až doteraz uvedené triedne šablóny mali iba jeden generický argument. Je možné a občas dokonale odôvodnené použiť viac generických argumentov v triednej šablóne. Zoberme napríklad modifikáciu predchádzajúcich tried Node<R> a List<R>, ktoré budú podporovať tri rôzne objekty v každom uzle. Ako treba kód modifikovať ukazuje Výpis 5:

Výpis 5 Použitie viacerých generických argumentov v triednej šablóne

```
   // vytvor triednu sablonu na uchovanie troch roznych datovych typov
v kazdom uzle

   template<class R, class S, class T> class Node {
   Node<R,S,T>* previous;
   Node<R,S,T>* next;
   R* r_data;
   S* s_data;
   T* t_data;

   public:
     Node(Node<R,S,T>*,
                  Node<R,S,T>*,
     R*,S*,T*);
   ~Node();
   };
```

```
template<class R,class  S,class T>
Node<R,S,T>::Node(Node<R,S,T>* p,Node<R,S,T>* n,R* r,S* s,T*t)
{
  previous =p;
  next = n;
  r_data =r;
s_data =s;
  t_data =t;
}

template<class R,class S,class T>
Node<R,S,T>::~Node()
{
  delere r_data;
  delere s_data;
  delere t_data;
}

// pouzi predchadzajucu triednu sablonu v dalsej
// triednej sablone

template<class R,class S,class T> class List {

  Node<R,S,T> *head;
  Node<R,S,T>* current;

public:
  List();
  void add(R*);
  void remove();
  ~List();
};

template<class R,class S,class T>
List<R,S,T>::List()
{
  head=current=0;
}

template<class R,class S,class T >
void List<R,S,T>::add(R* object)
{
  if(!head) {
    // vytvor prvy uzol zoznamu
    head=current=new Node<R,S,T>(0,0,object);
  } else {
    // pripoj novy uzol do zoznamu
    //....
  }
}

template<class R,class S,class T >
void List<R,S,T>::remove()
{
  // odpoj aktualny uzol zo zoznamu
//...

// vymaz uzol
delete current;
}
```

```
template<class R,class S,class T >
void List<R,S,T>::~List()
{
  //vymaz vsetky prvky zoznamu
current=head;
while(current)
  remove();
}
```

Notácia s viacerými generickými argumentmi je v podstate to isté ako pri jednom generickom argumente. Deklarovať môžeme toľko argumentov, koľko chceme. Môžeme deklarovať obidva generické i typové argumenty v tej istej triede takto:

```
template<class FIRST,int i,class SECOND,char *name>
class A {

public:
  A();
  //....
};
```

Triedne šablóny ako friend

Triednu šablónu môžeme deklarovať i ako friend inej triedy. Dôsledkom je, že každá ďalšia vygenerovaná funkcia z tejto triednej šablóny bude tiež friend tejto triedy. Zoberme napríklad kód z Výpis 6

Výpis 6 Triedna šablóna ako friend

```
// deklaracia celej triednej sablony ako friend
template<class A>
class Small {
  friend class Large<B>;

  public:
    Small();
    //....
};

template<class A>
class Large {
  public:
    Large();
    //..
};
```

Odteraz hociktorá šablónová trieda typu Large<A> bude friend ku Small<A>. Pojem je celkom silný, pretože určuje friend vzťah nie pre jednu triedu ale pre celú triedu tried.

Funkčné šablóny

Šablóny sa dajú použiť nie len pre triedy ale i pre funkcie. Občas narazíme na situáciu, v ktorej by bola užitočná tá istá funkcia, ale s rôznymi typmi argumentov. C programátori typicky tento problém riešili pomocou makier, ale riešenie nie je ideálne. Zoberme napríklad určenie väčšej z dvoch entít. nasledujúce makro rieši všetky dátové typy:

```
#define max(a,b)  { (x>y)?x:y)
```

takéto makrá používajú C programátori roky, ale má svoje problémy. Makro funguje, ale bráni kompilátoru kontrolovať typy pri porovnávaní. Makro môžeme zavolať s celým číslom a smerníkom na char * a negeneruje sa žiadne varovanie.

Na porovnanie dvoch hodnôt môžeme napísať funkciu:

```
int max(int a,int b)
{
  return a>b?a:b;
}
```

ale takáto funkcia bude spracovávať iba typ int alebo dáta, ktoré sa dajú konvertovať na celočíselný typ. Funkcia nedokáže spracovať iné dátové typy, napríklad float alebo char *. Ak nás zaujímajú iba skalárne čísla, mohli by sme funkciu napísať takto:

```
double max(double a,double b)
{
  return a>b?a:b;
}
```

Táto funkcia by spracovávala všetky dátové typy od char po double, pričom kompilátor prevádza typy argumentov. Táto funkcia má prinajmenšom dve významné obmedzenia:

1. Návratová hodnota je vždy typu double, bez ohľadu na typy argumentov prenášaných do max(...). Použitie takéhoto max(...) vo výraze môže spôsobiť problémy. Zoberme napríklad kód:

```
void main()
{
  cout<< "Max('3','5') is "
          << max('3','5')
          << endl;
}
```

Kód zobrazí s právu:

```
Max('3','5') is 53
```

Problém spočíva v tom, že vkladací operátor prúdu predpokladá, že pracuje s double hodnotou a vytlačí 53 namiesto znaku 5. Explicitné pretypovanie by tento problém vyriešilo, ale pretypovanie možno sotva považovať za všeobecné riešenie.

1. Spracovávať možno iba dáta, ktoré sa dajú konvertovať na double. Fukcia max(double, double) nemôže spracovať smerníky, štruktúry alebo triedne objekty.

C++ používa nový druh deklarácie na implementovanie funkcie ako je max(...). Nová deklarácia sa nazýva funkčná šablóna. Použitie funkčnej šablóny by sme mohli definovať zovšeobecnený tvar funkcie max(...):

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

Deklarácia začína slovom template, ktoré je novým kľúčovým slovom C++. Potom nasleduje výraz <class T>, ktorý indikuje, že nejaká premenná všeobecného typu, nazvaný T (môže sa použiť ľubovoľný identifikátor)bude vo funkcii použitý. Funkcia max(...) je deklarovaná tak, aby akceptovala dva všeobecné T objekty a vracala T objekt. Šablónová funkcia nemusí spracovávať iba triedne premenné: môžeme použiť akýkoľvek dátový typ, ako sú štruktúry, smerníky, celé čísla atď.

Ak použijeme funkčnú šablónu so špecifickým dátovým typom, vytvoríme šablónovú funkciu. Podobne ako u triednych šablón, pojem je mätúci, a tak dôsledne rozlišujeme funkčnú šablónu od šablónovej funkcie. Šablónovú funkciu využíva kód vo Výpis 7:

Výpis 7 Jednoduchý program so šablónovou funkciou

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

void main()
{
  cout<< "Max(3,5) is "
          << max(3,5)
          << endl;

  cout<< "Max('3','5') is "
          << max('3','5')
          << endl;
}
```

Keď kompilátor narazí na šablónovú funkciu, generuje funkciu na spracovanie dátových typov, použitých v príkaze volania. Nie sú potrebné žiadne implicitné konverzie. Ak je prvý argument celé číslo, druhý argument musí byť tiež celé číslo. Ak prvý argument je double, druhý musí byť tiež double.

Použitie šablónovej funkcie nás oslobodzuje od písania rôznych funkcii pre každé možné argumenty, ktoré môže program obsahovať a zároveň umožňuje kompilátoru sledovať typy argumentov. Šablónovú funkciu možno prekrývať rovnako ako každú inú funkciu s tou výhodou, že na prekrývanú funkciu možno aplikovať implicitné konverzie (pozri Výpis 8):

Výpis 8 Prekrytie šablónovej funkcie

```
#include <stdlib.h>
int max(char *a,char *b)
{
  int x,y;
  x=atoi(a);
  y=atoi(b);
  return x>y?x:y;
}
```

Keď sa volá funkcia max(char *, char *), kompilátor sa nepokúsi generovať šablónovú funkciu, ale použije explicitnú funkciu max(char *a,char *b). Hoci šablónové funkcie sú veľmi užitočné, šablóny sú užitočné najmä s triednymi objektmi.