Programare Orientată Obiect

Laborator 10

Despre ce vorbim azi

- Funcții și clase template
- Domenii de nume (namespaces)

Funcții template

- Numite și funcții șablon
- Constituie o nouă modalitate de a refolosi cod sursă (după compunere și derivare)
- Constau în precizarea unui tip de dată generic pentru parametru/parametri

Exemplu

int add(int x, int y) { return x+y; }

- Observăm că funcția funcționează doar pentru întregi
- În cazul în care dăm parametri float sau double se va face conversie cu pierdere de informație (degradantă)
- Putem rezolva problema prin definirea unei funcții template

template <class TIP>
TIP add(TIP x, TIP y) { return x+y; }

Precizări

- Funcția va funcționa atât pentru tipurile de bază (int, float, decimal) cât și pentru tipurile definite de utilizator
- Atenție însă! Funcția va funcționa doar pentru tipurile de bază ce au dat un înțeles operatorului plus (de exemplu folosirea funcției pe tipul **char*** nu va avea sens, ea făcând adunare de pointeri)
- De asemenea pentru ca funcția să funcționeze pentru tipurile definite de utilizator, clasa respectivă trebuie să aibă implementată supraîncărcare pentru operator+

Clase template (șablon)

- Asemeni funcțiilor putem defini clase ce folosesc tipuri de date generice
- Prototipul unei astfel de clase

template <class T1, class T2, … , tip1 c1, tip2 c2, …>

class nume_clasa { … };

Unde *Ti* este un tip generic de dată, *tipi* un tip concret de dată și *ci* este o constantă de acest tip.

Precizări

- Funcțiile inline nu necesită o sintaxă specială
- Funcțiile scrise în afara clasei trebuie să rescrie definiția șablonului
- Să ne uităm pe exemplele 1 și 2

Instanțierea claselor template (1)

- Pentru a face diferența între clase trebuie să specificăm atunci când folosim o clasă template tipul de dată concret pe care dorim să îl folosim
- Așa cum am văzut în exemplele anterioare avem Vector<int>, Vector<float> și vector<Student>
- De asemenea putem folosi instanțieri în funcții ce nu folosesc tipul template, ci un tip concret
- Exemplu: *int masoara(Vector<int> &v1, Vector<int> &v2)*

Instanțierea claselor template (2)

- Putem rescrie funcția de măsurare și ca:

```
template <class TT>
int masoara(Vector<TT>&, Vector<int>&)
```

- Sau ca:

```
template <class TT1, class TT2>
int masoara(Vector<TT1>&, Vector<TT2>&)
```

Constante în clase template

- Așa cum spuneam la început putem introduce inclusiv tipuri de bază și constante de acel tip în definiția claselor template
- Să ne uităm pe exemplul 3

Instanțierea constantelor în clase șablon

- Putem preciza și valori predefinite pentru constantele din definiția șablonului
- *Ex: template <class T=int, int n=10>*
- *Folosire: Vector<> v1; Vector<double> v2; Vector<char, 4> v3;*
- Așadar *v1* este un vector de 10 întregi, *v2* este un vector de 10 elemente de tip double și *v3* este un vector de 4 elemente de tip char

Specializări în clase template

- Folosite pentru tipuri concrete de date ce vor fi folosite în locul șablonului
- Pentru clasa din Exemplul 1 putem observa că șablonul nu funcționează pentru tipul char*
- Specializările sunt prioritare față de definiția standard
- Dezavantajul este că vor trebui specializate toate funcțiile membre

Derivarea claselor template

- Clasă template derivată din clasă template

template <class T>

class Vector_sort : public Vector<T>

- Clasă template derivată din clasa non template
- Clasă non template derivată din clasa template

Class Vector_Sort_Int : public Vector<int>

Compunerea claselor template

- Pot folosi tipurile generice ale clasei “mamă”
- Pot defini alte tipuri generice

Domenii de nume (namespaces)

- Permit organizarea codului sursă
- Astfel putem avea variabile/clase cu același nume (ele fiind în namespace-uri diferite)

Utilizare:

- using namespace *nume*;
- *nume*::*variabilă*
- *nume*::*clasă*

Unde *nume* este numele namespace-ului, iar *variabilă*, respectiv *clasă*, numele unor variabile, respectiv clase din acel namespace