PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 9

Template

Template

Necesitatea parametrizării tipului de date

- Fie funcția ce realizează suma elementelor unui vector de elemente de tip double

```
double Sum0(double* a, int n)
{
    double res = 0;

    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
    return res;
}
```

Necesitatea parametrizării tipului de date

- Fie un vector cu elemente de tip float. Poate fi utilizată funcția *Sum0* pentru suma elementelor?

```
double a[10];
float  b[10];
// ...
double sa = sum0(a, 10); // OK
float  sb = sum0(b, 10); // eroare de compilare
```

- Soluție clasică:

```
float Sum0(float* a, int n)
{
    float res = 0;

    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
    return res;
}
```

- Cum se poate generaliza această funcție ?

Necesitatea parametrizării tipului de date

- Pentru majoritatea tipurilor de date utilizate ar trebui definită câte o funcție
- Spațiul ocupat de program crește

- Posibilitatea de a trimite tipul de date ca parametru ar permite existența unei singure funcții pentru suma elementelor unui vector

```
template<typename T>
T sum1(T* a, int n)
{
    T res = 0;
    for(int i = 0; i < n; ++i)
    {
        res += a[i];
    }
  return res;
}
```

Necesitatea parametrizării tipului de date

```
double a[10];
float  b[10];
int    c[10];
// ...
double sa = sum1(a, 10); // OK
float  sb = sum1(b, 10); // OK
int    sc = sum1(c, 10); // OK
```

Introducere

- Template-urile (*șabloane*) furnizează suport pentru programare generică utilizând tipurile de date ca parametri
- Mecanismul template al limbajului C++ permite ca un tip de date sau o valoare să fie un parametru în definirea unei clase, funcții sau alias.
- Template-urile furnizează o cale directă de a reprezenta o gamă largă de concepte generale și căi simple de a le combina.
- Rezultatul din punct de vedere al claselor și funcțiilor poate fi un cod mai puțin general la run-time și eficient din punct de vedere al spațiului.
- Template-urile au fost introduse având ca scop proiectarea, implementarea și utilizarea librăriei standard.
- Librăria standard cere un mare grad de generalitate, flexibilitate și eficiență.

Introducere

- În consecință, tehnicile care pot fi utilizate în proiectarea și implementarea libăriei standard sunt eficace și eficiente în proiectarea soluțiilor pentru o mare varietate de probleme.
- Aceste tehnici permite unui programator să ascundă implementări sofisticate în spatele unei simple interfețe și să expună complexitatea utilizatorului atunci când acesta are nevoie și cere acest lucru.

Un simplu template string

- Se consideră un șir de caractere definit ca o clasă ce memorează caractere și furnizează operații precum cautare, comparare, concaternare.
- Se dorește ca acest comportament să poate fi aplicat mai multor tipuri de carcatere: char, unsigned char, caractere chinezești, caractere grecești etc.
- Cu alte cuvinte se dorește reprezentarea noițiunii de „șir de caractere” cu o dependeță cât mai mică de un specific tip de caracter

String template

- Fie clasa String:

```
class String
{
    char *ptr;
    int sz;
public:
    String() : sz(0), ptr(0) {};
    explicit String(const char*p);
    String(const String&);
    char& operator[](int n) { return ptr[n]; }
    String& operator+=(char c);
    String& operator=(const String&);
    ~String() { if (ptr) delete[] ptr; }
};
```

- Clasa gestionează șiruri de caractere de tip char.
- Pentru a deveni o clasă generală se parametrizează tipul de date

String template

```
template<typename T>
class String {
        T *ptr;
        int sz;
public:
        String() : sz(0), ptr(0) {};
        explicit String(const T*p);
        String(const String&);
        T& operator[](int n) { return ptr[n]; }
        String& operator+=(T c);
        String& operator=(const String&);
        String(String&& x);
        ~String() { if (ptr) delete[] ptr; }
};
```

- Prefixul `template<typename T>` specifică faptul că un template este declarat și că tipul argument *T* va fi folosit în declarație
- După introducere, tipul *T* este utilizat ca oricare alt tip de date

String template

- Domeniul tipului de date *T* se extinde până la finalul declarației prefixate de *template<typename T>*.
- Poate fi utilizat și prefixul echivalent *template<class T>*
- În ambele cazuri *T* este un nume de tip de date și nu numele unei clase.
- Utilizare:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

struct jchar { /* ... */ }; // caractere nipone

String<jchar> js;
```

- Cu excepția sintaxei speciale a numelui, obiectul *String <char> cs* are aceeași funcționalitate ca cea definită anterior.
- Făcând *String* un template permite furnizarea facilităților implementate pentru orice tip de caracter

String template

- Libraria standard oferă clasă template pentru manipularea sirurilor de caractere sub numele de *basic_string* similar clasei templetizate prezentate în curs.
- În librăria standard, *string* este un sinonim pentru *basic_string<char>*.

```
using string = std::basic_string<char>;
```

- Astfel pentru secvența *basic_string<char> sir;* poate fi utilizat *string sir;*
- În general aliasurile sunt utile pentru evita denumirile lungi ale claselor generate. De asemenea se preferă a nu se cunoaște detaliile despre cum un tip este definit. Un alias permite ascunderea faptului că un tip este generat dintr-un template

Definirea unui template

- O clasă generată dintr-un template este o clasă normală. Din acest motiv utilizarea unei clase template nu implică nici un mecanism run-time comparativ cu o clasă normală.
- Utilizarea template-urilor conduce la o descreștere a codului generat deoarece codul pentru o funcție membră a unei clase template este generată dacă acel membru este utilizat.
- Înainte de a proiecta o clasă template este preferabil de a proiecta, testa, depana o clasă concretă (String). Astfel o serie de erori de implementare pot fi eliminate în contextul exemplului concret.
- Pentru a putea înțelege generalitatea unui template se va imagina comportarea acestuia pentru un timp concret de date.
- Pe scurt: o componentă generică este dezvoltată ca o generalizare a unui sau mai multe exemple concrete și nu pornind de la principii

Definirea unui template

- Membri unei clase template sunt declarați și definiți exact ca la clasele nontemplate
- Totuși când un membru este declarat în afara clasei trebuie în mod explicit declarat template-ul

```
template<typename T>
String<T>::String() : sz{0}, ptr{ch}
{
    ch[0] = {};
}

template<typename T>
String& String<T>::operator+=(T c)
{
    // ... concaternare
    return *this;
}
```

Definirea unui template

- Parametrul template *T* este un parametru și nu un nume pentru un tip specific. În domeniul *String<T>*, identificatorul *<T>* este numele template-ului. Astfel numele constructorului este *String<T>::String*.
- Nu este posibilă supraîncărcarea numelui unei clase template.

```
template<typename T>
class String { /* ... */ };

class String { /* ... */ }; // eroare
```

- Un tip utilizat ca template trebuie să furnizeze interfața așteptată de template

Instanțierea unui template

- Procesul de generare a unei clase ori funcții dintr-o listă de argumente template este denumit adesea *template instantiation*. O versiune a template-ului pentru o listă specifică de argumente este numită specializare.

```
String<char> cs;

void f(void)
{
    String<jchar> js;
    cs = ”Abcd efg hgi";
}
```

- Pentru secvența de mai sus compilatorul generează declarația claselor `String<char>` și `String<jchar>`, pentru fiecare în parte destructori și constructori cu listă vidă de parametri și funcția operator `String<char>::operator=(char*)`
- Alte funcții membre nefiind folosite nu sunt generate.

- Evident, template-urile furnizează o modalitate de a genera foarte mult cod din relativ mici definiții. În consecință o atenție sporită trebuie avută pentru a nu umple memoria cu definiții de funcții aproape identice.

Derivare

```
template<typename T>
class B
{
    /* ... */
};

template<typename T>
class D : public B<T>
{
    /* ... */
};

template<typename T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb); // f<int>(pb)
    f(pd); // f<int>(static_cast<B<int>*>(pd));
        // conversie standard D<int>* to B<int>*
}
```

Sfaturi

- A se utiliza template-uri pentru
  - a implementa algoritmi ce se adresează mai multor tipuri de argumente
  - a implementa containere
- Cele două declarații template<typename T> și template<class T> sunt sinonime
- Înainte de a proiecta un template se va proiecta și implementa o versiune nontemplate. Ulterior se va generaliza parametrizând tipul/tipurile de date
- O funcție virtuală nu poate fi o funcție template membră
- A se supraîncărca funcțiile template pentru a se obține aceeași semantică pentru mai multe tipuri de argumente ()
- A se utiliza aliasuri de template-uri pentru a simplifica notația și a ascunde detaliile de implementare
- A se include definițiile template în orice fisier sursă unde este necesar

Vă mulțumesc !