Programmazione Generica: Template

Introduzione ai Template

- In C++ e' possibile definire delle funzioni o classi dette *generiche* ovvero che abbiano come *parametro* il *tipo* di dato
- Una funzione (classe) generica definisce una serie di operazioni applicabili ad un qualsiasi tipo di dato
- Ovvero l'algoritmo implementato si applica a qualsiasi tipo
- Una funzione (classe) generica si chiama funzione (classe) template

Sintassi e uso dei Template

- Sintassi:

```
 template<class TIPO> retType NomeFunzione(TIPO);
template<typename TIPO> retType NomeFunzione(TIPO);
```

- Uso con dichiarazione esplicita di tipo:

```
int main(){
int a=2;
cout<<NomeFunzione<int>(a);
return 0;
}
```

- Uso con dichiarazione implicita di tipo:

```
int main(){
int a=2;
cout<<NomeFunzione(a);
return 0;
}
```

Sintassi e uso dei Template

- Esempio:

```
template<class T> void swap(T& a, T& b){
   T temp;
   temp=a;
   a=b;
   b=temp;
}
```

- Uso:

```
int a=1;int b=2;
swap(a,b);
char c_a='x';char c_b='y';
swap(c_a,c_b);
```

Macro e Template

- In C esisteva un modo per realizzare codice indipendente dal tipo: le macro
- Si vuole “riscrivere” del codice sostituendo in modo appropriato le variabili
- Esempio:

```
#define max(a,b)   ((a) < (b) ? (b) : (a))
```

- Svantaggi:
  - Il codice deve stare su una linea (logica)
  - Non e' type safe (il compilatore non avverte di errori di tipo)
  - Sono possibili errori inaspettati
- Vantaggi:
  - Non e' una funzione ma una espansione inline (ma si pou' fare anche con i template basta dichiararli inline) e quindi piu' efficiente in tempo

Errori inaspettati con le Macro

- Se non si presta attenzione all'uso delle parentesi si possono avere errori inaspettati

- Si devono racchiudere tra parentesi gli argomenti

```
#define RADTODEG(x) (x * 57.29578)
```

`RADTODEG(a + b)` diventa `(a + b * 57.29578)`

- Si deve racchiudere tra parentesi l'intera espressione

```
#define RADTODEG(x) (x) * 57.29578
```

`1/RADTODEG(a)` diventa `1/(a) * 57.29578`

Il Compilatore e i Template

- Il compilatore genera tutte le istanze utili della funzione template
- Il procedimento è equivalente ad un overloading automatico
- **Nota:** Vengono generate tutte le istanze che servono, che vengono poi effettivamente utilizzate nel codice esecutivo
- Eventuali errori vengono segnalati al tempo di compilazione

I problemi dei template (in C++)

- Supporto scarso da parte dei compilatori vecchi (protabilita')
- Errori segnalati dai compilatori poco leggibili
- Il volume di codice generato automaticamente puo' essere notevole

- Perche' la versione giusta sia generatada un template e' necessario sapere come deve essere istanziato
- Questa informazione e' presente in genere nel corpo della definizione di una funzione
- Questo obbliga ad avere il codice di definizione nel file header!

Vantaggi dei template

- Overloading: il compilatore sceglie la funzione giusta automaticamente in funzione dei parametri
- Codice generato al momento della compilazione (e conseguente rilevazione di errori al tempo della compilazione)
- Sono type-safe (cioe' il compilatore garantisce il controllo di consistenza dei tipi)
- Si possono specializzare (per gestire casi eccezionali)
- Usano vincoli strutturali “lazy” (vincoli imposti al momento dell'uso)

Genericita' dei Template

- Le funzioni template sono limitate rispetto al caso generale di overloading perché non si possono specificare comportamenti diversi della funzione al variare del tipo

- Perche' una istanziazione di una funzione template per un certo tipo abbia successo si deve garantire che per quel tipo siano definiti tutti gli operatori e le funzioni/dati membro utilizzati nella funzione
- Se per un tipo non esiste l'operatore di confronto e nella funzione template si confrontano due variabili di quel tipo si genera un errore al tempo di compilazione

Differenza overloading/template

```
#include <iostream>

void f(int i){cout<<"il valore è: "<<i;}
void f(char i){cout<<"il carattere è: "<<i;}
template<class T> void g(T i){cout<<"val: "<<i;}

int main(){
  int a=56;
  char b='x';

  f(a);//Stampa: il valore è: 56
  f(b);//Stampa: il carattere è: x

  g(a);//Stampa: val: 56
  g(b);//Stampa: val: x

  return 0;
}
```

Funzioni con più di un tipo generico

- Nel caso in cui si debbano specificare più di due tipi che possono essere diversi:

- Sintassi

```
template<class T1, class T2> retType F(T1,T2);
```

- Esempio:

```
template<class T1, class T2>
void func(T1 x, T2 y){
    cout<<"prima:"<<x<<" poi:"<<y;
}
```

Deduzione automatica del tipo

- Quando viene utilizzata una funzione template è generalmente possibile per il compilatore dedurre automaticamente il tipo

```
template<class T> retType NomeFunzione(argType (T));
```

- altre volte è impossibile:

```
template<class T1,class T2> retType NomeFunzione(T1,T2=0);
```

- in questo ultimo caso infatti si potrebbe avere ambiguità:

```
template<class T1,class T2> void f(T1,T2=0);
void main(){
  int a,b;
  float c,d;

  f(a,b); f(c,d); f(a,d);
  f(a); //caso ambiguo
  f<int,float>(a); //ok
}
```

Uso delle funzioni generiche

```
template<class T> T max(T *v, int size){
  T max=v[0];
   for(int i=1;i<size;i++)
        if(max<v[i]) max=v[i];
   return max;
}

main(){
   int arrayI[7]={4,7,2,4,9,3,2};
   double arrayD[6]={7.1,9.4,2.6,5.7,4.8,6.9}
   int resI;
   double resD;
   resI=max(arrayI,7);
   resD=max(arrayD,6);
}
```

Specializzazione di template

- Un modo migliore e' dire al compilatore che si intende usare una versione specializzata di un template
- Lo si fa utilizzando (dopo la dichiarazione del template generico) la forma template<> e indicando il tipo del parametro

```
template<class T> void g(T i){cout<<"val: "<<i;}
template<> void g<char>(char i)
{cout<<"il carattere è:"<<i;}

int main(){
  int a=56;
  char b='x';
  g(a);//Stampa: il valore è: 56
  g(b);//Stampa: il carattere è: x
  return 0;
}
```

Esempio di Specializzazione

```
template <class T> // primary template
T mymax(const T t1, const T t2)
{
 return  t1 < t2 ? t2 : t1;
}

template <> // specialization
const char* mymax<const char*>(const char* t1, const char* t2)
{
 return (strcmp(t1, t2) < 0) ? t2 : t1;
}

int main()
{
 int highest=mymax(5,10); // #1 calls primary
 char c=mymax ('a', 'z'); // #2 calls primary
 string s1="hello", s2="world";
 string maxstr=mymax(s1,s2); // #3 calls primary
 const char *arr1="hello";
 const char *arr2="world";
 const char * p=
  mymax (arr1, arr2);  // #4  calls specialization
}
```

Quando si usa la specializzazione?

- Quando si deve gestire un caso particolare
  - Max fra puntatori
- Quando per ragioni di efficienza si possono usare accorgimenti particolari
  - uso di contenitori con indirizzamento diretto

Overloading esplicito

- Se si esegue un overloading esplicito di una funzione template questa maschera quella generata implicitamente

```
template<class T> void g(T i){cout<<"val: "<<i;}
void g(char i){cout<<"il carattere è: "<<i;}

int main(){
  int a=56;
  char b='x';
  g(a);//Stampa: il valore è: 56
  g(b);//Stampa: il carattere è: x
  return 0;
}
```

Overloading e specializzazione

- Nella specializzazione di template si usa il meccanismo di risoluzione del template
- Nell'overloading esplicito si usa una funzione non template che ha la stessa struttura di una funzione template
- Nell'overloading esplicito vengono anche usate funzioni di conversione implicita dei tipi mentre nella specializzazione no
- Es:
  - double max(double,double) puo' funzionare anche quando uno dei parametri e' un int attuando una conversione implicita
  - Diventa problematico qui usare i template perche' si dovrebbero avere due tipi generici e non si sa cosa usare come tipo restituito!

Esercizi

```
template<typename T1, typename T2>
void f( T1, T2 );                          // 1
template<typename T> void f( T );          // 2
template<typename T> void f( T, T );       // 3
template<typename T> void f( T* );         // 4
template<typename T> void f( T*, T );      // 5
template<typename T> void f( T, T* );      // 6
template<typename T> void f( int, T* );    // 7
template<> void f<int>( int );             // 8
void f( int, double );                     // 9
void f( int );                             // 10
```

```
int                 i;
double              d;
float               ff;
complex<double>     c;

f( i );             // a
f<int>( i );        // b
f( i, i );          // c
f( c );             // d
f( i, ff );         // e
f( i, d );          // f
f( c, &c );         // g
f( i, &d );         // h
f( &d, d );         // i
f( &d );            // j
f( d, &i );         // k
f( &i, &i );        // l
```

Classi Template

- Una classe generica o template può definire i propri membri in modo generico
- Sintassi in dichiarazione:

```
template<class T> class NomeClasse{};
```

- Sintassi in definizione:

```
template<class T>
retType NomeClasse<T>::funcName(argType parameter){}
```

- Sintassi in uso:

```
main(){
    NomeClasse<type> obj(init);
}
```

Deduzione per Classi Template

- Per le classi si deve sempre esplicitare il tipo nella dichiarazione
- Non ci sono meccanismi di deduzione automatica
- In generale infatti non e' necessario che un costruttore abbia parametri dei tipi opportuni

Esempio

```
//Vettore generico
template<class T>
class Vector{
public:
   Vector(int usr_size=10){size=usr_size; v=new T[size];}
   ~Vector(){delete [] v;}
   T& operator[](int);
private:
   int size;
   T* v;
};

template<class T>
T& Vector<T>::operator[](int i){
   return v[i];
}
```

Esempio

```
void main{
   Vector<int> vI(100);
   Vector<char> vC(5);

   int i;
   for(i=0;i<100;i++)
        vI[i]=i*i;

   for(i=0;i<5;i++)
        vC[i]='e';

   for(i=0;i<100;i++)
        cout<<vI[i]<<" ";
   cout<<endl;
}
```