UNIVERSITÀ
DI PARMA

programmazione generica

Alberto Ferrari

definizione

- ***generic programming*** (programmazione generica) è uno ***stile*** di programmazione in cui gli ***algoritmi*** sono scritti a un ***alto livello*** di astrazione ***indipendentemente*** dal ***tipo*** di dati su cui questi operano
- si tratta di un ***concetto comune*** che i vari linguaggi definiscono con ***termini*** e ***implementazioni*** differenti
- ***generics***
  - Ada, Eiffel, Java, C#, VisualBasic.NET
- ***polimorfismo parametrico***
  - ML, Scala, Haskell
- ***templates***
  - C++

generic programming

- ***funzioni*** (metodi) e ***tipi*** (classi) che ***differiscono*** solo per i ***tipi di dato*** su cui operano
- è un modo per rendere un linguaggio ***più espressivo*** e ***ridurre*** la ***duplicazione*** del codice
- gli ***algoritmi*** sono scritti in termini di ***tipi generici***
- i ***tipi*** vengono passati come ***parametri***
- ***funzione generica***
  - esegue la ***stessa operazione*** su ***diversi tipi*** di dato
- ***tipo generico*** (*classe*)
  - memorizza i ***valori*** ed esegue ***operazioni*** su ***diversi tipi*** di dato

funzioni generiche in C++

- ***funzione generica***
  - esegue la ***stessa operazione*** su diversi tipi di dati
  - ***astrazione*** algoritmica
- come implementare una funzione generica in C ++
  - ***overloading***
  - ***puntatori void***
  - ***templates***
- esempio:
  - *scambia il valore di due variabili*

overloading: insieme di metodi

- con lo ***stesso nome***
- con ***firma diversa***

```
void my_swap (int &f, int &s ) {
     int tmp = f; f=s; s=tmp;
}

void my_swap (string &f, string &s ) {
     string tmp = f; f=s; s=tmp;
}
```

compile time error
no know conversion from double to &int ...

```
int main() {
   string a, b; a = "hello"; b = "world";
   cout <<"before a="<<a<<" b="<<b<<endl;
   my_swap (a,b);
   cout << "after a="<<a<<" b="<<b<<endl;

   int x, y; x = 33; y = 44;
   cout <<"before x="<<x<<" y="<<y<<endl;
   my_swap(x,y);
   cout <<"after x="<<x<<" y="<<y<<endl;

   double d1, d2; d1 = 3.3; d2 = 4.4;
   cout << "before d1="<<d1<<"d1="<<d2<<endl;
   my_swap(d1,d2);
   cout <<"after d1="<<d1<<"d2="<<d2<<endl;
   return 0;
}
```

funzioni generiche - void pointers

- possiamo scrivere una ***funzione*** che accetta un ***puntatore void*** come argomento e utilizzarlo passando un puntatore di ***qualsiasi tipo***
- la funzione è più ***generica***
- è necessario un ***casting*** da un puntatore void a un puntatore di un tipo specifico

```
void my_swap (void* &f, void* &s ) {
      void* tmp = f;
      f=s;
      s=tmp;
 }
```

no compile time error, no runtime error
output a = 1919907594 :(

```
int main() {
    void* a; void* b;
    a = new std::string("hello");
    b = new std::string("world");
    cout<<*((string*) a)<<*((string*) b)<<endl;
    my_swap (a,b);
    cout<<*((string*) a)<<*((string*) b)<<endl;
    void* x; void* y;
    x = new int(33); y = new int(44);
    cout <<  *((int*) x) <<  *((int*) y) << endl;
    my_swap(x,y);
    cout <<  *((int*) x) <<  *((int*) y) << endl;

    cout <<"a ="<< *((int*) a) <<endl;
    cout <<"a ="<< *(static_cast<int*>(a))<<endl;
    return 0;
}
```

funzioni generiche - templates

- aggiungiamo un ***parametro*** di ***tipo*** alla funzione

```
template <class T>
void my_swap(T& f, T& s) {
    T tmp = f;
    f = s;
    s = tmp;
}
```

```
int main() {
    int a = 3; int b = 4;
    cout<<"before a ="<<a<<" b ="<<b<<endl;
    my_swap<int> (a,b);
    cout<<"after  a ="<<a<<" b ="<<b<<endl;

    string s1 = "hello";
    string s2 = "world";
    cout<<"bef. s1="<<s1<<"s2="<<s2<<endl;
    my_swap<string> (s1,s2);
    cout<<"after s1="<<s1<<"s2="<<s2<<endl;

    return 0;
}
```

compilatore e template

- una funzione template corrisponde a un ***insieme*** di definizioni di funzione
- il ***compilatore*** ne produce ***una per ogni tipo*** per cui si usa il template
- è possibile avere template di funzioni con ***più parametri*** di tipo
  - `template<class T1, class T2>`
  - oppure
  - `template<typename T1, typename T2>`

esempio: elemento centrale array

```
template <typename T>
T centralElement(T data[], int cont)
{
    return data[cont/2];
}
```

T must be a type → primitive type, class

```
int i[] = {10,20,30,40,50};
int ci = centralElement(i,5);

string s[] = {"alpha","beta","gamma"};
string cs = centralElement(s,3);

float f[] = {2.2,3.3,4.4};
float cf = centralElement<float>(f,3);
```

Type parameters are **inferred** from the values in a function invocation
Or **explicitly** passed as type parameter

classi template

- ***classi*** che possono operare su valori di ***tipo differente***
- sono generalmente utilizzate per implementare ***contenitori***
- l'***istanziazione*** degli oggetti avviene specificando il ***tipo*** come ***parametro***

esempio Pair (coppia di oggetti)

```
template <typename F, typename S>
class Pair
{
public:
    Pair(const F& f, const S& s);
    F get_first() const;
    S get_second() const;
private:
    F first;
    S second;
};
```

```
template <typename F, typename S>
Pair<F,S>::Pair(const F& f, const S& s) {
    first = f;
    second = s;
};

template <typename F, typename S>
F Pair<F,S>::get_first() const {
    return first;
};

template <typename F, typename S>
S Pair<F,S>::get_second() const {
    return second;
};
```

istanziazione

```
Pair<int,double> p1(2,3.4);
int p1_first = p1.get_first();
double p1_second = p1.get_second();


Pair<string,int> p2("alpha",5);
string p2_first = p2.get_first();
int p2_second = p2.get_second();
```

- quando dichiariamo un variabile di una classe template ***dobbiamo specificare*** i parametri di ***tipo***
  - *C++17 ha la template deduction dei costruttori*

Lista generica (Nodo.h)

```
// Forward reference
template <typename E> class Lista;

template <typename E>
class Nodo {
private:
   E info;
   Nodo * next;
public:
   Nodo(E e) : info(e), next(nullptr){};
   E getInfo() const { return info; };
   Nodo * getNext() const { return next; }
   friend class Lista<E>;
};
```

- `template <typename E> class Lista;`
  - *necessario per dichiarare Lista<E> come classe friend*
- l'informazione associata al nodo è di tipo generico E

Lista generica (Lista.h)

```
template <typename E> std::ostream & operator<<(std::ostream & os,const Lista<E> & lst);
template <typename E>
class Lista {
private:
   Nodo<E> * testa;
public:
   Lista(); ~Lista();
   void insTesta(const E &value);
   void insCoda(const E &value);
   bool elimTesta(E &value);
   bool elimCoda(E & value);
   bool vuota() const;
   Lista<E>* operator+(Lista<E> altraLista);
   friend std::ostream & operator<< <>(std::ostream & os, const Lista<E> & lst);
};
```

Lista generica (implementazione e utilizzo)


```
template <typename E>
Lista<E>::Lista() : testa(nullptr) { }

template <typename E>
Lista<E>::~Lista() {
   while (testa) {
      Nodo<E> * temp = testa;
      testa = testa->next;
      delete temp;
   }
}

. . .
```

```
template <typename E>
void Lista<E>::insTesta(const E &value) {
   Nodo<E> * newNodo = new Nodo<E>(value);
   newNodo->next = testa;
   testa = newNodo;
}
. . .
. . .
int main(){
    Lista<string> lis;
    string s;
    lis.insTesta("hello");
    cout << "lis = " << lis << endl;
…
}
```