

Polimorfismo

- “la capacità di assumere forme diverse” (Webster’s Dictionary)
- **funzioni polimorfe**: possono essere applicate ad argomenti di vario tipo
- in ML, abbiamo “polimorfismo parametrico”, in cui le espressioni di tipo sono parametrizzate (ad esempio, gestione di liste omogenee, ma con tipo base parametrico)
- C++ permette il polimorfismo parametrico per mezzo del concetto di **template** (cenni più avanti)
- C++ prevede anche un altro tipo di polimorfismo: più definizioni per uno stesso nome di funzione — **sovrapposizione, sovraccarico (overloading)**
- Abbiamo già visto una applicazione di questo concetto: i costruttori
- oltre alle funzioni definite nel programma, possono essere ridefiniti (o estesi) gli operatori del linguaggio

61

Sovrapposizione di operatori (infissi)

- nei linguaggi tradizionali si definiscono operatori che “estendono” gli operatori predefiniti (ad esempio “+” infisso) con nome e notazione molto diversa (ad esempio Somma(..))
- la sovrapposizione degli operatori (e la definibilità per ogni tipo) in C++ permette di superare l’asimmetria
- per ogni operatore θ , l’espressione $o_1\theta o_2$ è un’abbreviazione per $o_1.operator\theta(o_2)$
- Esempio: $a + b$ sta per $a.operator+(b)$
- gli operatori di assegnazione (=), indirizzo di (&) e virgola sono predefiniti per ogni tipo, ma possono essere ridefiniti; gli altri vanno definiti esplicitamente

62

```
#include <iostream.h>
class NumeroComplesso {
public:
    void stampa() { cout << Re << '+' << Im << 'i' << endl; }
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso Somma(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re;
        c1.Im = c.Im + Im;
        return c1;
    }
private :
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};

void main() {
    NumeroComplesso c1(-3.2,2.0);
    NumeroComplesso c2(4.2,1.5);
    NumeroComplesso c3, c4;
    c1.stampa();
    c2.stampa();
    c3 = c2.Somma(c1);
    c3.stampa();
}
```

63

```
#include <iostream.h>
class NumeroComplesso {
public:
    void stampa() { cout << Re << '+' << Im << 'i' << endl; }
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso operator+(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re;
        c1.Im = c.Im + Im;
        return c1;
    }
private :
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};

void main() {
    NumeroComplesso c1 (-3.2,2.0);
    NumeroComplesso c2 (4.2,1.5);
    NumeroComplesso c3 , c4;
    c1.stampa();
    c2.stampa();
    c3 = c1 + c2;
    c3.stampa();
    c4 = c2.operator+(c1); //equivalente
    c4.stampa();
}
```

64

Assegnazione e costruttore di copia

- sono anch'essi funzioni del C++, predefinite
- le loro definizioni standard (di default) sono uguali, con assegnazione alle componenti omologhe
- vengono richiamati in contesti diversi:
 - l'assegnazione: usuale; ad esempio:
`NumeroComplesso c2;
c2 = c1;`
 - il costruttore di copia:
 - * nelle inizializzazioni, ad esempio:
`NumeroComplesso c2 = c1;`
 - * nel passaggio di parametri per valore
 - * nella restituzione di risultati delle funzioni

65

- C++ permette di ridefinire anche l'operatore di assegnazione e il costruttore di copia;
- quando ciò può essere utile? ad esempio per la gestione “profonda” di liste (senza condivisione)
- assegnazione e costruttore di copia sono distinti proprio perché le esigenze, nei rispettivi contesti, possono essere diverse, e quindi, ove necessario, possono essere ridefiniti separatamente

66

Assegnazione

- è un operatore che modifica l'oggetto di invocazione (“side-effect”) e ha un argomento, dello stesso tipo, che viene passato per riferimento e non viene modificato;

- ad esempio, l'assegnazione su interi:

```
int& operator=(const int&)
```

- il risultato viene:
 1. assegnato all'oggetto di invocazione
 2. restituito dalla funzione =, che può quindi essere composta: `a = b = c`
- la sovrapposizione dell'assegnazione per una classe C si dichiara con

```
C& operator=(const C&)
```

67

Ridefinizione dell'assegnazione

```
#include <iostream.h>

class cell {
friend class list;
    cell(int i) { info = i; next = NULL; }
    cell(int i, cell* n) { info = i; next = n; }
    int  info;
    cell* next;
};

class list {
private:
    cell* first;
    void destroy(cell *p) { cell* tmp; while(p) { tmp = p; p = p->next; delete tmp; } }
    cell* duplicate(cell* p) {
        if (p == NULL) return NULL;
        else { cell* tmp = new cell(p->info,duplicate(p->next)); return tmp; }
    }
public:
    list() { first = NULL; } // costruttore
    ~list() { destroy(first); first=NULL; } // distruttore
    list& operator=(const list& l) { // ridefinizione dell'assegnazione
        destroy(first); // distrugge la lista "vecchia"
        first = duplicate(l.first); // crea una copia
        return *this;
    }
    void insert(int i) { cell* p; p = new cell(i,first); first = p; }
```

68

```

void togliSecondo(){
    if ((first != NULL) && (first->next != NULL)) {
        cell* aux = first->next; first->next = first->next->next;
        delete aux;
    }
}

void print() {
    cell* p = first;
    while (p != NULL) { cout << p->info << " "; p = p->next; } cout << endl;
}

};

main() {
    list a; a.insert(2); a.insert(1);
    a.print(); // 1 2
    list b; b = a; b.insert(5); b.print(); // 5 1 2
    list c(b); c.print(); // 5 1 2
    a = b; b.togliSecondo(); b.print(); // 5 2
    c.print(); // 5 2 lo ha tolto anche a c
    b.insert(4); b.print(); // 4 5 2
    a.print(); // 5 1 2
    b.togliSecondo(); // errore !
    b.print(); // 4 2
    c.print(); // errore in esecuzione
    b.togliSecondo(); b.print(); // 4
    c.print(); // errore in esecuzione
} // produce un errore di esecuzione anche il distruttore

```

69

Ridefinizione del costruttore di copia

- il costruttore di copia:
 - è un costruttore (quindi una funzione con lo stesso nome della classe)
 - monadico (unario)
 - con argomento dello stesso tipo della classe, non modificabile
- quindi la sovrapposizione del costruttore di copia per una classe C si dichiara con

C (const C&)
- il parametro è passato per riferimento costante (così non genera copie)
- la ridefinizione è in genere diversa da quella dell'assegnazione (spesso più semplice; i distruttori completano il quadro)

70

```

#include <iostream.h>

class cell {
friend class list;
    cell(int i) { info = i; next = NULL; }
    cell(int i, cell* n) { info = i; next = n; }
    int info;
    cell* next;
};

class list {
private:
    ... // come prima
public:
    list() { first = NULL; } // costruttore
    ~list() { destroy(first); first=NULL; } // distruttore
    list& operator=(const list& l) { // ridefinizione dell'assegnazione
        destroy(first); // distrugge la lista "vecchia"
        first = duplicate(l.first); // crea una copia
        return *this;
    }
    list(const list& l) { first = duplicate(l.first); } // ridefinizione del costruttore di copia
    ... // come prima
}

```

71

```

main() {
    list a; a.insert(2); a.insert(1);
    a.print(); // 1 2
    list b; b = a; b.insert(5); b.print(); // 5 1 2
    list c(b); c.print(); // 5 1 2
    a = b; b.togliSecondo(); b.print(); // 5 2
    c.print(); // 5 1 2
    b.insert(4); b.print(); // 4 5 2
    a.print(); // 5 1 2
    b.togliSecondo(); //
    b.print(); // 4 2
    c.print(); // 5 1 2
    b.togliSecondo(); b.print(); // 4
    c.print(); // 5 1 2
}

```

72

Polimorfismo parametrico: i template

- nei linguaggi tradizionali, strutture simili che differiscano solo al livello di “tipo componente” richiedono definizioni diverse, anche se molto simili
- vediamo alcuni esempi
 - pila di interi
 - pila di caratteri
 - pila di numeri complessi

73

Pila di caratteri

```
#include <iostream.h>

class stack_char {
    char* v;
    char* p;
    int maxsize;
public:
    stack_char(int s) { v = p = new char[maxsize=s]; }
    ~stack_char() { delete[] v; }
    void push (char el) { *p = el; p++; }
    char pop() { p--; return *p; }
    int size() { return p-v; }
    void stampa() { // stampa dall'affiorante verso l'interno
        char* i = p - 1;
        while (i >= v) { cout << (*i) << ' '; i--; };
        cout << endl;
    }
};

main(){
    stack_char sc(100); // stack di caratteri
    sc.push('Z'); sc.push('Y'); sc.push('X');
    sc.stampa();
    char c;
    c = sc.pop();
    sc.stampa();
}
```

74

Pila di interi

```
#include <iostream.h>

class stack_int {
    int* v;
    int* p;
    int maxsize;
public:
    stack_int(int s) { v = p = new int[maxsize=s]; }
    ~stack_int() { delete[] v; }
    void push (int el) { *p = el; p++; }
    int pop() { p--; return *p; }
    int size() { return p-v; }
    void stampa(){ // stampa dall'affiorante verso l'interno
        int* i = p - 1;
        while (i>=v){ cout << (*i) << ' '; i--; };
        cout << endl;
    }
};

main(){
    stack_int sc(100); // stack di interi
    sc.push(3); sc.push(2); sc.push(1);
    sc.stampa();
    int c;
    c = sc.pop();
    sc.stampa();
}
```

75

Classi e funzioni parametrizzate in C++: i template

- C++ permette di definire parametricamente (rispetto a tipi) funzioni e classi

template <lista argomenti_parametrici > dichiarazione

- esempio, funzione parametrizzata (funTempl.cpp):

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

- classi parametrizzate: molto utili

76

Pila parametrica

```
#include <iostream.h>

template <class T>class stack {
private:
    T* v;
    T* p;
    int maxsize;
public:
    stack(int s) { v = p = new T[maxsize=s]; }
    ~stack() { delete[] v; }
    void push (T el) { *p = el; p++; }
    T pop () { p--; return *p; }
    int size() { return p-v; }
    void stampa(){ // stampa dall'affiorante verso l'interno
        T* i = p-1;
        while (i >= v){
            cout << (*i) << ' ';
            i--;
        }
        cout << endl;
    }
};
```

77

```
class NumeroComplesso {
public:
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso operator+(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re;
        c1.Im = c.Im + Im;
        return c1;
    }
    float Reale() { return Re; }
    float Immaginaria() { return Im; }

private:
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};

ostream& operator<<(ostream& os, NumeroComplesso& c) {
    return os << '(' << c.Reale() << ',' << c.Immaginaria() << ')';
}
```

78

Sovrapposizione e funzioni esterne

```
ostream& NumeroComplesso::operator<<(ostream& os) {
// overloading dell'operatore "<<" come funzione interna
    return os << Reale() << '+' << Immaginaria() << 'i';
}

ostream& operator<<(ostream& os, NumeroComplesso& c) {
// overloading dell'operatore "<<" come funzione esterna:
// permette di utilizzarlo in maniera simile alle istruzioni
// di output "standard"
    return os << c.Reale() << '+' << c.Immaginaria() << 'i';
}
```

79

```
main(){
    stack<char> sc(100); // stack di caratteri
    sc.push('Z'); sc.push('Y'); sc.push('X');
    sc.stampa();
    char c;
    c = sc.pop();
    sc.stampa();

    stack<int> si(100); // stack di interi
    si.push(3); si.push(2); si.push(1);
    si.stampa();
    int i;
    i = si.pop();
    si.stampa();

    stack<NumeroComplesso> scx(100); // stack di complessi
    scx.push(NumeroComplesso(3,3));
    scx.push(NumeroComplesso(2,2));
    scx.push(NumeroComplesso(1,1));
    scx.stampa();
    NumeroComplesso cx;
    cx = scx.pop();
    scx.stampa();
}
```

80

```

#include <iostream.h>

template <class T> struct elem {
    elem<T>* next;
    T info;
};

template <class T> class stack {
private:
    elem<T>* p;
    void destroy(elem<T>* p) { elem<T>* tmp; while(p) { tmp = p; p = p->next; delete tmp; } }
public:
    stack(int s);
    ~stack() { destroy (p); }
    void push (T el);
    T pop ();
    int size();
    void stampa(){ // stampa dall'affiorante verso l'interno
        elem<T>* paux = p;
        while (paux) {
            cout << paux->info << ' ';
            paux=paux->next;
        }
        cout << endl;
    }
};

template<class T> stack<T>:: stack(int s){ p = NULL; } //s inutilizzato

```

81

```

template<class T>
void stack<T>::push(T el) {
    elem<T>* paux = new elem<T>;
    paux ->info = el;
    paux ->next = p;
    p = paux;
}

template<class T>
T stack<T>::pop() {
    if (p != NULL) {
        T val = p->info;
        elem<T>* paux = p;
        p = p->next;
        delete paux;
        return val;
    }
}

```

82

```

class NumeroComplesso {
public:
    NumeroComplesso(float r, float i) { Re = r; Im = i; }
    NumeroComplesso() { Re = 0.; Im = 0.; }
    NumeroComplesso operator+(NumeroComplesso c) {
        NumeroComplesso c1;
        c1.Re = c.Re + Re;
        c1.Im = c.Im + Im;
        return c1;
    }
    float Reale() { return Re; }
    float Immaginaria() { return Im; }

private :
    float Re; // parte reale
    float Im; // coeff dell'immaginario
};

ostream& operator<<(ostream& os, NumeroComplesso& c) {
    return os << '(' << c.Reale() << ', ' << c.Immaginaria() << ')';
}

```

83

```

main(){

    stack<char> sc(100); // stack di caratteri
    sc.push ('Z');
    sc.push ('Y');
    sc.push ('X');
    sc.stampa();
    cout << "tolgo" << ' ' << sc.pop () << endl;
    sc.stampa();

    stack<int> si(100); // stack di interi
    si.push (3);
    si.push (2);
    si.push (1);
    si.stampa();
    cout << "tolgo" << ' ' << si.pop () << endl;
    si.stampa();

    stack<NumeroComplesso> scx(100); // stack di complessi
    scx.push (NumeroComplesso(3,3));
    scx.push (NumeroComplesso(2,2));
    scx.push (NumeroComplesso(1,1));
    scx.stampa();
    cout << "tolgo" << ' ' << scx.pop () << endl;
    scx.stampa();

}

```

84