Aspetti avanzati delle classi nel linguaggio C++

prof. Riccardo Torlone
Università di Roma Tre

Classi e funzioni parametriche

Classi e funzioni possono essere resi parametrici rispetto a uno o più tipi. Si fa uso della parola chiave **template**

```
template <class T>
T max (T a, T b) {
        if (a > b) return a;
        else return b;
}
```

Classi parametriche (un esempio già visto)

```
#include <iostream.h>
const int N=100;
template <class T> class collezione {
    T v[N];
    int nelem;
public:
    collezione() { nelem=0; }
    collezione(T x) {int i; for(i=0;i<N;i++) v[i]=x; nelem=0; }
    void aggiungi (T x) { v[nelem]=x; nelem++; }
    T togli() {T x=v[nelem]; nelem--; return x; }
    int numeroElementi() { return  nelem; }
    void stampa() { // stampa tutti gli elementi della collezione
        int i; for(i=0;i<nelem;i++) cout << v[i]; cout << endl;
    }
};
```

Istanziazioni della classe parametrica

```
main( ) {
        collezione<char> cc('X'); // collezione di caratteri
        cc.aggiungi('A'); cc.aggiungi('B'); cc.aggiungi('C');
        cc.stampa();
        char c;
        c = cc.togli();
        cc.stampa();

        collezione<int> ci(0); // collezione di interi
        ci.aggiungi(1); ci.aggiungi(2); ci.aggiungi(3);
        ci.stampa();
        int n = ci.togli();
        ci.stampa();
}
```

Un esempio più complesso

- Vogliamo gestire liste (collezioni di dimensioni non note a priori) parametriche rispetto al tipo dell'elemento mediante strutture collegate
- Incominciamo a definire l'elemento di una lista; usiamo una struct

```
#include <iostream.h>
template<class T> // elemento della lista
struct elemento {
   T info;
   elemento<T>* next;
};
```

La classe listagenerica (parametrica rispetto a T)

```
template<class T> class listagenerica {
   public:
      listagenerica();      // costruttore base
      listagenerica<T>& operator=(const listagenerica<T>&); // overl. di  =
      listagenerica(const listagenerica<T>&); // overl. del costr. di copia
      ~listagenerica();    // distruttore
      bool empty();        // test di lista vuota
      void add(T);         // aggiunge un elemento
      T del();             // elimina il primo elemento
      void elim(T);        // elimina l'elemento passato in input
      void stampa();       // stampa la lista
   private:
      elemento<T>* primo; // puntatore al primo elemento ella lista
      elemento<T>* Copia(elemento<T>*); // restituisce una copia della lista
      void Distruggi(elemento<T>* p); // dealloca la lista
};
```

Costruttore e overloading di ass. e costr. di copia

```
template<class T>  listagenerica<T>::listagenerica() {  primo = NULL;  }
template<class T>
listagenerica<T>& listagenerica<T>::operator=(const listagenerica<T>& b) {
        if (b.primo != primo) {
                elemento<T>* temp = primo;
                primo = Copia(b.primo);
                Distruggi(temp);
        }
        return *this;
}
template<class T>
listagenerica<T>::listagenerica(const listagenerica<T>& b) {
        primo = Copia(b.primo);
}
```

Definizioni del distruttore e di empty e add

```
template<class T>
listagenerica<T>::~listagenerica() { Distruggi(primo); }

template<class T>
bool listagenerica<T>::empty() { return primo == NULL; }

template<class T>
void listagenerica<T>::add(T t){
        elemento<T>* aux = new elemento<T>;
        aux->info = t;
        aux->next = primo;
        primo = aux;
}
```

Definizioni delle funzioni del e stampa

```
template<class T> T listagenerica<T>::del() {
    if (primo != NULL) {
        T auxT = primo->info;
        elemento<T>* aux = primo;
        primo = primo->next;
        delete aux; return auxT;
    }
}
template<class T> void listagenerica<T>::stampa() {
    elemento<T>* temp = primo;
    cout << "{ ";
    while (temp!= NULL) { cout << temp->info << " "; temp = temp->next; }
    cout << "}" << endl;
}
```

Definizione funzione elim

```
template<class T> void listagenerica<T>::elim(T t) {
        elemento<T>* aux = new elemento<T>;
        aux->next = primo; primo = aux;
        while (aux->next != NULL) {
                if (aux->next->info == t) {
                        elemento<T>* aux2 = aux->next;
                        aux->next = aux->next->next;
                        delete aux2; aux2 = primo;
                        primo = primo->next; delete aux2; return;
                }
                aux = aux->next;
        }
        aux = primo; primo = primo->next; delete aux;
}
```

Definizioni di Copia e Distruggi

```
template<class T>
elemento<T>* listagenerica<T>::Copia(elemento<T>* sorgente) {
    if (sorgente == NULL) return NULL;
    else {
        elemento<T>* aux = new elemento<T>;
        aux->info = sorgente->info;
        aux->next = Copia(sorgente->next); return aux;
    }
}
template<class T> void listagenerica<T>::Distruggi(elemento<T>* punt) {
    while (punt != NULL) {
        elemento<T>* temp = punt; punt = punt->next; delete temp;
    }
}
```

Esempi di uso della classe listagenerica

```
void main() {
        listagenerica<int> li; // lista di interi
        li.add(2);
        li.add(3);
        int c=li.del();
        li.stampa();

        listagenerica<char> lc; // lista di caratteri
        lc.add('a'); lc.add('b'); lc.add('c');
        lc.elim('b');
        lc.stampa();
}
```

Derivazione in C++

- E’ un meccanismo che permette di definire nuove classi a partire da classi già definite
- Proprietà di base:
  - Le proprietà (dati e funzioni) di una classe vengono ereditate dalle classi derivate da essa;
  - il tipo della classe derivata è “compatibile” con il tipo della classe base
    - *compatibilità*: gli oggetti della classe derivata sono utilizzabili dove è previsto che compaiano oggetti della classe base (per esempio nei parametri delle funzioni);
  - il tipo della classe base non è compatibile con il tipo della classe derivata

Un semplice esempio

```
class Persona {
        char Nome[20];
        int Eta;
public:
        Persona(char n[20], int e) { strcpy(Nome,n); Eta=e; }
        void outDati(char n[20], int& e) { strcpy(n,Nome); e=Eta; }
};

// Funzione esterna che ha come parametro un oggetto Persona
void stampaDati(Persona p) {
        char n[20];
        int e;
        p.outDati(n,e);
        cout << n << “ ha " << e << << “ anni " endl;
}
```

Derivazione di Persona

```
class Studente : public Persona {
        int Matricola; // eredita da Persona le proprietà Nome ed Età,
                       // il costruttore e il metodo outDati
public:
        Studente(char n[20], int e, int m):Persona(n,e) { Matricola=m; }
};

main( ) {
        Persona p("Giorgio Berti",25);
        Studente s("Marina Guali",26,33333);
        stampaDati(p);  // Giorgio Berti ha 25 anni
        stampaDati(s);  // Marina Guali ha 26 anni
}
```

Elementi privati e derivazione

Gli elementi privati restano tali anche per le classi derivate

```
class Persona{
    private:
        int Anni;
};
class Bambino : public Persona {
    private:
        int Mesi;
    public:
        void stampaEta() {
            if (Anni > 0 ) cout << Anni <<" anni e " ;  // Errore!!!!!
            cout << Mesi <<" mesi";
        }
};
```

Campi protected

- Esiste un qualificatore di campi di classe che si usa per ovviare a questo inconveniente: **protected**;
- I campi protetti di una classe sono accessibili solo dalla classe stessa e dalle classi derivate;
- Quindi i campi per una classe C possono essere:
  - pubblici: visibili per tutti
  - privati: visibili solo all'interno di C
  - protetti: visibili per le classi derivate da C (per le altre classi è come se fossero privati)

Elementi protetti

```
class Persona{
   protected:
          int Anni;
};
class Bambino : public Persona {
   private:
          int Mesi;
   public:
          void stampaEta() {
                 if (Anni > 0 ) cout << Anni <<" anni e " ;    // OK!
                 cout << Mesi <<" mesi";
          }
};
```

Tipologie di derivazione

- Le derivazioni possono essere **pubbliche**, **protette** o **private**. Cambiano solo i livelli di accesso delle varie parti della classe:
  - La derivazione pubblica lascia inalterati i livelli di accesso;
  - La derivazione protetta trasforma in protetti i campi pubblici e lascia inalterati gli altri;
  - La derivazione privata trasforma in privati i campi pubblici e protetti e lascia inalterati i privati;

Accessibilità campi nei vari tipi di derivazione

Livelli di accesso nella classe derivata

Classe base	Derivazione pubblica	Derivazione protetta	Derivazione privata
Campo privato	inaccessibile	inaccessibile	inaccessibile
Campo protetto	protetto	protetto	privato
Campo pubblico	pubblico	protetto	privato

Uso dei vari tipi di derivazione

Derivazione pubblica o privata?

- La derivazione pubblica permette di estendere un tipo, introducendo nuove proprietà (abbiamo visto esempi)
- La derivazione privata (o protetta) permette di utilizzare la classe base come supporto per la rappresentazione della classe derivata (nascondendo i dettagli realizzativi della classe base, ovvero del supporto)
- Con il meccanismo delle **classi astratte** (più avanti) potremo realizzare tipi astratti con diverse implementazioni e utilizzo indipendente dalla rappresentazione

Esempio di uso della derivazione privata

- Supponiamo di avere a disposizione una classe che implementa le liste
- Tale lista è realizzata con record e puntatori
- Definizione del record della lista

```
struct elemento {
friend class lista;
        int info;
        elemento* next;
};
```

- E' definita friend perché è al servizio della classe Lista

Dichiarazione della classe lista

```
class lista { // Prototipo della classe (quanto ci basta per usarla)
public:
        lista();    // costruttore base
        lista& operator=(const lista&); // overloading dell'operatore "="
        lista(const lista&); // overloading del costr. di copia
        ~lista();  // distruttore
        bool empty();      // test di lista vuota
        void addFront(int); // aggiunge un elemento in testa
        void addBack(int); // aggiunge un elemento in coda
        int  out(); // estrae ed elimina l'elemento in testa
        void stampa(); //stampa la lista
private:
        elemento* primo; // puntatore al primo elemento
        elemento* Copia(elemento*); // restituisce una copia della lista
        void Distruggi(elemento* p); // rilascia la lista
};
```

Code e Pile definite usando lista come supporto

```
// nella coda il primo ad uscire (essere servito) è il primo che è entrato
class coda : private lista {
  public:
        coda () {lista();}
        void inCoda(int i) {addBack(i); };
        int  outCoda() { return out(); };
};

// nella pila il primo il primo ad uscire (essere servito) è l'ultimo che è entrato
class pila : private lista {
  public:
        void inPila(int i) { addFront(i); };
        int  out() { return lista::out(); }; // ridefinizione di out()
};
```

Uso delle classi Pila e Coda

```
main() {
        coda q;
        pila s;
        q.inCoda(1);
        q.inCoda(2);
        q.inCoda(3);
        s.inPila(1);
        s.inPila(2);
        s.inPila(3);
        cout << q.outCoda() << q.outCoda() << q.outCoda() << endl; // 1 2 3
        cout << s.out() << s.out() << s.out() << endl; // 3 2 1
}
```

Altre caratteristiche della derivazione

- All'invocazione del costruttore di una classe derivata viene eseguito automaticamente anche il costruttore senza argomenti della classe base;
- Il distruttore si comporta in maniera simile tranne che l'ordine di invocazione è invertito;
- Nella definizione di un costruttore della classe derivata si può richiamare esplicitamente un costruttore della classe base (lista di inizializzazione);
- Se una funzione propria è ridefinita in una classe derivata allora maschera la funzione originale indipendentemente dal numero e il tipo degli argomenti.

Costruttori e distruttori per classi derivate

```
class Persona{
protected:
   char Nome[10];
   int Anni;
public:
   Persona(char* N,int A) { strcpy(Nome,N); Anni = A; }
};
class Bambino : public Persona {
private:
   int Mesi;
public: // costruttore con invocazione esplicita del costruttore classe base
   Bambino(char* N,int A,int M): Persona(N,A) { Mesi = M; }
};
main() {
   Bambino b("Nico",9,11); Persona p("Mario",20);
}
```

Derivazione di classi e overloading di funzioni

```
class Persona{
protected:
   char Nome[10];  int Anni;
public:
   Persona(char* N,int A) { strcpy(Nome,N); Anni = A; };
   void StampaMesi() { cout << Anni*12 << endl; }
   void AggiornaEta(int a) { Anni = a; }
};
class Bambino : public Persona {
private:
   int Mesi;
public:
   Bambino(char* N,int A,int M) : Persona(N,A) { Mesi = M;}
   void StampaMesi() { cout << Anni*12 + Mesi << endl; }
   void AggiornaEta(int a, int m){ Anni = a; Mesi = m; }
         // AggiornaEta maschera l'omologa funzione nella classe base
};
```

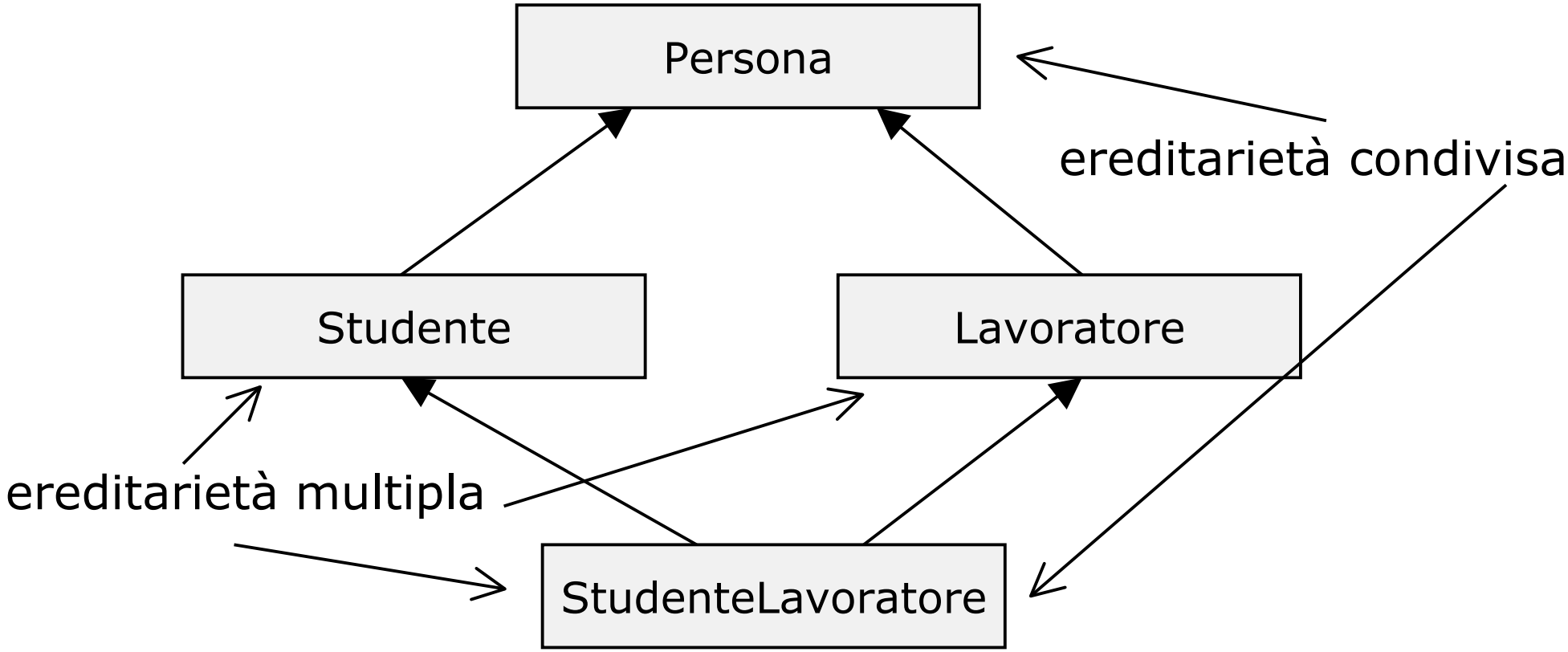
Invocazione di funzione per classi base e derivate

```
main(){
    Bambino b("Nico",9,11);
    b.AggiornaEta(10); // errore: la funzione con un argomento e' mascherata
    b.AggiornaEta(10,1);            // OK
    Persona p("Mario",20);
    p.StampaMesi();                 // 240
    b.StampaMesi();                 // 121
    // invocazione esplicita di funzione mascherata:
    b.Persona::StampaMesi();        // 120
}
```

Derivazione multipla e late binding

- Da una classe base è possibile derivare diverse classi
- Una classe si può ottenere per derivazione di più classi base (**ereditarietà multipla**)
- Si possono costruire complesse gerarchie di derivazione
- E' possibile che una classe sia derivata in diverse maniere dalla stessa classe base (**ereditarietà condivisa**)
- Nella derivazione condivisa bisogna prestare attenzione a casi di ambiguità che può essere risolta con la **derivazione virtuale**

Gerarchia di derivazione



Più derivazioni dalla stessa classe base

```
class Persona {
protected:
        char  nome[20];
        int  eta;
...
};
class Studente : public Persona { // ha anche nome ed eta
protected:
        int  matricola;
....
};
class Lavoratore : public Persona { // ha anche nome ed eta
protected:
        int  stipendio;
...
};
```

Ereditarietà multipla (e condivisa)

```
// Derivazione condivisa: la classe StudenteLavoratore eredita
// due volte il nome e l'eta'
class StudenteLavoratore : public Studente, public Lavoratore {
private:
        char occupazione[10];
...
public:
        void mostra_dati() {
                cout    << Studente::nome << " "
                        << Lavoratore::eta  << " "
                        << matricola << " "
                        << stipendio << " "
                        << occupazione;
};
```

Derivazione virtuale

```
class Persona {
protected:
        char  nome[20];
        int  eta;
...
};
class Studente : virtual public Persona {
protected:
        int  matricola;
....
};
class Lavoratore : virtual public Persona {
protected:
        int  stipendio;
...
};
```

Ereditarietà condivisa tramite derivazione virtuale

```
// Ora la classe StudenteLavoratore eredita una sola volta il nome e l'eta'
class StudenteLavoratore : public Studente, public Lavoratore {
private:
        char occupazione[10];
...
public:
        void mostra_dati() {
                cout    << nome << " "
                        << eta  << " "
                        << matricola << " "
                        << stipendio << " "
                        << occupazione;
};
```

Late binding

- Nell'**early (static) binding** l'associazione tra invocazione di una funzione e sua definizione è fissa ed è stabilita a priori — questo è il meccanismo standard nella programmazione tradizionale
- Nel **late binding** l'associazione tra invocazione di una funzione e sua definizione può variare ed è rimandata — questo è un meccanismo introdotto con la derivazione (programmazione OO)
- Le **funzioni virtuali** producono late binding perché hanno la particolarità di invocare il codice della funzione relativo:
  - all'oggetto di invocazione e non alla classe nelle quali sono definite, oppure
  - al parametro attuale e non a quello formale

Esempio

```
#include <iostream.h>
class Persona{
protected:
    char Nome[10];
    int Anni;
public:
    Persona(char* N,int A) { strcpy(Nome,N); Anni = A; };
    void StampaAnni() { cout << Anni << " anni" << endl; }
    void StampaDati() { cout << Nome << ':'; StampaAnni(); }
};
```

Static binding

```
class Bambino : public Persona {
private:
    int Mesi;
public:
    Bambino(char* N,int A,int M) : Persona(N,A) { Mesi = M; }
    void StampaAnni() { cout << Anni*12 + Mesi << " mesi" << endl; }
};
main(){
    Persona p("Mario",20);
    Bambino b("Nico",2,4);
    p.StampaDati();    // Mario: 20 anni
    b.StampaDati();    // Nico: 2 anni !! (static binding)
}
```

Definizione di funzione virtual

```
#include <iostream.h>
class Persona{
protected:
    char Nome[10];
    int Anni;
public:
    Persona(char* N,int A) { strcpy(Nome,N); Anni = A; };
    virtual void StampaAnni() { cout << Anni << " anni" << endl; }
    void StampaDati() { cout << Nome << ':'; StampaAnni(); }
};
```

Late binding

```
class Bambino : public Persona {
private:
    int Mesi;
public:
    Bambino(char* N,int A,int M) : Persona(N,A) { Mesi = M; }
    void StampaAnni() { cout << Anni*12 + Mesi << " mesi" << endl; }
};
main(){
    Persona p("Mario",20);
    Bambino b("Nico",2,4);
    p.StampaDati();    // Mario: 20 anni
    b.StampaDati();    // Nico: 28 mesi (late binding)
}
```

Altro approccio per il late binding

```
#include <iostream.h>
class Persona{
protected:
    char Nome[10];
    int Anni;
public:
    Persona(char* N,int A) { strcpy(Nome,N); Anni = A; };
    virtual void StampaDati() {
            cout    << Nome << ':'
                    << Anni << " anni" << endl; }
};
```

L'argomento per riferimento consente il late bind.

```
class Bambino : public Persona {
private:
   int Mesi;
public:
   Bambino(char* N,int A,int M) : Persona(N,A) { Mesi = M;}
   void StampaDati() {
         cout << Nome << ':' << Anni*12 + Mesi << " mesi" << endl;  }
};
void Informazioni1(Persona p){ p.StampaDati(); }
void Informazioni2(Persona& p){ p.StampaDati(); }
main(){ Persona p("Mario",20); Bambino b("Nico",2,4);
   Informazioni1(b);    // Nico: 2 anni (static binding)
   Informazioni2(b);    // Nico: 28 mesi (late binding)
}
```

Esempio completo di derivazione

```
class Persona {
public:
        Persona() { strcpy(nome,"ignoto"); }
        Persona(char*, data, char*);
        char* Nome() { return nome; }
        data& DataNascita() { return data_nasc; }
        char* CittaNascita() { return citta_nasc; }
        char* Occupazione() { return occupaz; }
        virtual void mostra_dati();
protected:
        char occupazione[20]; // campo visible solo dalle classi derivate
private:
        char  nome[20];
        data  data_nasc; // bisogna includere data.h
        char  citta_nasc[20];
};
```

Definizione funzioni Persona e funzione esterne

```
Persona::Persona(char* n, data d, char* c) {
        strcpy(nome,n);
        data_nasc = d;
        strcpy(citta_nasc,c);
        strcpy(occupaz,"nessuna");
};
void Persona::mostra_dati() {
        cout    << nome << " e' nato a “ << citta_nasc
                << " il “ << data_nasc << endl;
};
// funzioni esterne Eta e Info
int Eta(Persona p, data d) { // puo' essere invocata anche su classi derivate
        return d.Anno() - p.DataNascita().Anno();
};
// questa funzione provocherà late binding
void Info(Persona& p) { p.mostra_dati(); };
```

Classe studente derivato da Persona

```
class Studente : public Persona {
public:
        Studente(char*, data, char*, char*, int);
        char*  Scuola() { return scuola; }
        int Matricola() { return matricola; }
        void mostra_dati(data d);// ridefinisce la funzione omologa di
                                        // Persona anche se ha header diverso
private:
        char  scuola[20];
        int  matricola;
};
```

Definizione funzioni Studente

```
Studente::Studente(char* n, data d, char* c, char* s, int m): Persona(n,d,c) {
        strcpy(scuola,s);
        matricola = m;
        strcpy(occupaz,"studente");
};

void Studente::mostra_dati(data d) {
                cout    << Nome() << " e' nato a "
                        << CittaNascita() << " ha "
                        << Eta(*this,d)
                        << "anni e studia a "
                        << scuola << endl;
};
```

Classe Lavoratore

```
class Lavoratore : public Persona {
public:
        Lavoratore(char*, data, char*, char*, int);
        char*  Datore() { return datore; }
        int Stipendio() { return stipendio; }
        void mostra_dati(); // ridefinisce la funzione omologa di Persona
private:
        char datore[20];
        int  stipendio;
};
```

Definizione funzioni Lavoratore

```
Lavoratore::Lavoratore(char* n, data d, char* c, char* dat, int s):
                    Persona(n,d,c) {
        strcpy(datore,dat);
        stipendio = s;
        strcpy(occupazione,"lavoratore");
};

void Lavoratore::mostra_dati() { // ridefinisce la funzione omologa di Persona
        cout    << Nome() << " e' nato a " << CittaNascita()
                << " il " << DataNascita()
                << " e lavora per " << datore << endl;
};
```

Classe StudenteLavoratore

```
class StudenteLavoratore : public Studente, public Lavoratore {
public:
        StudenteLavoratore(char*, data, char*, char*, int, char*, int);
        void mostra_dati(); // necessaria per disambiguare
private:
        char occupazione[10]; // campo necessario per disambiguare
};
StudenteLavoratore::StudenteLavoratore(char* n, data d, char* c, char* sc,
        int m, char* dat, int st): Studente(n,d,c,sc,m), Lavoratore (n,d,c,dat,st) {
                strcpy(occupaz,"lavoratore e studente");
};
// funzione necessaria per disambiguare
void StudenteLavoratore::mostra_dati() {  Lavoratore::mostra_dati();
        cout << " e studia a "<< Scuola() << endl;
};
```

Programma main che usa le varie classi

```
void main() {
        data oggi(18,3,1997);
        Persona p1("mario rossi",data(11,11,1975),"Roma");
        Persona p2("mario rossi",data(11,11,1975),"Roma");
        Studente s1("franco bini",data(10,8,1970),"Pisa","Roma Tre", 333333);
        Lavoratore l1("anna vanni",data(10,8,1968),"Milano","Upim", 40);
        StudenteLavoratore sl1("maria lotti",data(10,8,1974),"Roma",
                                        "Roma Tre",333333,"McDonalds",20);
        cout << l1.Nome() << " ha " << Eta(l1, oggi) << " anni" << endl;
        l1.mostra_dati();
        s1.mostra_dati(oggi);
        p1.mostra_dati();
        sl1.mostra_dati();
        Info(l1);
}
```

Classi base astratte

- Classi utilizzate solo per la derivazione: non hanno oggetti propri, ma sono usate per definire altre classi attraverso la derivazione
- Contengono funzioni virtuali pure, cioè funzioni virtuali senza corpo (convenzionalmente al posto del corpo c'è "=0"), che sono invece definite nelle classi derivate
- Sono utilizzate per descrivere tipi astratti, rimandando la rappresentazione (o le rappresentazioni) alle classi derivate
- Si sfrutta il meccanismo del late binding

Esempio di classe astratta

```
#include <iostream.h>


class stack {
public:
    virtual bool empty () = 0;
    virtual void push (int el) = 0;
    virtual int pop () = 0;
};
```

Implementazione dello stack con array

```
class stackWithArray : public stack {
    int* v;
    int* p;
    int maxsize;
public:
    stackWithArray(int s) { v = p = new int[maxsize=s]; }
    ~stackWithArray() { delete[] v; }
    bool empty () { return (v==p); }
    void push (int el) { *p = el; p++; }
    int pop() { p--; return *p; }
};
```

Implementazione dello stack con strutt. collegate

```
class stackWithList : public stack {
    struct elem { elem* next; int info; };
    elem* p;
    void destroy(elem* p) { elem* tmp; while(p) { tmp = p; p = p->next;
                                                    delete tmp; } }
public:
    stackWithList(int s){ p = NULL; } ~stackWithList() { destroy (p); }
    bool empty () { return (p==NULL); }
    void push (int el) { elem* paux = new elem; paux ->info = el;
                                        paux ->next = p; p = paux; }
    int pop () { if (p != NULL) { int val = p->info; elem* paux  = p; p = p->next;
                                        delete paux; return val; }
    }
};
```

Funzione che provoca late binding

```
void Stampa(stack& s) {
    if (!s.empty()) {
        int a = s.pop();
        cout << a << ' ';
        Stampa(s);
        s.push(a);
    }
    else cout << endl;
}
```

Uso delle classi derivate e della funzione Stampa

```
main(){
    stackWithArray sA(100);
    sA.push (3); sA.push (2); sA.push (1);
    Stampa(sA); // 1 2 3
    int c;
    c = sA.pop ();
    Stampa(sA); // 2 3
    stackWithList sL(100);
    sL.push (3); sL.push (2); sL.push (1);
    Stampa(sL); // 1 2 3
    c = sL.pop ();
    Stampa(sL); // 2 3
}
```

Derivazione e template (diversi casi possibili)

- La classe base è un template e quella derivata istanzia il parametro (e quindi non è un template):

```
template <class T>
class Base { ... };
class Derivata : public Base<char> { .. };
```

- La classe derivata è un template e quella base no:

```
class Base { ... };
template <class T>
class Derivata : public Base { .. };
```

- Sia la classe base che la classe derivata sono dei template rispetto allo stesso tipo parametrico:

```
template <class T>
class Base { ... };
template <class T>
class Derivata : public Base<T> { .. };
```

Classe base virtuale parametrica e funz. esterna

```
#include <iostream.h>
template <class T>
class stack {
public:
   virtual bool empty () = 0;
   virtual void push (T el) = 0;
   virtual T pop () = 0;
};

template <class T> void Stampa(stack<T>& s) {
   if (!s.empty()) {
      T a = s.pop(); cout << a << ' '; Stampa(s); s.push(a);
   }
   else cout << endl;
};
```

Classe derivata parametrica

```
template <class T>
class stackWithArray : public stack<T> {
   T* v;
   T* p;
   int maxsize;
public:
   stackWithArray(int s) { v = p = new T[maxsize=s]; }
   ~stackWithArray() { delete[] v; }
   bool empty () { return (v==p); }
   void push (T el) { *p = el; p++; }
   T pop() { p--; return *p; }
};
```

Altra classe derivata parametrica

```
template <class T>
class stackWithList : public stack<T> {
   struct elem {
      elem* next; T info;
   };
   elem* p;
   void destroy(elem* p) { elem* tmp;
            while(p) { tmp = p; p = p->next; delete tmp; }
   }
public:
   stackWithList(int s){ p = NULL; }
   ~stackWithList() { destroy (p); }
   bool empty () { return (p==NULL); }
   void push (T);
   T pop ();
};
```

Implementazione delle funzioni di stackWithList

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

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

Classe Numero Complesso

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

Uso delle varie classi

```
main(){
   stackWithList<char> sc(100); // stack di caratteri con liste
   sc.push ('Z'); sc.push ('Y'); sc.push ('X'); Stampa(sc); // X Y Z
   cout << "tolgo" << ' ' << sc.pop() << endl; // tolgo X
   Stampa(sc); // Y Z
   stackWithArray<int> si(100); // stack di interi con array
   si.push (3); si.push (2); si.push (1); Stampa(si); // 1 2 3
   cout << "tolgo" << ' ' << si.pop() << endl; // tolgo 1
   Stampa(si); // 2 3
   stackWithList<NumeroComplesso> scx(100); // stack di complessi con liste
   scx.push (NumeroComplesso(3,3)); scx.push (NumeroComplesso(2,2));
   scx.push (NumeroComplesso(1,1));
   Stampa(scx); // (1,1) (2,2) (3,3)
   cout << "tolgo" << ' ' << scx.pop(); cout << endl; // tolgo (1,1)
   Stampa(scx); // (2,2) (3,3)
}
```