Classi e Oggetti nel linguaggio C++

prof. Riccardo Torlone
Università di Roma Tre

Una classe in C++

```
class data {
        public:
                int Giorno() { return giorno; }
                int Mese() { return mese; }
                int Anno() { return anno; }
                void AvanzaUnGiorno();
        private:
                int giorno;
                int mese;
                int anno;
                int GiorniDelMese();
};
```

Definizione della funzione AvanzaUnGiorno

```
void data::AvanzaUnGiorno() {
        if (giorno == GiorniDelMese())
                if (mese == 12) {
                        giorno = 1; mese = 1; anno++;
                }
                else {
                        giorno = 1;
                        mese++;
                }
        else giorno++;
}
```

Definizione della funzione GiorniDelMese

```
int data::GiorniDelMese() {
        bool AnnoBisestile=false;
        if (anno % 4 == 0) AnnoBisestile=true;
        if ((anno % 100 == 0) && (anno % 400 != 0))
                AnnoBisestile = false;
        switch (mese) {
                case 2: return (AnnoBisestile)?28:29;
                case 4: case 6: case 9: case 11: return 30;
                default: return 31;
        }
}
```

Alcune caratteristiche delle classi

- Le funzioni definite in una classe sono dette funzioni proprie o funzioni membro;
- Le funzioni proprie definite congiuntamente alla definizione della classe sono automaticamente inline;
- Una classe si usa come tutti gli altri tipi predefiniti e non, le variabili di una classe sono detti **oggetti** (detti anche *instanze* della classe);
- Come le variabili tradizionali, gli oggetti possono essere creati:
  - in maniera statica

    data oggi; //definisce (e genera) un oggetto di tipo data
  - in maniera dinamica tramite un puntatore

    data *pd; //definisce un puntatore ad un oggetto di tipo data

    pd = new data; //genera un oggetto ti tipo data puntato da pd

Information hiding

- La parte della classe individuata dalla parola chiave **public** è detta parte pubblica, quella individuata dalla parola chiave **private** (che è opzionale) è detta parte privata della classe;
- Si può accedere dall'esterno, tramite oggetti della classe, solo alla parte pubblica; per accedere ai campi pubblici di una classe (dati e funzioni) si usa la notazione punto;
- Per esempio:

```
cout  <<  oggi.Giorno()  << "/"
      <<  oggi.Mese()  << "/"
      <<  oggi.Anno();
```

- In questo caso oggi è detto **oggetto di invocazione** delle funzioni proprie

Side-effect su oggetti di una classe

- Una funzione di una classe produce un side-effect quando modifica l'oggetto di invocazione della funzione

```
// funzione con side effect
void data::AvanzaUnGiorno() {
  if (giorno == GiorniDelMese())
   if (mese == 12) { giorno = 1; mese = 1; anno++; }
   else { giorno = 1; mese++; }
  else giorno++;
}
```

Esempio di funzione senza side effect

```
data data::IlGiornoDopo() {
  data d;
  if (giorno == GiorniDelMese())
      if (mese == 12) {
              d.giorno = 1;
              d.mese = 1;
              d.anno = anno+1;
      }
      else { d.giorno = 1; d.mese = mese+1; }
  else d.giorno = giorno+1;
  return d;
}
```

Puntatore “this” in C++

- All’interno della definizione di una funzione membro di una classe si può usare, in qualunque momento il puntatore this
- Tale puntatore viene fornito automaticamente all’atto dell’invocazione della funzione e punta sempre all’oggetto di invocazione della funzione
- Per esempio:

```
int Giorno() { return giorno; }
```

  si può anche scrivere:

```
int Giorno() { return this->giorno; }
```

- Vedremo nel seguito come, in alcuni casi, può risultare utile questo puntatore

Differenza tra classi e record (struct) in C++

- Il costrutto struct si comporta in C++ in maniera del tutto simile al costrutto class con un'unica, importante, differenza:
  - per default i campi di una classe (dati o funzioni) sono privati;
  - per default i campi di un record (dati o funzioni) sono pubblici
- In genere si usa il tipo struct quando si vuole implementare un dato strutturato semplice non avente funzioni proprie (come il record in C o in Pascal)
- Si usa invece la class quando si vuole implementare un tipo astratto di dato (avente dati e funzioni)

Costruttori delle classi

- Il costruttore di una classe è una funzione pubblica speciale che serve ad inizializzare un oggetto e, in genere, si invoca all'atto della sua creazione

```
class data {
  public:
   data(); // costruttore di default
   data(int); // costruttore unario
   data(int, int, int);// costruttore con tre argomenti
   int Giorno() { return giorno; }
   int Mese() { return mese; }
   int Anno() { return anno; }
   void AvanzaUnGiorno();
  private:
   int giorno; int mese; int anno; int GiorniDelMese();
};
```

Implementazione dei costruttori della classe

```
data::data() {
        giorno = 1; mese = 1; anno = 2002;
}

data::data(int g) { // g denota l'iesimo giorno dell'anno
        data d;
        for (int i=1;i<g;i++) d.AvanzaUnGiorno();
        giorno = d.Giorno(); mese = d.Mese();
        anno = 2002;
}

data::data(int g, int me, int a) {
        giorno = g; mese = me; anno = a;
}
```

Alcune osservazioni sui costruttori

- Il costruttore ha lo stesso nome della classe e non restituisce nessun valore
- Per la proprietà dell'overloading del C++ si possono definire più costruttori per un classe
- Esiste un costruttore standard predefinito senza argomenti per ogni classe che lascia indefinite tutte le variabili

Invocazione del costruttore

- I costruttori si possono invocare in diverse maniere

```
data una_data; // invocazione del costruttore di default
data oggi(13,3,2002);
            // invocazione implicita del costruttore
            // con tre argomenti in una dichiarazione
data giornata = data(13,5,1998);
            // invocazione esplicita del costruttore
            // con tre argomenti in una dichiarazione
data d1;
d1 = data(11,12,2000);
            // invocazione esplicita del costruttore
            // con tre argomenti in un assegnazione
```

Altre invocazioni del costruttore

```
data *pd;
pd = new data;
  // invocazione implicita del costruttore
  // di default in una allocazione dinamica
data d2 = data(212);
  // invocazione esplicita del costruttore
  // unario (agisce da conversione di tipo)
data d3 = 78;
  // invocazione implicita del costruttore
  // unario (agisce da conversione di tipo)
data d4 = d3; // invocazione del costruttore di copia
data d5(d3);  // invocazione del costruttore di copia
```

Costruttore di copia

Il costruttore di copia è un costruttore predefinito per una classe (per esempio la classe data) che viene invocato:

- Nelle definizioni con inizializzazione aventi la seguente forma:

```
data d1 = d2;
data d3(d1); // invoca il costruttore di copia
```

- Nel passaggio di parametri a una funzione per valore

```
int trasforma(data d) { ... };
data oggi;
int x;
x = trasforma(oggi);  // invoca il costruttore di copia
```

- Nella restituzione di un valore da parte di una funzione

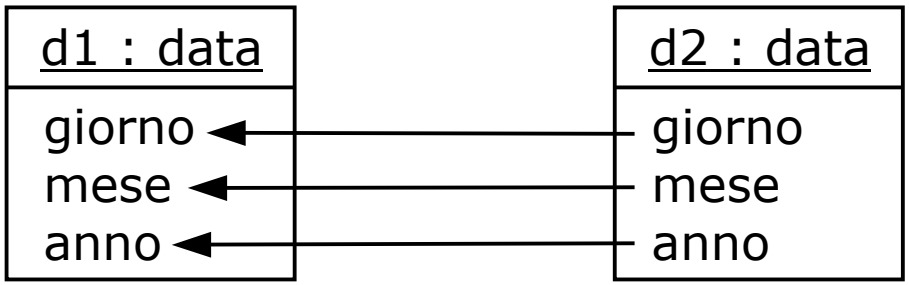
```
data trasforma(int i); {
        data d;
        ....;
        return d;  // invoca il costruttore di copia
}
```

Assegnazione

- Anche l'assegnazione (standard) è predefinita per ogni classe

```
data d1,d2;
d1 = d2; // assegnazione tra oggetti
```

- Sia il costruttore di copia che l'assegnazione standard effettuano una assegnazione dei campi rispettivi degli oggetti coinvolti



Campi static di una classe

I campi static di una classe sono associati alla classe e sono quindi condivisi da tutti gli oggetti della classe.

```
class data {
  public:
      data(int, int, int);
      int Giorno() { return giorno; }
      int Mese() { return mese; }
      int Anno() { return anno; }
      static data QuestoNatale();
      static int noggetti;
  private:
      int giorno; int mese; int anno;
};
```

Definizione e uso dei campi static

```
data::data(int g, int me, int a) {
   giorno = g; mese = me; anno = a;
   noggetti++;
}
data data::QuestoNatale() { return data(25,12,2002); }
int data::noggetti=0;
void main() {
   data d1(3,7,2001),d2(1,3,2002);
   cout  << data::QuestoNatale().Giorno() << "/"
                  << d1.QuestoNatale().Mese() << "/"
                  << d2.QuestoNatale().Anno() << endl;
   cout  << data::noggetti;  // stampa 2
   cout  << d1.noggetti;          // e' equivalente alla istr. precedente
   cout  << d2.noggetti;          // e' equivalente alla istr. precedente
}
```

Overloading di operatori

- E’ possibile ridefinire gli operatori del C++. In questa maniera si possono usare funzioni in forma infissa

```
class data {
  public:
   data(int, int, int);
   int Giorno() { return giorno; }
            int Mese() { return mese; }
            int Anno() { return anno; }
   void AvanzaUnGiorno();
   boolean operator<(data);
   int operator—(data);
  private:
            int giorno; int mese; int anno;
};
```

Definizione degli operatori sulle date

```
boolean data::operator<(data d) {
        return anno < d.anno || (anno == d.anno && mese < d.mese) ||
        (anno == d.anno && mese == d.mese && giorno < d.giorno) ;
}
int data::operator—(data d) {
        data temp = *this; int i = 0;
        while (d<temp) { d.AvanzaUnGiorno(); i++; }
        return i;
}
void main() {
        data oggi(13,3,2002), d(11,12,2005);
        if (oggi<d) cout << "d e' un giorno futuro";
        cout << d — oggi; // equivale a d.operator—(oggi);
}
```

Overloading di operatori con funzioni esterne

- A volte risulta conveniente ridefinire degli operatori per una classe con delle funzioni esterne

```
/* overloading dell'operatore << come funzione propria */
ostream& data::operator<<(ostream& s) {
  return s << giorno << '/' << mese << '/' << anno;
}
void main() {
  data oggi(13,3,2002);
  // se ridefinisco << come funzione propria
  // l'uso e' controintuitivo:
  oggi << cout; // corrisponde a:  oggi.operator<<(cout)
}
```

Ridefinizione di << come funzione esterna

```
/* overloading dell'operatore << come funzione esterna */
ostream& operator<<(ostream& s, data& d) {
        return s << d.Giorno() << '/' << d.Mese() << '/' << d.Anno();
}

void main() {
  data oggi(5,5,2002), d(11,12,2004);
  cout << “oggi e’ il “ << oggi << “ e un giorno sara’ il “ << d;
  /* (cout << oggi) corrisponde a: operator<<(cout,oggi) */
}
```

Funzioni friend

- Sono funzioni esterne a una classe ma che possono accedere ai campi privati della classe. E' la classe che deve dichiararli friend.

```
class data { // dichiarazione di funzioni friend
  friend ostream& operator<<(ostream&, data&);
  public:
   ...
}
/* overloading dell'operatore << come funzione friend
    esterna: l'accesso ai dati e' semplificato */
ostream& operator<<(ostream& s, data& d) {
  return s  << d.giorno << '/' << d.mese << '/' << d.anno;
}
void main() {
  data oggi(13,3,2002); cout << oggi;
}
```

Inizializzazione di campi di una classe

- L'inizializzazione dei campi di una classe può essere fatta utilizzando la seguente sintassi:

```
// Definizione del costruttore con tre argomenti della classe data:
data::data(int g, int me, int a): giorno(g), mese(me), anno(a) {}
// Equivale alla seguente  definizione:
data::data(int g, int me, int a) {
  giorno = g; mese = me; anno = a;
}
```

- In questo caso viene invocato il costruttore di copia del tipo corrispondente

Ridefinizione dell'assegnazione e del costruttore di copia

- Vanno ridefiniti quando ci sono dei fenomeni di side-effect che producono comportamenti non desiderati
- Bisogna ricordare che i prototipi dell'assegnazione e del costruttore di copia di una classe C sono, rispettivamente:

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

Esempio: classe lista

```
typedef int tipoelem;


class lista {
public:
        lista() { first=NULL; } :
        lista(const lista& l) { first=copia(l.first); } // ridefinizione costr. di copia
        lista& operator=(const lista& ll)  { // ridefinizione dell'assegnazione
                elem *aux=first;
                first = copia(ll.first);
                rilascia(aux);
                return *this;
        }
```

Parte privata della classe lista

```
private:
        struct elem  {
                tipoelem inf;
                elem *next;
        };
        elem* first; // puntatore al primo elemento della lista

        void rilascia(elem*); // funzione di servizio che dealloca
                // tutta la lista in ingresso

        elem* copia(elem*); // funzione di servizio che restituisce
                // una copia della lista in ingresso
};
```

Definizione della funzione copia

```
lista::elem* lista::copia(elem* p) {
        if (p==NULL)  { return NULL; }
        elem *aux = new elem;  aux->inf = p->inf;  aux->next = NULL;
        elem *q = aux;
        p=p->next;
        while (p != NULL)    {
                q->next = new elem;
                q = q->next;
                q->inf = p->inf;
                q->next = NULL;
                p = p->next;
        }
        return aux;
};
```

Nell'uso si evita condivisione di memoria

```
void lista::rilascia(elem* p) {
        while (p != NULL)  {
                elem *tmp = p;
                p = p->next;
                delete tmp;
        }
};
void main() {
        lista l1;
        lista l2=l1; // viene invocato il costruttore di copia
        lista l3(l1); // viene invocato il costruttore di copia
        lista l4;
        l4 = l1; // viene invocata l'assegnazione
}
```

Distruttore di una classe

- E’ una funzione senza parametri di una classe che ha lo stesso nome della classe preceduto dal simbolo “**~**” e che viene eseguita:
  - al termine del ciclo di vita di un oggetto della classe;
  - nella esecuzione di una istruzione **delete**

- Esiste un distruttore standard per ogni classe che rilascia la memoria degli oggetti allocata staticamente
- Nel caso di allocazioni dinamiche il distruttore va opportunamente ridefinito

Esempio di distruttore

```
typedef int tipoelem;
class lista {
public:
  lista() { first=NULL; }
  ~lista()  { rilascia(first); }
  ...
private:
  struct elem  {
            tipoelem inf;
            elem *next;
  };
  elem* first;
  void rilascia(elem*);
  elem* copia(elem*);
};
```

Invocazione implicita ed esplicita di distruttori

```
void main() {
  lista l1
  lista l2;
  ...
  delete l1;          // viene invocato il distruttore per l1
  ...
} // viene invocato il distruttore per l2
```

Esempio: gestione di stringhe

```
#include <iostream.h>
#include <string.h>
class Stringa {
  friend ostream& operator<<(ostream&,  Stringa&);
  public:
      Stringa(int l = 32);  Stringa(const char*);
      ~Stringa();
      Stringa(const Stringa&); Stringa& operator =(const Stringa&);
      Stringa operator +(const Stringa&); Stringa operator +(const char*);
      char& operator [ ] (int);
      Stringa& operator +=(const Stringa&);
      Stringa& operator += (const char*);
  private:
      int len;   char* str;
};
```

```
Stringa::Stringa(int l) {
   len = l;
   str= new char[l+1];
   str[0]='\0';
}

Stringa::Stringa(const char* p) {
   len = strlen(p);
   str= new char[len];
   strcpy(str,p);
}

Stringa::~Stringa() {
   delete [] str;
}
```

Overloading del costr. di copia e dell'assegnazione

```
Stringa::Stringa(const Stringa & s) {
        len = s.len;
        str= new char[len+1];
        strcpy(str,s.str);
}

Stringa& Stringa::operator=(const Stringa& s) {
        len = s.len;
        delete str;
        str = new char[len+1];
        strcpy(str,s.str);
        return *this;
}
```

Overloading dell'operatore + (due versioni)

```
Stringa Stringa::operator+(const Stringa& s){
   len += s.len;
   char* p = new char[len+1];
   strcpy(p,str); strcat(p,s.str);
   delete str;
   str = p;
   return *this;
}
Stringa Stringa::operator+(const char* s) {
   len += strlen(s);
   char* p = new char[len+1];
   strcpy(p,str); strcat(p,s);
   delete str;
   str = p;
   return *this;
}
```

Overloading degli operatori [ ] e +=

```
char& Stringa::operator [ ] (int elem) {
   if (elem > 0 && elem < strlen(str))
         return str[elem];
}

Stringa& Stringa::operator+=(const Stringa& s) {
   len += s.len;
   char* p = new char[len+1];
   strcpy(p,str);
   strcat(p,s.str);
   delete str;
   str = p;
   return *this;
}
```

Altro overloading dell'operatore += di <<

```
Stringa& Stringa::operator+=(const char* s) {
   len += strlen(s);
   char* p = new char[len+1];
   strcpy(p,str);
   strcat(p,s);
   delete str;
   str = p;
   return *this;
}

// overloading dell'operatore <<
ostream& operator<<(ostream& s,Stringa& st) {
   return s << st.str << " (stringa di " << st.len << " caratteri)" << endl;
}
```