11.1 Tipi funzione

Dichiarazione di una funzione di *n* argomenti:

- **associa ad un identificatore un tipo, determinato dalla n-upla ordinata dei tipi degli argomenti e dal tipo del risultato.**

Valori associati ai tipi funzione:

- **tutte le funzioni corrispondenti (non valgono le conversioni implicite).**

```
#include <iostream>
using namespace std;

int quadrato(int n)          // Istanza di tipo funzione int(int)
{  return n*n;  }

int cubo(int n)              // Istanza di tipo funzione int(int)
{  return n*n*n;  }

int calcola(int fp(int), int a)   // funzione arg.
{  int s = 0;
   s = fp(a);
    return s;
}

int main()
{
    cout << calcola(quadrato, 5) << endl;
    cout << calcola(cubo, 5) << endl;
    return 0;
}
```

```
25
125
```

11.3 Argomenti default

Argomenti formali di una funzione:

- **possono avere inizializzatori;**
- **costituiscono il valore default degli argomenti attuali (vengono inseriti dal compilatore nelle chiamate della funzione in cui gli argomenti attuali sono omessi);**
- **se il valore default viene indicato solo per alcuni argomenti, questi devono essere gli ultimi;**
- **gli inizializzatori non possono contenere né variabili locali né argomenti formali della funzione.**

Chiamata di funzione:

- **possono essere omessi tutti o solo alcuni argomenti default: in ogni caso gli argomenti omessi devono essere gli ultimi.**

Esempio (peso di un cilindro):

```
double peso(double lung, double diam = 10, double
dens = 15)
{
    diam /= 2;
    return (diam*diam * 3.14 * lung * dens);
}
int main()
{  // ...
    p = peso(125);      // equivale a peso(125, 10, 15)
    p = peso(35, 5);    // equivale a peso(35, 5, 15)
 // ...
}
```

11.4 Overloading (I)

Più funzioni possono avere lo stesso nome.

Funzioni che hanno lo stesso nome devono differire almeno per

- **numero di argomenti**

oppure

- **tipo di almeno un elemento**

Il compilatore è sempre in grado di risolvere la chiamata alla funzione.

Se le funzioni differiscono solo per il tipo di ritorno, viene segnalato un errore dal compilatore.

```
int massimo(int a, int b)
{
        cout << "Massimo per interi " << endl;
        return a > b ? a : b;
}

double massimo(double a, double b)
{
        cout << "Massimo per double" << endl;
        return a > b ? a : b;
}

/* int massimo(double a, double b)          ERRORE!
{       return int(a > b ? a : b);
}  */
```

11.4 Overloading (II)

```
#include <cstdlib>
#include <iostream>
using namespace std;

int massimo(int a, int b)
{
    cout << "Massimo per interi " << endl;
    return a > b ? a : b;
}

double massimo(double a, double b)
{
    cout << "Massimo per double" << endl;
    return a > b ? a : b;
}

int main ()
{
    cout << massimo(10, 15) << endl;
    cout << massimo(12.3, 13.5) << endl;

//  cout << massimo(12.3, 13) << endl;
        // ERRORE: ambiguo

    cout << massimo('a','r') << endl;

    return 0;
}
```

```
Massimo per interi 15
Massimo per double 13.5
Massimo per interi 114
```

14.2.1 Visibilità

Programmi semplici:

- **formati da poche funzioni, tutte contenute in un unico file, che si scambiano informazioni attraverso argomenti e risultati.**

Programmi più complessi:

- **si utilizzano tecniche di programmazione modulare:**
  - **suddivisione di un programma in diverse parti che vengono scritte, compilate (verificate e modificate) separatamente;**
  - **scambio di informazioni fra funzioni utilizzando oggetti comuni.**

Visibilità (*scope*):

- **campo di visibilità di un identificatore (parte di programma in cui l'identificatore può essere usato);**

Regole che definiscono la visibilità degli identificatori (regole di visibilità):

- **servono a controllare la condivisione delle informazioni fra i vari componenti di un programma:**
  - **permettono a più parti del programma di riferirsi ad una stessa entità (il nome dell'entità deve essere visibile alle parti del programma interessate);**
  - **impediscono ad alcune parti di un programma di riferirsi ad una entità (il nome dell'entità non deve essere visibile a tali parti).**

14.3 Blocchi

```
// Sequenza di istruzioni racchiuse tra parentesi graffe
#include <cstdlib>
#include <iostream>
using namespace std;

void f()
{
    int i = 2;                          // visibilita' locale
    cout << i << endl;                  // 2
}

int main()
{
//  cout << i << endl;                  ERRORE!
    int i = 1, j = 5;
    cout << i << '\t' << j << endl;     // 1 5

  {                                     // blocco
      cout << i << '\t' << j << endl;   // 1 5
      int i = 8;      // nasconde l'oggetto i del blocco super.
      cout << i << endl;                // 8
    }

    cout << i << endl;                  // 1

    f();                                // 2

    cout << i << endl;                  // 1
    return 0;
}
```

14.3 Visibilità livello di file

```
#include <cstdlib>
#include <iostream>
using namespace std;
int i;                    // visibilita' a livello di file

void leggi()              // visibilita' a livello di file
{
    cout << "Inserisci un numero intero " << endl;
    cin >> i;
}

void scrivi()             // visibilita' a livello di file
{
    cout << i << endl;
}

int main()
{
    leggi();
    scrivi();
    system("PAUSE");
    return 0;
}
```

```
Inserisci un numero intero
2
2
Premere un tasto per continuare . . .
```

- **Identificatori di oggetti con visibilità a livello di file individuano oggetti condivisi da tutte le funzioni definite nel file.**

14.4 Unità di compilazione (I)

Unità di compilazione:

- **costituita da un file sorgente e dai file inclusi mediante direttive #include;**
- **se il file da includere non è di libreria, il suo nome va racchiuso tra virgolette (e non fra parentesi angolari).**

Esempio:

```
// file header.h
int f1(int);  int f2(int);


// file main.cpp
#include "header.h"
int main()
{
    f1(3);
    f2(5);
    return 0;
}

// Unità di compilazione risultante
int f1(int);  int f2(int);
int main()
{
    f1(3);
    f2(5);
    return 0;
}
```

14.4 Unità di compilazione (III)

```
// Operatore :: unario (risoluzione di visibilita')

#include <cstdlib>
#include <iostream>
using namespace std;

int i = 1;                                  // visibilita' a livello di file

int main()
{
    cout << i << endl;                                  // 1
    {
        int i = 5;                                      // visibilita' locale
        cout << ::i << '\t' << i << endl;               // 1 5
        {
            int i = 10;                                 // visibilita' locale
            cout << ::i << '\t' << i << endl;           // 1 10
        }
    }
    cout << ::i << endl;                                // 1
    system("PAUSE");
    return 0;
}
```

```
1
1       5
1       10
1
Premere un tasto per continuare . . .
```

14.5 Spazio di nomi (I)

Spazio di nomi:

- **Insieme di dichiarazioni e definizioni racchiuse tra parentesi graffe, ognuna delle quali introduce determinate entità dette *membri*.**
- **Può essere dichiarato solo a livello di file o all'interno di un altro spazio dei nomi.**
- **Gli identificatori relativi ad uno spazio dei nomi sono visibili dal punto in cui sono dichiarati fino alla fine dello spazio dei nomi.**

```
namespace uno
{
    struct st {int a; double d; };
    int n;
    void ff(int a)
    {/*…*/}
    //…
}

namespace due
{
    struct st {int a; double d;};
}

int main()
{
    uno::st ss1;
    using namespace due;   //direttiva
    st ss2;
}
```

14.6 Collegamento (I)

Programma:

- **può essere formato da più unità di compilazione, che vengono sviluppate separatamente e successivamente collegate per formare un file eseguibile.**

Collegamento:

- **un identificatore ha *collegamento interno* se si riferisce a una entità accessibile solo da quella unità di compilazione;**
  - **uno stesso identificatore che ha collegamento interno in più unità di compilazione si riferisce in ognuna a una entità diversa;**

- **in una unità di compilazione, un identificatore ha *collegamento esterno* se si riferisce a una entità accessibile anche ad altre unità di compilazione;**
  - **tale entità deve essere unica in tutto il programma.**

Regola default:

- **gli identificatori con visibilità locale hanno collegamento interno;**
- **gli identificatori con visibilità a livello di file hanno collegamento esterno (a meno che non siano dichiarati con la parola chiave *const*).**

14.6 Collegamento (II)

Oggetti e funzioni con collegamento esterno:

- **possono essere utilizzati in altre unità di compilazione;**
- **in ciascuna unità in cui vengono utilizzati devono essere *dichiarati* (anche più volte).**

Oggetto:

- **viene solo dichiarato se si usa la parola chiave *extern* (e se non viene specificato nessun valore iniziale);**
- **viene anche definito se non viene usata la parola chiave *extern* (o se viene specificato un valore iniziale).**

Funzione:

- **viene solo dichiarata se si specifica solo l'intestazione (si può anche utilizzare la parola chiave *extern*, nel caso in cui la definizione si trovi in un altro file);**
- **viene anche definita se si specifica anche il corpo.**

Osservazione:

- **analogamente agli oggetti con visibilità a livello di file (oggetti condivisi), anche gli oggetti con collegamento esterno (oggetti *globali*) permettono la condivisione di informazioni fra funzioni.**

14.6 Collegamento (III)

```
// ------------------------ file file1.cpp----------------------- //

int a = 1;                          // collegamento esterno

const int N = 0;                    // const, collegamento interno

static int b = 10;                  // static, collegamento interno

// collegamento esterno
void f1(int a)
{                                   // a - collegamento interno
    int k;                          // k - collegamento interno
    /* ... */
}

// static, collegamento interno
static void f2()
{
    /* ... */
}

struct punto        // collegamento interno (dichiarazione)
{
    double x;
    double y;
};

punto p1;           // collegamento esterno
```

(continua ...)

14.6 Collegamento (IV)

```
// ---------------------------- file file2.cpp---------------------------- //
#include <cstdlib>
#include <iostream>
using namespace std;

extern int a;                          // solo dichiarazione
void f1(int);                          // solo dichiarazione
void f2();                             // solo dichiarazione
void f3();                             // solo dichiarazione
double f4(double, double);             // definizione mancante
                                       // OK, non utilizzata
int main()
{
    cout << a << endl;                 // OK, 1

    extern int b;                      // dichiarazione
//  cout << b << endl;                  ERRORE!

    f1(a);                             // OK
//  f2();                              ERRORE!
//  f3();                              ERRORE!
//  punto p2;                    ERRORE! punto non dichiarato

//  p1.x = 10;                    ERRORE! P1 non dichiarato
    system("PAUSE");
    return 0;
}
```

Stesso tipo in più unità di compilazione:

- **viene verificata solo l'uguaglianza tra gli identificatori del tipo;**
- **se l'organizzazione interna non è la stessa, si hanno errori logici a tempo di esecuzione.**

14.11 Moduli (I)

Modulo:
- **parte di programma che svolge una particolare funzionalità e che risiede su uno o più file;**
- **moduli *servitori* e moduli *clienti*.**

Modulo servitore:
- **offre (esporta) risorse di varia natura, come funzioni, variabili (globali) e tipi.**
- **costituito normalmente da due file (con estensione *h* e *cpp*, rispettivamente):**
  - **intestazione o interfaccia (dichiarazione dei servizi offerti);**
  - **realizzazione.**

Separazione fra interfaccia e realizzazione:
- **ha per scopo l'occultamento dell'informazione (*information hiding*);**
  - **semplifica le dipendenze fra i moduli;**
  - **permette di modificare la realizzazione di un modulo senza influenzare il funzionamento dei suoi clienti.**

Modulo cliente:
- **utilizza (importa) risorse offerte dai moduli servitori (include il file di intestazione di questi);**
- **viene scritto senza conoscere i dettagli relativi alla realizzazione dei moduli servitori.**

14.11 Moduli (II)

// ESEMPIO PILA
// MODULO SERVER

```
// file  pila.h
typedef  int T;
const int DIM = 5;
struct pila
{
   int top;
   T stack[DIM];
};

void inip(pila& pp);
bool empty(const pila& pp);
bool full(const pila& pp);
bool push(pila& pp, T s);
bool pop(pila& pp, T& s);
void stampa(const pila& pp);
```

14.11 Moduli (II)

```
// ESEMPIO PILA
// MODULO SERVER

// file pila.cpp

#include <cstdlib>
#include <iostream>
#include "pila.h"
using namespace std;

// inizializzazione della pila
void inip(pila& pp)
{
    pp.top = -1;
}
....
```

14.11 Moduli (III)

// ESEMPIO PILA
// MODULO CLIENT

```
// file  pilaMain.cpp
#include <cstdlib>
#include <iostream>
#include "pila.h"
using namespace std;

int main()
{
    pila st;
    inip(st);
    T num;
    if (empty(st)) cout << "Pila vuota" << endl;
    for (int i = 0; i < DIM; i++)
        if (push(st,DIM - i))
            cout << "Inserito " << DIM - i <<
                ". Valore di top: " << st.top << endl;
        else
            cerr << "Inserimento di " << i << " fallito" <<
    endl;
    if (full(st)) cout << "Pila piena" << endl;
    system("PAUSE");
    return 0;
}
```

14.11.1 Astrazioni procedurali

Astrazioni procedurali.

- **i moduli servitori mettono a disposizione dei moduli clienti un *insieme di funzioni*;**
- **le dichiarazioni di tali funzioni si trovano in un file di intestazione che viene incluso dai moduli clienti;**
- **la realizzazione di tali funzioni si trova in un file diverso (che non viene incluso).**
- **tali funzioni vengono usate senza che sia necessaria alcuna conoscenza della loro struttura interna.**

Esempio:

- **le funzioni di libreria per l'elaborazione delle stringhe sono contenute in un modulo il cui file di intestazione è <*cstring*>, e il loro utilizzo non richiede alcuna conoscenza sulla loro realizzazione.**

14.11.1 Tipi di dato astratto

Tipi di dato astratti:
•principio dell'occultamento dell'informazione:

- l'organizzazioine interna del tipo non deve essere accessibile ai moduli clienti;
- i moduli clienti possono definire oggetti di quel tipo, e accedervi soltanto attraverso le funzioni dichiarate nell'intestazione,
  senza possibilità di accedere alla loro organizzazione interna.

Le sole regole di visibilità e collegamento del linguaggio non consentono di avere moduli che realizzano compiutamente tipi di dato astratti.

```
#include "pila.h"
int main()
{
        pila st;
        inip(st);
        push(st, 1);
        cout << st.top << endl;// ATTENZIONE!
        st.top = 10;                // ATTENZIONE!
        // …
}
```

16.2 Tipi classe (I)

La struttura interna della pila è visibile ai moduli che utilizzano istanze della pila. In questo modo non si riesce a realizzare compiutamente il tipo di dato astratto.
Per ovviare a questo problema, il C++ mette a disposizione le classi.

```
#include <cstdlib>
#include <iostream>
using namespace std;

const int DIM = 5;
class pila
{     int top;
      int stack[DIM];
public:
       void inip();
       bool empty();
       bool full();
       bool push(int s);
       bool pop(int& s);
       void stampa();
};

int main()
{
      pila st;
      st.inip();
      st.push(1);
      st.top = 10;                        // ERRORE!
                  // 'int pila::top' is private within this context
```

16.2 Tipi classe (II)

basic-class-type-declaration
    class-type-specifier **;**
class-type-specifier
    class *identifier***|opt {** *class-element-seq* **}**
class-element
    *access-indicator***|opt** *class-member-section*
access-indicator
    access-specifier **:**
access-specifier
    private
    protected
    public

In genere utilizzeremo la forma semplificata seguente:

class *nome*
{ ***parte privata***
protected:
    parte protetta
public:
    parte pubblica
};

Un membro di una classe può essere:
- un tipo (enumerazione o struttura);
- un campo dati (oggetto non inizializzato);
- una funzione (dichiarazione o definizione);
- una classe (diversa da quella della classe a cui appartiene).

16.2 Tipi classe (III)

```
// Numeri Complessi (parte reale, parte immaginaria)
#include<cstdlib>
#include<iostream>
using namespace std;
class complesso
{
    double re, im;
public:
    complesso(double r=0, double i=0) {re = r; im = i;}
    double reale() {return re;}
    double immag() {return im;}
    /* … */
    void scrivi() {cout << '(' << re << ", " << im << ')';}
};

int main()
{
    complesso c1 (1.0, -1.0);
    complesso c2;
        //operat. di selezione
    c1.scrivi(); cout << endl;          // (1, -1)

    c2.scrivi(); cout << endl;          // (0, 0)

    complesso* cp = &c1;
    cp->scrivi(); cout << endl;         // (1, -1)
    system("PAUSE");
    return 0;
}
```

```
(1, -1)
(0, 0)
(1, -1)
Premere un tasto per continuare . . .
```