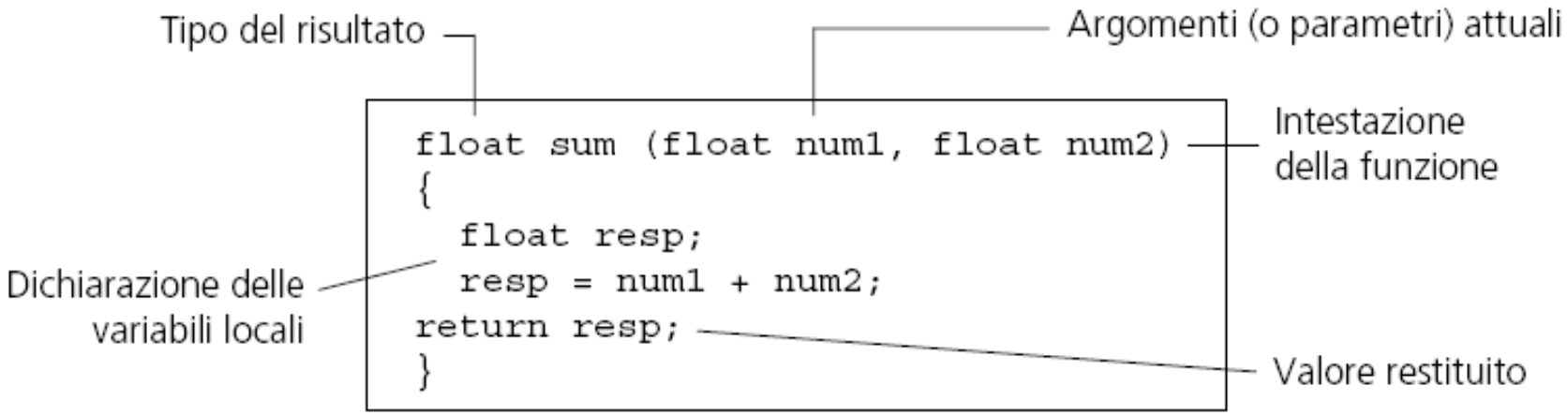
Funzioni

- unità di programma costituite da una intestazione, il tipo dell'eventuale risultato e gli eventuali parametri, seguita da un blocco che contiene dichiarazioni ed istruzioni
- facilitano la programmazione rendendola modulare
- possono essere raggruppate in librerie tematiche ed utilizzate in altri programmi



Funzioni

- mediante la parola riservata *return* si può ritornare il valore restituito dalla funzione al programma chiamante:

```
return(espressione);
return espressione;
```

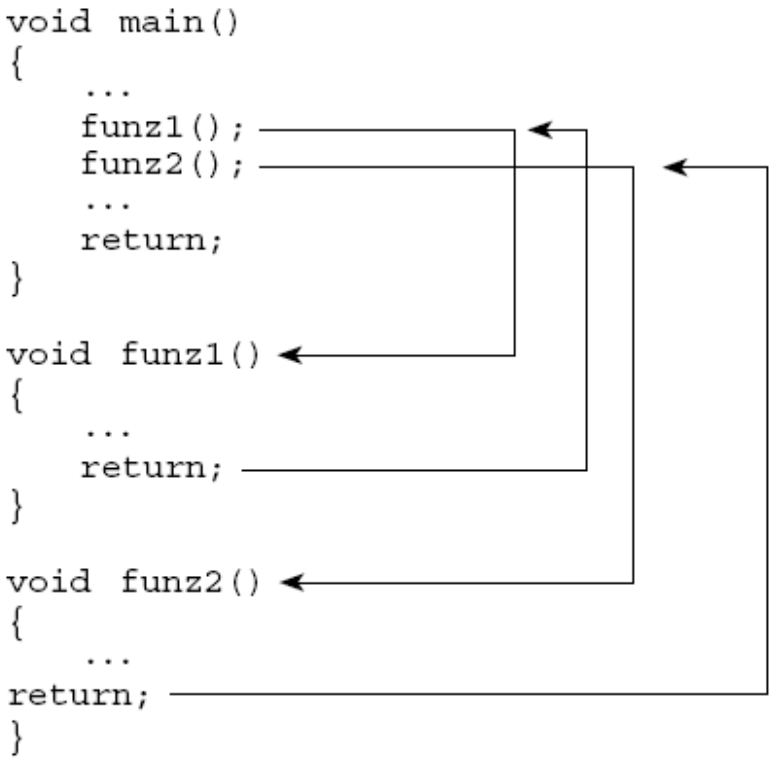
- ‘’espressione’’ deve essere del tipo definito come restituito dalla funzione;
- Se una funzione non restituisce un risultato e si utilizza la parola riservata *void* come tipo di dato restituito

```
void scrive_risultati(float totale, int num_elementi);
```

- il codice una funzione può avere più di una istruzione return e termina non appena si esegue la prima di esse
- un errore tipico è quello di dimenticare l’istruzione return o scriverla dentro una sezione di codice che non verrà eseguita; in questi casi il risultato della funzione è imprevedibile e probabilmente porterà a risultati scorretti

Funzioni

- L'esecuzione di una funzione avviene attraverso una chiamata. La chiamata determina la sospensione dell'esecuzione delle istruzione del blocco di codice chiamante, che riprenderà al termine dell'esecuzione della funzione
- Con la chiamata di una funzione viene fornita ad una funzione la lista dei parametri effettivi (valori di ingresso)
- La chiamata di una funzione avviene all'interno di una espressione in cui compare il nome della funzione seguito dai parametri effettivi tra ()
- Nell'espressione la funzione partecipa fornendo un valore del tipo restituito



Funzioni

- Come per le variabili le funzioni devono essere definite prima di poter essere utilizzate
- E' possibile anticipare solo le intestazioni delle funzioni (prototipi)
- Il prototipo di una funzione:
  - non ha il corpo (perché esso verrà appunto definito altrove)
  - deve specificare il tipo dei parametri formali ma non necessariamente il loro nome
  - deve terminare con il punto e virgola ;

```
tipo_restituito nome_funzione (tipi_parametri_formali);
```

- Il prototipo non è necessario se la definizione di una funzione appare prima del suo utilizzo
- i prototipi delle funzioni vengono generalmente definiti in file header (.h) che vengono inclusi nei file sorgenti che utilizzano le funzioni

Funzioni

- E’ possibile indicare parametri con un argomento di default.
- Se si chiama la funzione omettendo il parametro, il compilatore lo inserirà automaticamente inizializzandolo con il valore di default.
- Gli argomenti di default devono trovarsi agli ultimi posti della lista

Esempio:

```
int f(int a, int b=2, int c=3)
      {return a+b+c;}
```

In chiamata:

f(10) = 10+2+3 = 15

f(10,20) = 10+20+3 = 33

f(10,20,30) = 10+20+30 = 60

Funzioni

Esempio

```
#include<iostream>
int MinimoInteri(int i, int j); // prototipo della funzione

int main()
{ int M;
  int A=5, B=6;
  M=MinimoInteri(A,MinimoInteri(B,3));
  std::cout << "M=" << M << std::endl;
}

int MinimoInteri(int i, int j) {
  if (i<j) return i;
       else return j;
}
```

Header files

Esempio, consideriamo il seguente programma:

```
#include <iostream>
void a( void ); // function prototype
void b( void ); // function prototype
void c( void ); // function prototype

int main() {
 int x = 5; // local variable to main
 a(); // a has automatic local x
 b(); // b has static local x
 c(); // c has automatic local x
 b(); // b has static local x
 std::cout << "local x in main is " << x << std::endl;
return 0; }

void a( void ) {
 int x = 25; // initialized each time a is called
 ++x; }

void b( void ) {
 static int x = 50;  x++;
 std::cout << "local x in b is " << x << std::endl;} // e' inizializzata una sola volta, la
prima volta che tale funzione viene chiamata e il suo valore resta inalterato quando si esce
dalla funzione.

void c( void ) {
 int x = 1;
 x *= 10; }
```

Header files

Lo si può dividere in tre files:

```
// file myfunctions.h
#include <iostream>
void a( void ); // function prototype
void b( void ); // function prototype
void c( void ); // function prototype
```

```
// file myfunctions.cpp
#include "myfunctions.h"
void a( void ) {
 int x = 25; ++x; }

void b( void ) {
 static int x = 50;
 x++;
 std::cout << "local x in b
  is " << x << std::endl;
}
void c( void ) {
 int x = 1; x *= 10; }
```

```
// file main.cpp
#include "myfunctions.h"
int main() {
int x = 5; // local variable to main
a(); // a has automatic local x
b(); // b has static local x
c(); // c has automatic local x
b(); // b has static local x
std::cout << "local x in main is " << x << std::endl;
return 0; }
```

Funzioni

In C++ ci sono 3 modi per passare parametri ad una funzione:

- per valore
- per riferimento con puntatori
- per riferimento con alias

Passaggio per valore

- Si crea una copia locale del parametro passato
- La modifica della copia locale non ha influenza sul parametro originale
- Utile quando si vuole fornire un valore solo di ingresso alla funzione chiamata

```
#include <iostream>
int f(int);
int main()
{ int dato=2;
  std::cout<<f(dato)<<std::endl; // stampa 9
  std::cout<<dato<<std::endl; //stampa 2
}

int f(int num){ num=num+1; return num*num; }
```

Funzioni

Passaggio per riferimento con puntatori

- Viene passato l'indirizzo della variabile, si accede al dato passato tramite un puntatore
- La modifica all'interno della funzione cambia il parametro originale
- La sintassi cambia

```
#include <iostream>
int f(int *);
int main(){
int dato=2;
std::cout<<f(&dato)<<std::endl; // stampa 9
std::cout<<dato<<std::endl; //stampa 3
}

int f(int * num){
 (*num)=(*num)+1;
 return (*num)*(*num);
}
```

Funzioni

Passaggio per riferimento con variabile alias

- Si accede al dato passato tramite una variabile alias
- La modifica all'interno della funzione cambia il parametro originale
- Modifica della sintassi solo nel prototipo e nell'intestazione della funzione

```
#include <iostream>
int f(int &);
void main(){
int dato=2;
std::cout<<f(dato)<<std::endl; //stampa 9
std::cout<<dato<<std::endl; //stampa 3
}

int f(int & num){
    num=num+1;
    return num*num;
}
```

Funzioni

Passaggio per riferimento con variabile alias

L'alias (o ''reference'') è un nome diverso per una variabile già istanziata. Un alias si definisce utilizzando l'operatore & suffisso al tipo di dato riferito, int& è il tipo "riferimento al tipo int"

int x=1;

int &y=x; // una reference quando viene creata deve essere sempre inizializzata!

std::cout<<x; //stampa 1

y=2;

std::cout<<x; //stampa 2

Funzioni di libreria

Funzioni matematiche disponibili nella libreria cmath (math.h)

#include<cmath> o #include <math.h>

sqrt(x) radice quadrata di un numero

exp(x) esponenziale

log(x) logaritmo naturale (base *e*)

fabs(x) restituisce il valore assoluto di x

sin(x), cos(x), tan(x) funzioni trigonometriche con angoli espressi in radianti

pow(x,y) elevamento a potenza pow(5,2)=5^2=25

verifiche alfanumeriche:

isalpha(c) ritorna true se e solo se c è una lettera maiuscola o minuscola.

islower(c)/ isupper(c) ritorna true se e solo se c è una lettera minuscola/maiuscola.

isdigit(c) ritorna true se e solo se c è una cifra (cioè un carattere da 0 a 9).

tolower(c)/ toupper(c) converte la lettera c in minuscola/maiuscola, se non lo è già

Array come parametri di funzione

Un array può essere un parametro di una funzione (ma non il risultato di una funzione). Ma essendo una variabile array un puntatore nel passaggio per valore non vengono realmente copiati i valori dell'array!!! (si ha di fatto un passaggio per riferimento). E' possibile passare un array per valore se lo si definisce come campo di una struttura dati (oggetto) passata per valore.

```
#define N 5
#include<iostream>
int RicercaLineare(int V[N], int D) {
  int trovato=0;
   for (int i=0; i<N; i++)
        if (V[i]==D) trovato=1;
   return trovato; }

int main() {
    int MioVettore[N]={1,6,7,3,21};
    int MioElemento;
    std::cout << "Inserire l'elemento da cercare" << std::endl;
    std::cin >> MioElemento;
     if (RicercaLineare(MioVettore,MioElemento))
        std::cout << MioElemento << " trovato" << std::endl;
     else std::cout << MioElemento << " non trovato" << std::endl; }
```

Array come parametri di funzione

Modi equivalenti di passaggio di array a funzioni:

```
void myFunction(int *param, int size)

void myFunction(int param[], int size)
```

il qualificatore const

il qualificatore const, come abbiamo visto, indica che una variabile non può più essere modificata dopo la sua inizializzazione.

Può anche essere usato nella dichiarazione dei tipi dei parametri passati ad una funzione

Come e quando utilizzare const:

Principio del privilegio minimo:

Ad una funzione si devono accordare i privilegi di accesso minimi indispensabili ai parametri passati per completare la propria funzione e nulla più

```
double f(double a){
 const double pi=3.1415656;
return 2*a*pi; }
```

il qualificatore const

```
#include<iostream>
void f(const int*, const int);

int main(){
 int array[]={0,1,2,3,4,5,6,7,8,9};
 f(array,10);
}

void f(const int* array, const int dim){
   for(int i=0;i<dim;i++){
         std::cout<< array[i];
 }
}
```

La funzione f non può modificare il dato puntato, può solo avere accesso in lettura

il qualificatore const

Quanto detto per i puntatori vale anche per gli alias

```
#include<iostream>
void f(const int&, int&);

int main(){
    int a,b;
    a=100;
    b=10;
    f(a,b);
    std::cout<< " " <<a<< " "<<b; //stampa 100 11
 }

void f(const int& data, int& datb){
    datb++;
    std::cout<<data<< " "<<datb; //data si può solo leggere
}
```

Namespace

I namespace permettono di raggruppare classi, funzioni e variabili sotto un unico nome. E' un modo per dividere l'ambito di visibilità globale (global scope) in sotto-ambiti (sub-scopes). Utile quando vi è la possibilità che vi siano due o più un oggetti/variabili globali o funzioni che abbiano lo stesso nome.

```
namespace first{
    int var=5;
 }
namespace second{
    double var=3.14;
 }

void main(){
  std::cout<<first::var<< " "<<second::var<<std::endl;
}
```

Namespace

Per poter accedere agli elementi di un namespace come se questi fossero definiti nell'ambito globale, si usa la direttiva **using.** In questo modo non si deve risolvere ogni volta il namespace:

using namespace identificatore;

```
namespace My{
    double var=3.14;
}

using namespace My;

int main(){
     std::cout<<var<<std::endl;
}
```

Namespace

Uno degli esempi più utili di namespace è quello della libreria standard C++. Tutte le classi, oggetti e funzioni della libreria standard C++ sono definite nel namespace "std", quindi è possibile scrivere

```
#include<iostream>
int main(){
  std::cout<< "Ciao" <<std::endl;
  return 0;
}
```

oppure in modo equivalente:

```
#include<iostream>
using namespace std;
int main(){
      cout<< "Ciao"<<endl;
      return 0;
}
```

variabili struct

- Gli array consentono l'aggregazione di variabili dello stesso tipo
- Le struct sono insiemi (aggregati) di tipi di dati diversi
- Supponiamo di voler immagazzinare i dati di una collezione di compact disc; i campi della struttura CD potrebbero essere cinque:

Nome campo	**Tipo di dato**
titolo	Stringa
artista	Stringa
numero canzoni	Numero intero
prezzo	Numero float
data di acquisto	Stringa

- il formato della dichiarazione di una struct è:

```
struct <nome della struttura>
{
    <tipo_dato_campo1> <nome_campo_1>;
    <tipo_dato_campo2> <nome_campo_2>;
    ...
    <tipo_dato_campon> <nome_campo_n>;
};
```

variabili struct

- Esempi equivalenti (definizione di un tipo di dato struct in C++):

```
typedef struct{
 int hour;
 int minute;
 int second;
} Time;
```

```
struct Time {
 int hour;
 int minute;
 int second; };
```

- I dati nel corpo della struttura sono chiamati membri della struttura
- La dichiarazione di una struttura non alloca spazio in memoria ma dichiara solo un nuovo tipo di dato
- Le struct **possono** contenere qualsiasi tipo di dato (anche altre strutture)
- **possono** contenere puntatori al proprio tipo di struttura (auto-referenti)
  (utili per definire strutture dati come *liste* e *alberi*)

variabili struct

- Esempi di definizioni di variabili struct e array di struct

```
Time timeObj;
Time timeArray[100];  tipo_struttura nome_array [dimensione]
Time *timePtr;
Time &timeRef=timeObj;
```

- Si accede ai membri di una struct tramite l'operatore "." oppure "->" se si utilizzano i puntatori

```
std::cout<< timeObj.hour; std::cout<<
timeRef.hour; timePtr=&timeObj;
std::cout<< timePtr->hour;
std::cout<< (*timePtr).hour; // parentesi tonde necessarie
perché l'operatore . (punto) ha la precedenza rispetto all'operatore *
```

variabili struct

- Esempio

```
#include <iostream>
typedef struct{
        int hour;
        int minute;
        int second; }
Time;

// prototypes
void print24H( const Time & );
void printStandard( const Time & );
```

```
int main() {
 Time dinnerTime; // variable of new type Time
 // set members to valid values
 dinnerTime.hour = 18;
 dinnerTime.minute = 30;
 dinnerTime.second = 0;
 std::cout << "Dinner will be held at ";
 print24H( dinnerTime );
 std::cout << " 24H time,\nwhich is ";
 printStandard( dinnerTime );
 std::cout << " standard time.\n";
 //set members to invalid values
 dinnerTime.hour = 29;
 dinnerTime.minute = 73;
 std::cout << "\nTime with invalid values: ";
 print24H( dinnerTime );
 std::cout << std::endl;
 return 0; }
```

variabili struct

- Esempio

```
// Print the time in 24H format
void print24H( const Time &t ) {
 std::cout << ( t.hour < 10 ? "0" : "" ) << t.hour
<< ":" << ( t.minute < 10 ? "0" : "" ) << t.minute;
}

// Print the time in standard format
void printStandard( const Time &t ) {
 std::cout << ( ( t.hour == 0 || t.hour == 12 ) ? 12 : t.hour % 12 )
<< ":" << ( t.minute < 10 ? "0" : "" ) << t.minute
<< ":« << ( t.second < 10 ? "0" : "" ) << t.second
<< ( t.hour < 12 ? " AM" : " PM" );
}
```

variabili struct

- I campi di una variabile struct dati possono essere inizializzati durante la definizione
- Diversamente dagli array una variabile struct può essere assegnata ad un'altra struct

- Esempio:

```
Time dinnerTime = {12,0,0};
Time t2;
t2= dinnerTime;
```

variabili struct

- Struct annidate: un campo di una struct può a sua volta essere di tipo struct
- Esempio:

```
struct info
{
  string nome;
  string indirizzo;
  string citta;
  string provincia;
  int cod_postale;
};
struct impiegato
{
  info anagrafica;
  double salario;
};
```

Ogni campo della sottostruttura si raggiunge mediante un doppio uso dell'operatore punto (.); ad esempio, supponiamo di avere definita una variabile x di tipo impiegato; se dobbiamo assegnargli il nome leggendolo dalla tastiera scriviamo:

```
cout << "Introduca il codice postale dell'impiegato: ";
cin >> x.anagrafica.cod_postale;
```

variabili struct

- Se si cambia l'implementazione della struct occorre cambiare qualcosa in tutte le parti del programma in cui si è utilizzata la struct

- Non si possono trattare le struct come oggetti atomici:
  - Non si possono confrontare direttamente per vedere se due istanze sono uguali (ma si devono confrontare singolarmente tutti i membri)
  - Non si può stampare una struct (ma si deve farlo per ogni sua parte)

- Esempio di definizione di un tipo di dato vettore di struct

```
typedef Time orari[10];
```

variabili struct

- Costruttore di una struct, inizializza il valore dei membri di una struct quando viene istanziata.

```
struct TestStruct {
  int id;
  TestStruct() {
    id = 42;
  }
};
```

```
TestStruct miastr;
std::cout << miastr.id << std::endl; // stampa 42
```

variabili struct

- Una struct può essere sia un parametro che il risultato di una funzione

- Esempio:

```
... Time dinnerTime, dinnerTimeSucc; // variable of new type Time
// set members to valid values
dinnerTime.hour = 23;
dinnerTime.minute = 30;
dinnerTime.second = 0;
dinnerTimeSucc = printOraSucc(dinnerTime);
...

Time printOraSucc ( const Time &t ) {
  Time ts=t;
  if (ts.hour < 23) ts.hour+=1;
  else if (ts.hour == 23) ts.hour=0;
return ts;
}
```

Stringhe

- Le stringhe sono uno dei tipi di dato più utili forniti dalla libreria standard del C++
- Una stringa è una variabile (oggetto) che memorizza una sequenza di caratteri
- La classe string è definita nel namespace ‘’std’’
- Creazione e assegnamento

```
string testString;
testString = "This is a string.";
string str2 = testString;
```

- Oppure

```
string testString = "This is a string.";
```

- Per utilizzare le stringhe

```
#include <string>
```

Stringhe

- Caratteri di controllo speciali

  \n per andare a capo all'interno di una string
  \t tabulazione orizzontale
  \r ritorno carrello (CR)
  \\ barra invertita
  \' apice
  \" virgolette

Stringhe

- La funzione **length** ritorna il numero di caratteri in una stringa, inclusi spazi e punteggiatura

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

int main() {
 string small, large;
 small = "I am short";
 large = "I, friend, am a long and elaborate string indeed";
 cout << "The short string is " << small.length() << " characters." << endl;
 cout << "The large string is " << large.length() << " characters." << endl;
return 0; }
```

- Output:

The short string is 10 characters.

The large string is 48 characters.

Stringhe

- L'accesso ai singoli caratteri avviene con l'operatore [].
- Gli indici variano da **0 a str.length() – 1**

```
#include <string>
#include <iostream>
using namespace std; int main() {
 string test; test = "I am Q the omnipot3nt";
 char ch = test[5]; // ch is 'Q'
 test[18] = 'e'; // we correct misspelling of omnipotent
 cout << test << endl;
 cout << "ch = " << ch << endl;
 return 0; }
```

- Output

I am Q the omnipotent

ch = Q

Stringhe

- Test di uguaglianza/disuguaglianza con operatori **==** e **!=**
- Test lessicografico (alfabetico) CASE-SENSITIVE con operatori <,<=,>,>=
- Espressioni vere: **"A" < "B", "App" < "Apple", "help" > "hello", "Apple" < "apple“**
- getline(istream& is, string& str) legge una riga dal flusso “is” (std::cin oppure un descrittore di file), spazi compresi, fino al fine-linea, getline(istream& is, string& str, char delim) si arresta quando trova il carattere separatore “delim”, che viene estratto/consumato da “is” ma non viene aggiunto alla string “str”

```
#include <string> #include <iostream>
using namespace std;
int main() {
 string myName = "Neal"; string userName;
 while (true) {
  cout << "Enter your name (or 'quit' to exit): ";
  getline(cin,userName);
  if (userName == "Julie") { cout << "Hi, Julie! Welcome back!" << endl; }
  else if (userName == "quit") { cout << endl; break; }
  else if (userName != myName) { cout << "Hello, " << userName << endl; }
  else { cout << "Oh, it’s you, " << myName << endl; }
} return 0; }
```

Stringhe

- Concatenazione di stringhe con operatori + e +=

```
#include <string>
#include <iostream>
using namespace std;
int main() {
 string firstname = "Leland";
 string lastname = " Stanford";
 string fullname = firstname + lastname; // concat the two strings
 fullname += ", Jr"; // append another string
 fullname += '.'; // append a single char
 cout << firstname << lastname << endl;
 cout << fullname << endl; return 0;
}
```

- Output

Leland Stanford

Leland Stanford, Jr.

Stringhe

- Ricerca di una sotto-stringa in una stringa con la funzione **find**
- Valore di ritorno: posizione in cui viene trovata la sotto-stringa oppure **string::npos** se la sotto-stringa non viene trovata
- Se viene specificato un secondo parametro intero la ricerca inizia in tale posizione

```
#include <string>
#include <iostream>
using namespace std;
int main() {
string sentence = "Yes, we went to Gates after we left the dorm.";
int firstWe = sentence.find("we"); // finds the first "we"
int secondWe = sentence.find("we", firstWe + 1); // finds "we" in "went"
int thirdWe = sentence.find("we", secondWe + 1); // finds the last "we"
int gPos = sentence.find('G');
int zPos = sentence.find('Z'); // returns string::npos
cout << "First we: " << firstWe << endl;
cout << "Second we: " << secondWe << endl;
cout << "Third we: " << thirdWe << endl;
cout << "Is G there? ";
cout << (gPos != string::npos ? "Yes!" : "No!") << endl;
cout << "Is Z there? ";
cout << (zPos != string::npos ? "Yes!" : "No!") << endl;
return 0; }
```

output:

```
First we: 5
Second we: 8
Third we: 28
Is G there? Yes!
Is Z there? No!
```

Stringhe

- Estrazione di una sotto-stringa con la funzione **substr(start, length)**
- **start** è il carattere di partenza, **length>0** è il numero di caratteri da estrarre

```
#include <string>
#include <iostream>
using namespace std;
int main() { string oldSentence;
 oldSentence = "The quick brown fox jumped WAY over the lazy dog";
 int len = oldSentence.length();
 cout << "Original sentence: " << oldSentence << endl;
 int found = oldSentence.find("WAY ");
 string newSentence = oldSentence.substr(0, found);
 cout << "Modified sentence: " << newSentence << endl;
 newSentence += oldSentence.substr(found + 4);
 cout << "Completed sentence: " << newSentence << endl; return 0; }
```

- Output

Original sentence: The quick brown fox jumped WAY over the lazy dog

Modified sentence: The quick brown fox jumped

Completed sentence: The quick brown fox jumped over the lazy dog

Stringhe

- inserimento **str1.insert(start, str2)**, inserisce str2 in str1 alla posizione start
- Sostituzione **str1.replace(start, length, str2)** sostituisce la porzione di stringa di str1 di length caratteri a partire da start con la stringa str2

```
#include <string>
#include <iostream>
using namespace std;
int main() { string sentence = "this is a test string.";
cout << sentence << endl;
// Insert "phrase" at position 5 in sentence
sentence.insert(5, "phrase ");
cout << sentence << endl;
// Replace the 5 characters " test" in "this phrase is a test string."
// with the string "n example" in sentence
sentence.replace(16, 5, "n example");
cout << sentence << endl; return 0; }
```

- Output:

this is a test string.

this phrase is a test string.

this phrase is an example string.

Stringhe

- conversione tra variabili stringa e variabili numeriche e viceversa

```
#include <iostream> // std::cout
#include <string> // std::string, std::to_string
#include <sstream> // istringstream
using namespace std;
int main () {
 string pi = "pi is " + std::to_string(3.1415926);
 string perfect = std::to_string(1+2+4+7+14) + " is a number";
 cout << pi << '\n';
 cout << perfect << '\n';
 string Text = "456"; // string containing the number
 int Result; //number which will contain the result
 istringstream convert(Text); // stringstream used for the conversion
 convert >> Result;
 cout << Result << endl;
}
```

- Output:

pi is 3.141593

28 is a number

456

lettura/scrittura file di testo

- Esistono due classi principali per la gestione dei file

ofstream: classe stream per scrivere su un file

ifstream: classe stream per leggere da un file

- il sistema di I/O del C++ si basa sul concetto di canale (stream) inteso come mezzo attraverso cui fluiscono le informazioni provenienti o inviate dai diversi dispositivi (tastiera, monitor, files)

- i file di testo sono sequenze di caratteri

- Per leggere o scrivere su un file è necessario:

1. includere nel programma C++ la libreria <fstream>
2. dichiarare una variabile di tipo file ifstream oppure ofstream;
3. collegare la variabile dichiarata ad un file (apertura in lettura o scrittura)
4. Al termine delle operazioni sul file, lo stream dovrà essere chiuso

lettura/scrittura file di testo

- Esempio di scrittura su file

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
 ofstream myfile ("example.txt");
 int i=5;
 if (myfile.is_open()) { // verifica che il file sia stato aperto
  myfile << "This is a line.\n";
  myfile << "This is another line.\n";
  myfile << i << " " << i+5;
  myfile.close();
 }
 else cout << "Unable to open file";
return 0; }
```

- Contenuto del file:

This is a line.

This is another line.

5 10

lettura/scrittura file di testo

- Esempio di lettura da file

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
   string line;
   ifstream myfile ("example.txt");
   if (myfile.is_open()) {
     while ( getline (myfile,line) ) {
       cout << line << '\n'; }
     myfile.close();
    }
   else cout << "Unable to open file";
return 0;
}
```

- Il programma legge iterativamente il file riga per riga fino alla fine del file. Ciascuna riga viene salvata in una variabile string e stampata a video

lettura/scrittura file di testo

- Esempio di lettura da file

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () {
 string nome;
 string cognome;
 int matricola;
 ifstream myfile ("elenco.txt");
 if (myfile.is_open()) {
   while (myfile >> nome >> cognome >> matricola ) {
     cout << nome << " " << cognome << " "<< matricola << endl;
   }
   myfile.close(); }
   else cout << "Unable to open file";
 return 0; }
```

lettura/scrittura file di testo

- Esempio di lettura da file con separatori

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;
int main () {
  string str; string nome; string cognome; int matricola;
  ifstream myfile ("elenco_sep.txt");
  if (myfile.is_open()) {
     while (getline(myfile, nome, '|') ) {
         getline(myfile, cognome, '|');
         getline(myfile, str);
         istringstream token(str);
         token >> matricola;
         cout << nome << " " << cognome << " "<< matricola << endl; }
         myfile.close(); }
  else
         cout << "Unable to open file";
  return 0;
}
```

lettura/scrittura file di testo

- Non mischiare mai >> e ''getline'' per leggere da un file, perché >> si arresta prima del fine linea, mentre getline ''consuma'' anche il fine linea
- L'operatore >> ignora gli spazi iniziali dal flusso di input
- Esempio

Voglio leggere queste due righe (con struttura diversa) da un file:

25
pippo topolino

ERRATO

```
string frase;
int n;
myfile >> n;
getline(myfile, frase);
cout << n << endl;
cout << frase << endl;
```

CORRETTO

```
string str, frase;
int n; ifstream
getline(myfile, str);
istringstream tk(str);
tk >> n;
getline(myfile, frase);
cout << n << endl;
cout << frase << endl;
```

lettura/scrittura file di testo

- Esempio (esercizio 3 Esercitazione di laboratorio 1 modificato):

In un esercizio di telepatia, un sensitivo scommette di essere in grado di indovinare almeno 3 numeri consecutivi, in una sequenza di N numeri interi pensati da uno spettatore.
Si scriva un programma che carichi da un file la sequenza dei numeri dello spettatore (il file contiene N nella prima riga e i numeri della sequenza nella seconda riga separati da spazi), e verifichi se il sensitivo dice la verità.
Il programma acquisisce dal sensitivo i 3 numeri (da tastiera).
Il programma legge il file e crea un array di dimensione N, con allocazione dinamica, e verifica se esiste, nella sequenza di N numeri, una sotto-sequenza di 3 numeri esattamente uguale a quella inserita dal sensitivo.

lettura/scrittura file di testo

- Opzioni di apertura file

ios::in **Input (default per ifstream)**

ios::out **Output (default per ofstream)**

ios::app **Output con append (si apre un file già esistente e si scrive in fondo)**

Esempio (apertura di un file in scrittura con append):

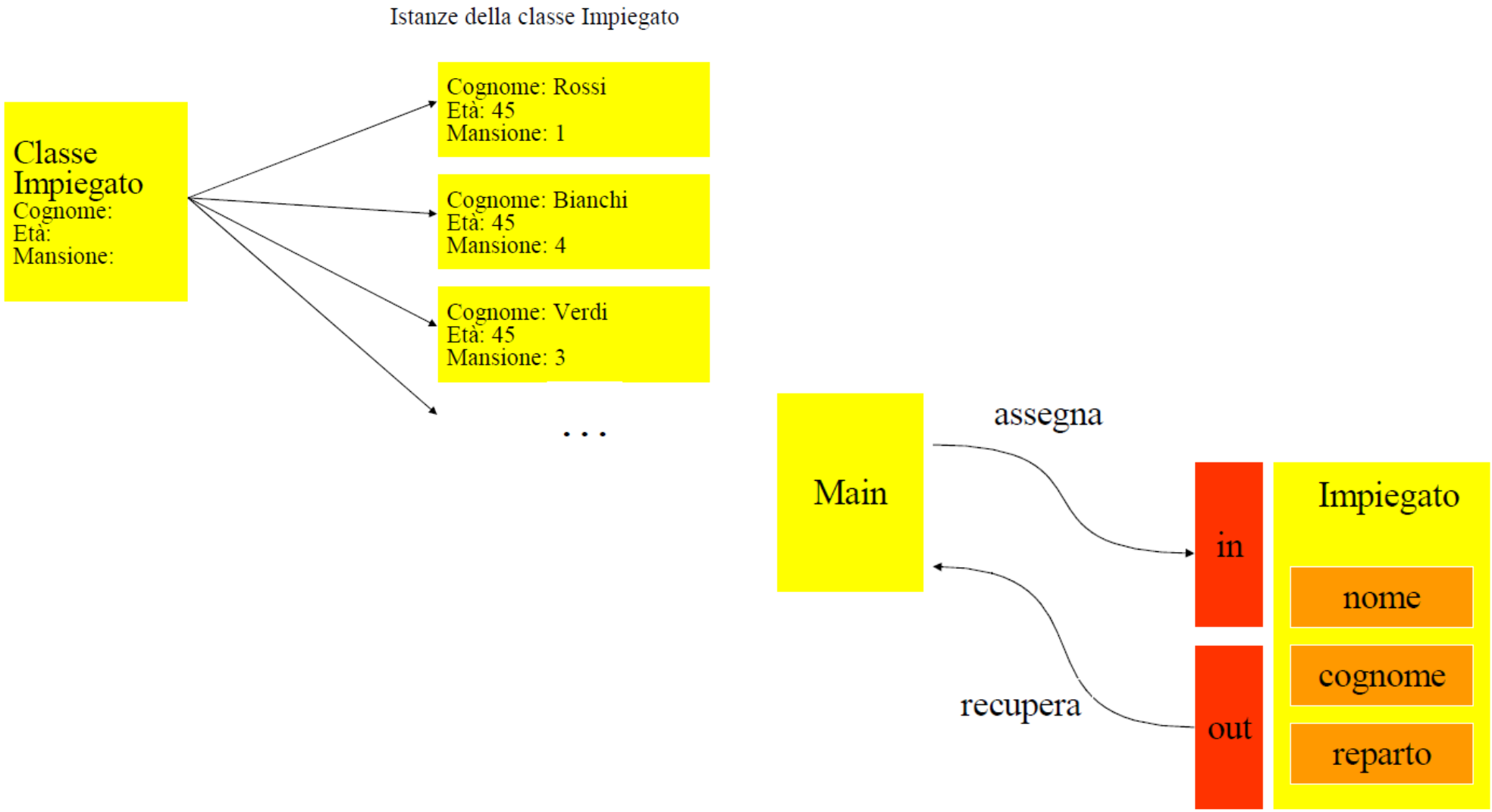
```
ofstream myfile ("example.txt", ios::app);
```

Classi

- Programmazione orientata agli oggetti (Object Oriented Programming OOP)
- si incapsulano i dati e le funzioni all'interno di **classi**
- le classi definiscono dei prototipi per gli **oggetti**
- un oggetto è una istanza di una classe
- fra una classe e un oggetto vi è la stessa relazione che sussiste fra un tipo e una variabile
- un programma viene visto come un insieme di oggetti che interagiscono tra loro

- proprietà fondamentale è l'occultamento delle informazioni (information hiding): i dettagli rimangono nascosti e non visibili al di fuori della classe
  - questo è un argomento fondamentale per garantire robustezza da un punto di vista dell'ingegneria del software
  - è come guidare un automobile senza conoscere il funzionamento del motore

Classi


Istanze della classe Impiegato
Classe Impiegato
Cognome:
Età:
Mansione:
Cognome: Rossi
Età: 45
Mansione: 1
Cognome: Bianchi
Età: 45
Mansione: 4
Cognome: Verdi
Età: 45
Mansione: 3
. . .
Main
assegna
recupera
in
out
Impiegato
nome
cognome
reparto

Classi

- Una classe si dichiara con la parola chiave *class* ed è costituita da:
- **Dati:** dati membro (chiamati anche attributi)
- **Funzioni:** funzioni membro (chiamati anche metodi)
- La dichiarazione del tipo dei suoi membri è fra "{ }" e terminata da ";"
- i dati membro e le funzioni membro possono essere in una fra le seguenti sezioni (vedi più avanti):
  - public
  - protected
  - private (è il caso di default)
- i dati membro possono essere di qualunque tipo valido, con eccezione del tipo della classe che si sta definendo
- definita una classe, possono essere generate istanze della classe, cioè oggetti:

```
nome_classe   identificatore ;

Punto P; Semaforo S;
```

Classi

- la definizione di un metodo consiste di quattro parti:
  - il tipo restituito dal metodo
  - il nome del metodo
  - la lista dei parametri formali (eventualmente vuota) separati da virgole
  - il corpo del metodo racchiuso tra parentesi graffe
- Le funzioni membro (metodi) di una classe possono essere definite:
  - all'interno della dichiarazione stessa della classe

```
class Articolo_Vendite {
public:
 bool articolo_uguale (const Articolo_Vendite & art) // definizione
   {return iva == art.iva; }                          // funzione
private:   // membri privati
   Articolo_Vendite iva;
};
```

  - all'esterno della classe tramite l'operatore "::" detto operatore binario di risoluzione di visibilità

Classi

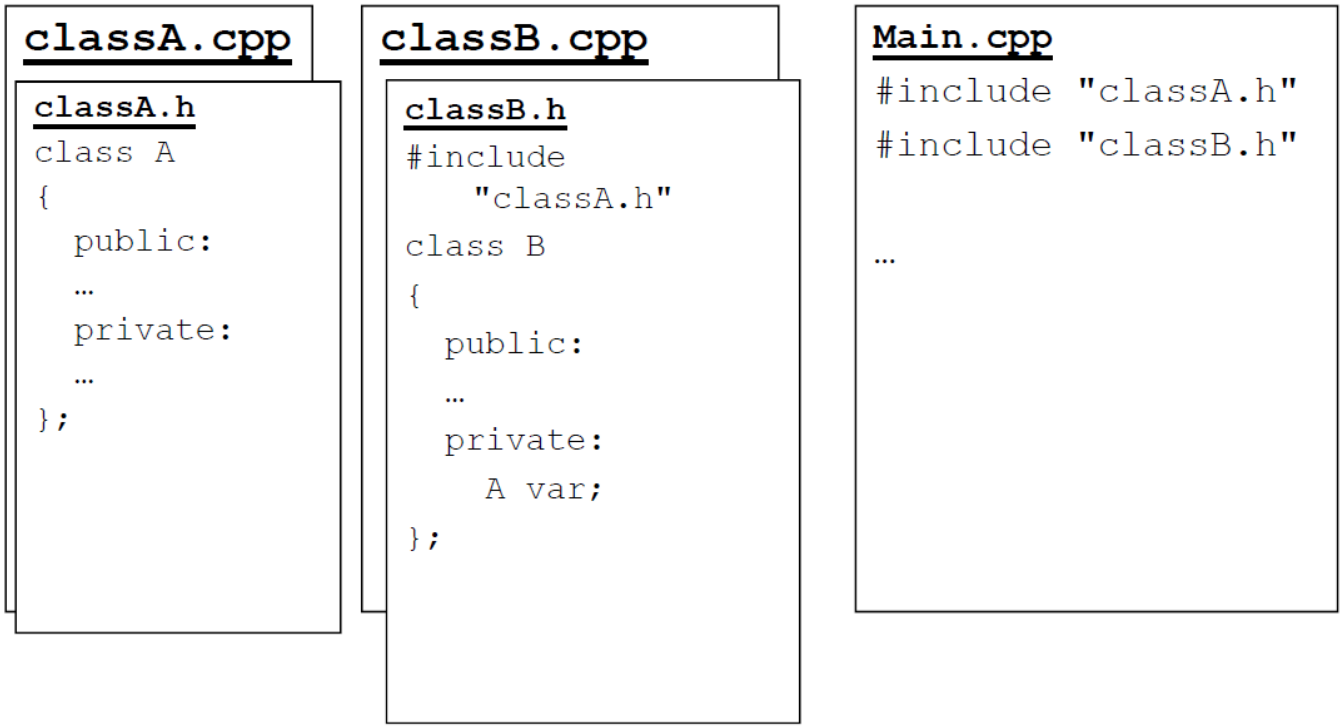
- Esempio di una classe e di un metodo definito esternamente ad essa

```
class NomeClasse {
   public:
     void set(int); //membri pubblici
   private:
     int a; //membri privati };

void NomeClasse::set(int var) {
   a=var;
}
```

Compilazione separata

- separazione tra la definizione di una classe e le altre parti del programma che la usano. Vantaggi:
  - riuso: parti separate facilmente riusabili (libreria)
  - compilazione selettiva



classA.cpp

classA.h

```
class A
{
  public:
  …
  private:
  …
};
```

classB.cpp

classB.h

```
#include
   "classA.h"
class B
{
  public:
  …
  private:
    A var;
};
```

Main.cpp

```
#include "classA.h"
#include "classB.h"

…
```

Classi

- Esempio di definizione di una classe in un file header

```
#include <iostream>

class Time {
 public:
  Time(); //constructor
  void setTime( int, int, int ); //set hour, minute, second
  void print24H(); //print 24H time format
  void printStandard(); //print standard time format
 private:
  int hour; // 0 - 23
  int minute; // 0 - 59
  int second; // 0 - 59 };
```

Classi

- Esempio di implementazione di una classe in un file sorgente che include il file header

```
#include "esempio_1_24_5_Time.h"
// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
  Time::Time() { hour = minute = second = 0; }
// Set a new Time value using 24H time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s ) {
 hour = ( h >= 0 && h < 24 ) ? h : 0;
 minute = ( m >= 0 && m < 60 ) ? m : 0;
 second = ( s >= 0 && s < 60 ) ? s : 0; }
// Print Time in 24H format
void Time::print24H() {
 std::cout << ( hour < 10 ? "0" : "" ) << hour << ":"
 << ( minute < 10 ? "0" : "" ) << minute; }
// Print Time in standard format
void Time::printStandard() {
 std::cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
 << ":" << ( minute < 10 ? "0" : "" ) << minute
 << ":" << ( second < 10 ? "0" : "" ) << second
 << ( hour < 12 ? " AM" : " PM" ); }
```

Classi

- Esempio file principale che contiene il main

```
int main() {
 Time t; // instantiate object t of class Time
 std::cout << "The initial 24H time is ";
 t.print24H();
 std::cout << "\nThe initial standard time is ";
 t.printStandard();
 t.setTime( 13, 27, 6 );
 std::cout << "\n\n24H time after setTime is ";
 t.print24H();
 std::cout << "\nStandard time after setTime is ";
 t.printStandard();
 t.setTime( 99, 99, 99 ); // attempt invalid settings
 std::cout << "\n\nAfter attempting invalid settings:" << "\n24H time: ";
 t.print24H();
 std::cout << "\nStandard time: ";
 t.printStandard();
 std::cout << std::endl;
 return 0;
}
```