Lezione 14

Stream, fstream e file

Output streams

- *stream:* sequenza di caratteri
- *ostream:* meccanismo per convertire valori di vario tipo in sequenze di caratteri
  - Output formattato
- *standard output ostream* e *standard error ostream:* ***cout*** e ***cerr***
  - tipicamente collegati al terminale da cui è fatto partire il programma
  - appartengono al namespace *std*

Input streams 1/2

- *istream*: meccanismo per convertire sequenze di caratteri in valori di diverso tipo
- *standard istream*: ***cin***
  - tipicamente associato al terminale da cui è fatto partire il programma
  - appartiene al namespace *std*

Input streams 2/2

- Come si leggono valori?
- Operatore >> (*leggi*, *estrai*)
- Input formattato ...

Input formattato

- Le cifre sono convertite in numeri se si leggono interi o reali
- I caratteri speciali sono saltati se si leggono singoli caratteri
  - ad esempio è saltato il newline '\n'
- Gli spazi bianchi (spazio, tab, newline, form-feed, ...) sono saltati se si leggono stringhe

Stream state

- Ciascun (*i/o*)*stream* è associato ad uno stato
  - Insieme di *flag* (valori booleani)
- Errori e condizioni non standard sono gestiti settando o testando in modo appropriato lo stato

End Of File (*EOF*)

- Classica condizione che impedisce di effettuare ulteriori operazioni: leggere la marca *EOF* da uno stream di input
  - Nel caso in cui si stia effettivamente leggendo un file attraverso l'*istream*, vuol dire che si è raggiunta la fine del file
  - Nel caso di input da un terminale UNIX, l'*EOF* è generato se si preme *Ctrl-D* (su riga vuota)

Controllo stato istream 1/2

- *cin* e *cin>>...*, oppure *!cin* e *!(cin>>...)*, possono essere utilizzati dove è atteso un valore booleano
  - Es.:, *cin>>dim* ha un valore di ritorno
  - Vero: la prossima operazione può aver successo
  - Falso: la prossima operazione fallirà
    - La precedente operazione è fallita: formato errato dell'input oppure incontrato *EOF*

Controllo stato istream 2/2

- Una volta in stato *non-buono*, lo stream ci rimane finché i flag non sono esplicitamente resettati
- Semplice modo per resettare lo stato dello stream:
  - *cin.clear()*
- Operazioni di input su stream in stato non-buono sono operazioni nulle
- Quindi: resettare *prima* di effettuare la prossima operazione di input

Esercizio

- Scrivere un programma che, dopo aver letto da *stdin* una sequenza di numeri interi, stampi la somma dei valori letti
- La lunghezza della sequenza **non è nota a priori**, **nè comunicata prima** di iniziare ad immettere i numeri
- Soluzione nella prossima slide

Esercizio

```
#include <iostream>

using namespace std ;

main()
{
      int i, somma = 0 ;
      while (cin>>i)
            somma += i ;
      cout<<"Somma: "<<somma<<endl ;
      // se volessi continuare ad usare
      // il cin, dovrei prima invocare
      // cin.clear()
}
```

Due argomenti extra

- I prossimi due argomenti, ossia ignorare caratteri e formattare l'output, non saranno argomento d'esame

Ignorare caratteri

- Quando una operazione di input fallisce, nessun carattere è rimosso dallo stream di input
  - *cin.ignore()* ignora il prossimo carattere (rimuovendolo dallo stream)

Soluzione non sicura

```
int main()
{
   int *p ; // attenzione, per ora contiene un valore casuale !!!
   int dim = -1 ;

   do { // immissione dimensioni array
      cout<<"Dimensioni array? " ;
      cin>>dim ;
   } while (dim < 0) ;

   p = new int[dim] ; // allocazione memoria

   ...
}
```

Che succede se si immette un carattere anziché una cifra?

Una soluzione sicura

```
int main()
{
   int *p ; // attenzione, per ora contiene un valore casuale !!!
   int dim = -1 ;

   do { // immissione dimensioni array
      cout<<"Dimensioni array? " ;
      while(!(cin>>dim)) {
            cin.clear() ;
            cin.ignore();
            cout<<"Devi immettere un numero: " ;
      }
   } while (dim < 0) ;

   p = new int[dim] ; // allocazione memoria

   ...
}
```

Però non esce in caso di EOF!
Se volete, la soluzione è lasciata a voi ...

Buffer

- Si indica col termine **buffer** (memoria tampone) un array di byte utilizzato nelle operazioni di I/O
- Le operazioni di uscita con gli stream sono tipicamente *bufferizzate*
- Ad esempio il passaggio dei caratteri da stampare al terminale non avviene carattere per carattere, bensì i caratteri vengono spediti tutti assieme quando si inserisce il newline

Buffer e incoerenza dell'uscita

- Si fa così per ragioni di efficienza
- Si possono però avere problemi di incoerenza delle informazioni in uscita
  - Ad esempio se un programma fallisce dopo una scrittura su *cout* in cui non si è inserito il newline, i corrispondenti caratteri potrebbero non essere mai passati al terminale
- Stiamo per vedere un problema simile con le scritture su file

Formattazione dell'output

- I seguenti argomenti di formattazione dell'output non saranno argomento di esame

Esempio di output formattato

```
What's your name? Paolo
Health (in hundredths)? 35
Welcome to GOTA, Paolo. And good luck!

Paolo----------------------------------------------------------------
| Health points: 035/100| ######                                     |
|--------------------------------------------------------------------|
>
```

Giustificato a sinistra

Lunghezza proporzionale agli health points

80 colonne (obbligatorio)

Formattazione dell'Output

- La formattazione è controllata da un insieme di flag e valori interi
- Semplice interfaccia: *funzioni* e *manipolatori*

setprecision

- *cout.setprecision(int n)*

  Setta il massimo numero di cifre per un numero in virgola mobile

  - l'effettivo output dipende dal formato (generale, scientifico, fisso)

  - l'effetto è *persistente*: influenza tutte le prossime operazioni di uscita, fino alla prossima evetuale chiamata di *setprecision*

Manipolatore

- Operazione che modifica lo stato nel mezzo degli oggetti che si stanno leggendo o scrivendo

- Esempi di manipolatori che non prendono argomenti:
  - *flush* svuota il buffer di uscita
  - *endl* inserisci un newline e svuota il buffer di uscita

Manipulatori con argom. 1/2

- Spesso si vuole riempire con del testo predefinito un certo spazio su una linea

- *cout<<...<<setw(int n)<<...*

  Setta il minimo numero di caratteri per la prossima operazione di uscita

- *cout<<...<<setw(...)<<setfill(char c)<<...*

  Sceglie il carattere in *c* come carattere di riempimento

Manipulatori con argom. 2/2

- Per usare manipolatori che prendono argomenti bisogna includere:

  #include <iomanip>

Stampa dello stato del gioco

```
#include <iostream>
#include <iomanip>

using namespace std ;

int main()
{
  int punti_salute = 35 ;

  cout<<left<<setw(80)<<setfill('-')<<"Paolo"<<endl ;
                              cout<<"|              Health              points:
    "<<right<<setw(3)<<setfill('0')<<punti_salute<<"/100 | " ;
  int num_asterischi = punti_salute*51/100 ;
  cout<<setw(num_asterischi)<<setfill('#')<<"" ;
  cout<<setfill(' ') ;
  cout<<setw(53 - num_asterischi)<<right<<" |"<<endl ;
  cout<<setw(80)<<setfill('-')<<""<<endl ;

  return 0 ;
}
```

streams

- *cout*, *cerr*, *cin* sono già pronti all'uso quando un programma parte
- Sono settati automaticamente
- Però possiamo anche creare i nostri *stream*
  - In generale, dobbiamo specificare a cosa sono associati

fstreams

- I seguenti tipi di *stream* sono supportati direttamente dalla libreria standard del C++
  - Sono associati ai *file*
- *ifstream*: file stream di ingresso (lettura)
- *ofstream*: file stream di uscita (scrittura)
- *fstream*: file stream di ingresso/uscita
- Presentati in <[*i* | *o*]*fstream*> o in <*fstream*> (tutti e tre)

Associazione a file

- Un (i|o)fstream viene associato ad un file mediante un'operazione chiamata **apertura** del file
  - Da quel momento in poi tutte le operazioni di ingresso/uscita fatte sullo *stream* si tradurranno in identiche operazioni sul file
  - E' il sistema operativo che si occuperà di tutti i dettagli (che variano da sistema a sistema) necessari per eseguire le operazioni sulla macchina reale

Associazione a file

- Come nome del file si può indicare tanto un percorso assoluto che un percorso relativo
- Esempio di percorso assoluto:
  `/home/paolo/dati.txt`
  File di nome dati.txt nella cartella /home/paolo
- Esempi di percorsi relativi (il file è cercato nella cartella corrente):
  `paolo/dati.txt`
  File di nome dati.txt nella sottocartella paolo della cartella corrente

  `dati.txt`
  File di nome dati.txt nella cartella corrente

Associazione a file

- E' possibile aprire più di un file contemporaneamente
- L'apertura di un file può fallire per diversi motivi
  - Ad esempio se si tenta di aprire in lettura un file inesistente
- E' opportuno controllare sempre l'esito dell'operazione di apertura prima di utilizzare un (*i*|o)*fstream*

Apertura file 1/3

- Un file è aperto in input definendo un oggetto di tipo *ifstream* e passando il nome del file come argomento

```
ifstream f(“nome_file”) ;
if (!f) cerr<<”l'apertura è fallita\n” ;
```

- Un file è aperto in output definendo un oggetto di tipo *ofstream* e passando il nome del file come argomento

```
ofstream f(“nome_file”) ;
if (!f) cerr<<”l'apertura è fallita\n” ;
```

Apertura file 2/3

- Un file può essere aperto per l'ingresso e/o l'uscita definendo un oggetto di tipo *fstream* e passando il nome del file come argomento
- Deve essere fornito un secondo argomento *openmode: ios_base::in* oppure *ios_base::out*
- Se non esiste, un file aperto in scrittura viene creato, altrimenti viene troncato a lunghezza 0

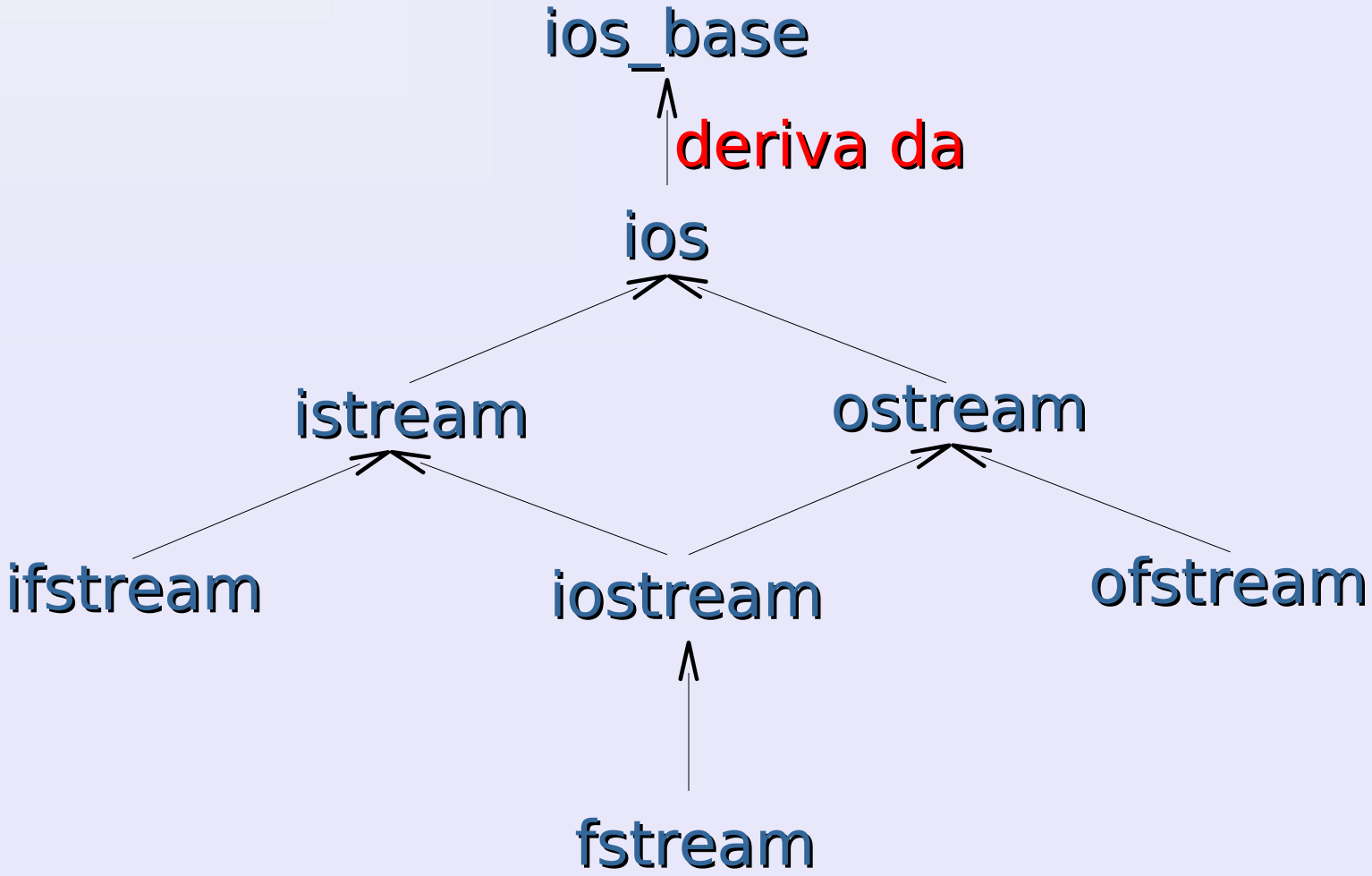
Apertura file 3/3

- Esempi:

```
// file aperto in ingresso
fstream f("nome_file", ios_base::in) ;
if (!f) cerr<<"apertura fallita\n" ;

// file aperto in uscita
fstream f2("nome_file", ios_base::out) ;
if (!f) cerr<<"apertura fallita\n" ;
```

Gerarchia degli stream



Conseguenza immediata

- Si possono usare tutti gli operatori, i flag di stato, e le funzioni di utilità per la formattazione viste per gli stream di ingresso/uscita standard

- Quindi si può quindi controllare lo stato di un oggetto (*i*|*o*)*fstream* `f` usando il suo identificatore in una espressione condizionale
  Es.:

```
if (!f)
      cerr<<"La precedente operazione è"
          <<"fallita"<<endl ;
```

Controllo successo operazione

- Come visto negli esempi, con oggetti di tipo **`ifstream`**, **`ofstream`** o **`fstream`** si controlla il successo o meno dell'apertura controllando lo stato
- Più modi possono essere combinati mediante l'operatore |
- *in*: input, *out*: output, *app*: append (si continua a scrivere a partire dal fondo)

  ofstream f("nome_file", ios_base::app) ;

Chiusura file

- Un file può essere chiuso invocando la funzione *close()* sullo stream associato

- Un file è chiuso implicitamente alla distruzione dello *stream* associato
  Es.: `f.close() ;`

- La chiusura (esplicita o implicita) di un file è importante perché solo all'atto della chiusura ne è garantito l'effettivo aggiornamento (svuotamento dei buffer)

open

- Si può anche aprire un file invocando la funzione *open* su uno *stream* non associato ad alcun file (non inizializzato o deassociato mediante *close*)
  - Per brevità non vedremo la *open* in queste lezioni

Esercizio

- *file/file.cc*

Domanda

- Siete riusciti a risolvere l'esercizio con letture e scritture formattate?
- Probabilmente no, perché l'operatore >> ha saltato i caratteri `'\n'`
- Stiamo per vedere la soluzione mediante I/O non formattato

Esercizio

- Leggere da un file di testo “dati.txt” una sequenza di numeri interi di al più 100 elementi, finché non si trova il primo elemento uguale a 0. Memorizzare tutti i numeri letti in un vettore.

Soluzione

```
main()
{
   int vett[100] ;
   ifstream f(“dati.txt”);
   if (!f)
            cerr<<”Errore di apertura file\n";
   else
            for (int i = 0 ; i < 100 && f>>vett[i]
                              && vett[i] != 0 ; i++)
                              ;
}
```

Esercizi (senza soluzione)

- ESERCIZIO 1: Leggere da un file di testo “dati.txt” una sequenza di numeri interi terminata da 0. Memorizzare in un vettore tutti i numeri negativi.

- ESERCIZIO 2: Leggere da un file di testo “dati.txt” una sequenza di numeri interi terminata da 0. Memorizzare in un vettore tutti i numeri compresi tra –30 e +30 escluso lo 0. Ordinare il vettore in modo crescente e stampare tutti i numeri positivi.

- ESERCIZIO 3: Leggere da un file di testo “dati.txt” una sequenza di caratteri terminata da *. Memorizzare in un vettore tutti i caratteri alfabetici.

Esercizio

- Scrivere in un file di testo “valori_pos.txt” tutti i numeri strettamente positivi di un vettore contenente 100 valori interi.
  Alla fine, inserire il valore -1

Soluzione

```
main()
{
   int vett[100] ;
   ofstream f(“dati.txt”);
   if (!f)
            cerr<<”Errore di apertura file\n";
    else {
            for (int i=0; i<100; i++)
               if (vett[i]>0)
                     f<<vett[i];

            f<<-1 ;
        }
 }
```

Esercizi (senza soluzione)

- ESERCIZIO 1: Scrivere in un file di testo “carat.txt” tutti i caratteri di una stringa letta da input. Terminare la sequenza di caratteri del file con *.

- ESERCIZIO 2: Leggere da un file di testo “dati_inp.txt” una sequenza di numeri interi terminata da 0, e copiare in un vettore solo gli elementi positivi. Copiare tutti i valori del vettore compresi fra 10 e 100 in un file di testo “dati_out.txt”.

- ESERCIZIO 3: Come l’esercizio 4.c. In più, stampare su schermo il contenuto del file “dati_out.txt”.

Formattazione

- Le operazioni formattate interpretano il contenuto delle caselle di uno *stream* come dei caratteri
- Le operazioni non formattate non interpretano il contenuto delle caselle di uno *stream* e non effettuano alcuna trasformazione
  - Trasferiscono semplicemente sequenze di byte da uno *stream* agli oggetti passati per argomenti, o dagli oggetti passati per argomento ad uno *stream*

Funzioni membro

- Le funzioni di lettura e scrittura non formattate che vedremo sono delle **funzioni membro**
  - appartengono di fatto al tipo di dato per cui sono definite
- Per utilizzarle bisogna usare la notazione a punto
  - Se **f** è un *istream*, allora per usare una funzione membro `fun(char &)`, bisogna scrivere `f.fun(c) ;`

  Es.:

```
char c ;
cin.fun(c) ;
```

Buffer ed I/O non formattato

- Come già visto, un **buffer** è un array di bytes utilizzato nelle operazioni di I/O
- Il tipo `char` ha esattamente la dimensione di un byte
- Per questo motivo il tipo `char` è utilizzato anche per memorizzare byte nelle letture e le scritture non formattate

Input non formattato

- Sono disponibili le seguenti funzioni:

```
get()

get(char &)

get(char *buffer, int n,
    char delimitatore='\n')

read(char *buffer, int n)

gcount()
```

get()

- Ritorna, su un **`int`**, il valore del prossimo byte nello stream di ingresso a cui è applicata
  - Ritorna il valore ***EOF*** in caso di fine input
- Esempio:

```
main()
{
 int i = cin.get() ;
  ...
}
```

get(char &c)

- Preleva un byte e lo assegna alla variabile passata come argomento
  - La variabile è lasciata inalterata in caso di fine input
- Esempio:

```
main()
{
 char c ;
 cin.get(c) ;
 ...
}
```

Esercizio

- ***file_conta_linee.cc***

Lettura in un buffer

```
get(char *buffer, int n,
      char delimitatore='\n')
```

- Legge bytes dallo stream di ingresso e li trasferisce in buffer aggiungendo il carattere `'\0'` finale
- La lettura va avanti finché non si sono letti n byte oppure non è stato incontrato il delimitatore
  - Se il terzo argomento non è passato, si usa come delimitatore il codice del carattere `'\n'`

Esempio di lettura in un buffer

```
main()
{
 char buf[100] ;

 // lettura di al più 50 bytes, a meno
 // non si incontri il codice del
 // carattere '-'
 // in fondo è inserito '\0'
 cin.get(buf, 50, '-') ;
 ...
}
```

Lettura in un buffer 2

`read(char *buffer, int n)`

- Legge **`n`** bytes e li memorizza nell'array **`buffer`**

`gcount()`

- Ritorna il numero di caratteri letti nell'ultima operazione di lettura

Scrittura non formattata

`put(char c)`

- Trasferisce un byte (il contenuto di **`c`**) sullo *stream* di uscita

`write(const char *buffer, int n)`

- Trasferisce i primi **`n`** byte di **`buffer`** sullo *stream* di uscita

Nota

- Nel caso del **`cin`**, le letture non formattate leggono necessariamente caratteri in codifica ASCII
- Nel caso del **`cout`** e del **`cerr`** le scritture non formattate possono non far apparire alcun carattere
- Al contrario, letture e scritture non formattate sono fondamentali per i file binari, come stiamo per vedere

Modo testo e file di testo

- Se si effettuano solo letture e scritture formattate su uno stream, si dice che lo si sta usando in **modo testo**
- In maniera simile, un file i cui byte sono da interpretarsi come codici di caratteri si definisce un **file di testo**
- Altrimenti si parla di **file binario**

Esempio

```
void fun(const char *a)
{
     ofstream f(“nome”) ;
     f.write(a, sizeof(char) * 2) ;
}

main()
{
     char b[3] = {'h', 'z', 'k'} ;
     fun(b) ;
}
```

Puntatori ed array

- Estendiamo le nostre conoscenze
- Il tipo
  `<tipo> *`
  memorizza indirizzi
- Come mai possiamo passare un array come argomento attuale nella posizione corrispondente ad un parametro formale di tipo `<tipo> *` ?
- Perché passare il nome di un array è equivalente a passare l'indirizzo del primo elemento dell'array

Oggetti e puntatori

- Il tipo **`char *`** memorizza indirizzi
  - Possiamo scriverci dentro l'indirizzo di oggetti di **qualsiasi classe di memorizzazione**: statica, automatica o dinamica
  - Non solo, possiamo anche scriverci dentro l'indirizzo di oggetti di **tipo diverso** da array di caratteri
    - Dobbiamo però convertire il tipo dell'indirizzo mediante un:

```
reinterpret_cast<char *>(<indirizzo>)
```

  - Si può anche aggiungere il const

Trasferimento array

- Se vogliamo leggere/scrivere un array da/su file
- Come facciamo a trasferire solo l'effettivo numero di byte occupati dall'array?
- Utilizziamo l'operatore `sizeof` per conoscere le dimensioni di ogni elemento, e moltiplichiamo per il numero di elementi
- Ovviamente siamo liberi trasferire qualsiasi numero di byte, ma attenzione ad errori di gestione della memoria

Esempio trasferimento array

```
void fun(const int *a)
{
   ofstream f("nome") ;
   f.write(
   reinterpret_cast<const char *>(a),
            sizeof(int) * 3) ;
}

main()
{
      int b[3] = {1, 2, 7} ;
      fun(b) ;
}
```

Scrittura su file binari

- Cosa abbiamo fatto?
- Abbiamo scritto nel file *nome* la rappresentazione in memoria, byte per byte, dell'array `b`
- *nome* è certamente un file binario
- Esercizio: *file/scrivi_leggi_array.cc*

Domanda

- Potevamo scrivere gli elementi uno alla volta nel file binario?

Domanda

- Potevamo scrivere gli elementi uno alla volta nel file binario?
- Sì, ma sarebbe stato molto inefficiente
- Abbiamo dovuto invece scrivere gli elementi uno alla volta nel caso del file di testo
  - Ma l'efficienza non si è persa, perché, come già detto, l'operatore di uscita si preoccupa di bufferizzare le informazioni al posto nostro

Esempio con struct 1/2

```
main()
{
 struct part {char nome[10]; int tempo}
        mario ;

 strcpy(mario.nome, "Mario") ;
 mario.tempo = 30 ;
 char * p =
        reinterpret_cast<char *>(& mario) ;
}
```


L'operatore unario & ritorna l'indirizzo dell'oggetto passato come argomento

Esempio con struct 2/2

- In `p` è finito l'indirizzo in memoria dell'oggetto struttura `mario`
- La conversione si è resa necessaria perché `p` punta ad oggetti di tipo diverso
- Come facciamo ad accedere solo all'effettivo numero di byte occupati da `mario`?
- Utilizziamo l'operatore `sizeof`

Scrittura su file binari 1/2

```
main()
{
 struct part {char nome[10]; int tempo}
        mario ;

 strcpy(mario.nome, "Mario") ;
 mario.tempo = 30 ;
 char * p =
        reinterpret_cast<char *>(& mario) ;
 ofstream f("dati.dat") ;
 f.write(p, sizeof(mario)) ;
}
```

Scrittura su file binari 2/2

- Cosa abbiamo fatto?
- Abbiamo scritto nel file *dati.dat* la rappresentazione in memoria, byte per byte, dell'oggetto **`mario`**
- *dati.dat* è certamente un file binario

Lettura da file binari

- Come facciamo a rimettere in memoria le informazioni salvate nel file?
- *file_binario.cc*
- Prima di andare avanti è opportuno osservare che quanto fatto con un oggetto di tipo `struct` è solo un esempio di lettura/scrittura da file binario
- Si potevano fare esempi con matrici o array di oggetti struttura, e così via ...

Accesso sequenziale e casuale

- Uno stream è definito ad **accesso sequenziale** se ogni operazione interessa caselle dello stream consecutive a quelle dell'operazione precedente
- Uno stream è definito ad **accesso casuale** se per una operazione può essere scelta arbitrariamente la posizione della prima casella coinvolta
- Per **`cin`**, **`cout`** e **`cerr`** è definito solo l'accesso sequenziale

Accesso casuale ai file

- La casella a partire dalla quale avverrà la prossima operazione è data da un contatore che parte da 0 (prima casella)
- Il suo contenuto può essere modificato con le funzioni membro

`seekg(nuovo_valore)` per file in ingresso
(la g sta per get)

`seekp(nuovo_valore)` per file in uscita
(la p sta per put)

Accesso casuale ai file

- Le due funzioni possono anche essere invocate con due argomenti

  `seekg(offset, origine)`

  `seekp(offset, origine)`

- L'origine può essere:

  `ios::beg` offset indica il numero di posizioni a partire dalla casella 0 (equivalente a non passare un secondo argomento)

  `ios::end` offset indica il numero di posizioni a partire dall'ultima casella (muovendosi all'indietro)

Lettura della posizione

- Per gli ifstream è definita la funzione

  `tellg()`
  Ritorna il valore corrente del contatore

- Per gli ofstream è definita la funzione

  `tellp()`
  Ritorna il valore corrente del contatore

Passaggio di stream

- Uno stream può essere passato per riferimento ad una funzione

  `cin` può essere passato come **`istream &`**

  `cout` può essere passato come **`ostream &`**

  un **`ifstream`** può essere passato come **`istream &`**

  un **`ofstream`** può essere passato come **`ostream &`**

Esempio

```
void stampa(ostream &o) {
 o<<"Stringa"<<endl ; }

void leggi(istream &i) {
 char s[100] ;
 i>>s ; cout<<s<<endl ; }

main()
{
 stampa(cout) ; // stampa su stdout
 ofstream f("nome_file.txt") ;
 stampa(f) ; // scrive nel file
 leggi(cin) ; // legge da stdin
 ifstream f2("nome_file.txt") ;
 leggi(f2) ;  // legge dal file
}
```

Esercizio

- Dato un file binario in cui sono memorizzati oggetti di tipo

```
struct  persona {
            char    codice[7];
            char    Nome[20];
            char    Cognome[20];
            int     Reddito;
            int     Aliquota;
        }   ;
```

ed assumendo che ogni persona abbia un codice univoco …

Esercizio

- Scrivere una funzione che prenda in ingresso un oggetto *P* di tipo persona per riferimento, ed un istream (associato al file binario). La funzione cerca nel file un oggetto con lo stesso valore del campo codice dell'oggetto *P* e, se trovato, riempie i restanti campi dell'oggetto *P* con i valori dei corrispondenti campi dell'oggetto trovato nel file.
  La funzione ritorna *true* in caso di successo, ossia se l'oggetto è stato trovato, *false* altrimenti

Soluzione

```
bool ricerca_file(persona &P, istream &f)
{
  bool trovato=false;
    while (true) {
        persona buf[1] ;
        f.read(reinterpret_cast<char *>(buf),
                    sizeof(persona)) ;
        if (f.gcount() <= 0)
              break ;
        if (strcmp(buf[0].Nome, P.Nome) == 0) {
              P = buf[0] ;
              trovato = true ;
              break ;
        }
  }
   return trovato;
}
```

Osservazioni finali 1/2

- I file sono una delle strutture dati fondamentali per la soluzione di problemi reali
- Infatti, nella maggior parte dei casi reali, i dati non si leggono da input e non si stampano su video, ma si leggono da file e si salvano su file

Osservazioni finali 2/2

- Per questo motivo, si rende necessario gestire grandi quantità di dati su supporti di memoria secondaria in modo molto efficiente
  - Vedrete come negli insegnamenti di “BASI DI DATI”
- I file saranno inseriti nella prova di programmazione, quindi …