Parte seconda

13) Le eccezioni per il controllo degli errori di programma try catch e throw

La parola try già l' abbiamo usata nel capitolo 9 dentro il testo Pdf, oppure dentro il forum con il messaggio 9 , ora vedremo meglio di che cosa si tratta intanto la sintassi è:

try { blocco try}

catch(tipo1 argomenti) {blocco catch} catch(tipo2 argomenti) {blocco catch} ecc… catch(tipoX argomenti) { blocco catch}

La vera eccezione la lancia throw eccezione ; dove possiamo dire che è uguale all' istruzione return di una funzione dove ci restituisce un risultato, invece throw ci restituisce un errore , infatti per eccezioni si intende un errore, oppure throw è simile alla istruzione goto perché ci manda verso la funzione catch, cosi salta tutto quello che arriva dopo l' eccezione, nel caso che l' errore non è intercettato il programma segue il normale flusso, l' eccezione è un tipo di dato che può essere int, char double , l' eccezione di throw viene poi gestita o catturata da catch (tipo_gestione) { descrizione tipo errore } throw deve essere eseguita dentro un blocco try questo viene fatto con chiamata diretta oppure indiretta tramite funzioni , l' importante che sia prima del relativo catch che tradotto vuol dire proprio cattura, mentre throw vuol dire getta o gettare mentre try la troviamo anche come processare oppure verificare ecc….

L' eccezione possiamo gestirla da dentro il main oppure da dentro una funzione vediamo un uso dentro il main() quando chiamiamo la funzione catch la variabile che uso ER non è importante, ma serve a noi per leggere il tipo errore, ma è importante il tipo di dato emesso da throw che nel nostro caso è double se nella funzione catch inserite int anzi che double il compilatore VS2015 da errori e esce una finestra per le eccezioni di errore vedete l'istruzione cout quella non verrà mai eseguita finche' c'è l' istruzione throw ecco il programma anzi che usare cout uso cerr che serve proprio per la gestione degli errori cerr si usa uguale a cout :

```
#include<iostream> using namespace std; int main() { cout << " inizio delle eccezioni \n";

    try    {

cerr << "\n Qui siamo dentro il blocco try e lanciamo una eccezione di tipo\n double 10.2 ora il programma trova l'istruzione [ throw 10.2 ] ";

        throw 10.2;                cerr << " NUOVA ISTRUZIONE che viene saltata ";        }
```

catch (double ER) { cerr << "\n con [catch (double ER)], (double ER) raccoglie il tipo\n di errore lanciato da throw, ER e'usato come variabile \n";
cerr << " Abbiamo raccolto l'errore numero ER: " << ER << endl; cerr << "\n Puo seguire altre istruzioni sul tipo di errore\n e come risolvere il problema.\n"; }

cout << "\n Per USCIRE usiamo [system (\"pause\")] con i comandi di try\n si esce direttamente come se ci fosse un exit(1) \n "; system("pause"); }

Effettivamente se si toglie throw, basta mettere come se fosse un commento, vedete cambia tutto il risultato troviamo la NUOVA ISTRUZIONE e subito dopo l' istruzioni per l' uscita, ora non c'è traccia di catch e delle sue istruzioni.

Ora lo stesso programma ma try throw e catch tutto dentro una funzione tER è tipo_errore , se abbiamo la condizione tra if (tER) throw tER; con le chiamate delle funzioni funzione(3); funzione(5.1); funzione(0); funzione('z'); tutte sono chiamate eccetto la funzione (0); , per funzione('z'); abbiamo il numero 122 che corrisponde al carattere z se cambiate il carattere cambia anche il numero anche tra caratteri maiuscoli e minuscoli per far funzionare il mio programma copiato o incollato ricordate di spostare using namespace std; sotto l' #include :

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

void funzione (int tER)

{ try

{ cerr << "\n Qui siamo dentro il blocco try dentro la funzione ora\n lanciamo una eccezione di tipo tER ora il programma \n trova (tipo ERRORE = tER )tramite l'istruzioni \n [ if(tER) throw tER ] ";

if (tER) throw tER;

} catch (int ER) { cerr << "\n con [ catch (double ER) ], (double ER) raccoglie il tipo\n di errore lanciato da throw, ER e'usato come variabile \n";
cerr << " Abbiamo raccolto l'errore numero ER: " << ER << endl;

cerr << "\n (Puo seguire altre istruzioni sul tipo di errore\n e come risolvere il problema.) \n"; } }

int main() { cerr << " inizio delle eccezioni \n";

funzione(3); funzione(5.1); funzione(0); funzione('z');

cout << "\n Per USCIRE usiamo [ system (\"pause\") ] con i comandi di try\n si esce direttamente come se ci fosse un exit(1) \n "; system("pause"); }
```

Come si vede dal risultato ogni volta che la funzione è chiamata viene inizializzata, poi secondo il confronto viene scritta o meno l' istruzioni seguenti all' if, con la funzione 0 la parte di catch non viene eseguita

Possiamo utilizzare più istruzioni catch legate ad un comando try anche qui si deve avere un tipo di overload ovvero non possono essere uguali ad esempio possiamo vedere un throw che lavora su diversi tipi pertanto dobbiamo raccogliere diversi tipi di errori sviluppiamo il nuovo programma che grosso modo è simile a quello già sviluppato in precedenza ovvero viene inserito poche differenze vedete tutto quello racchiuso dentro le righe è la nuova modifica che è segnato con il marcatore tutto il resto rimane come prima, l' errore lo genera il comando della funzione (0); :

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

void funzione (int tER)

{  try    { cerr << "\n Qui siamo dentro il blocco try dentro la funzione ora\n lanciamo una eccezione di tipo tER  ora il programma \n trova (tipo ERRORE = tER )tramite   l'istruzioni \n [  if(tER)  throw tER ] ";

          if (tER) throw tER;                else throw "\n[ NEW ERRORE funzione(0) NEW ]  \n";    }

   catch (int ER)     {              cerr << "\n con [ catch (double ER) ],  (double ER) raccoglie il tipo\n di errore lanciato da throw, ER e'usato come variabile  \n";
   cerr << " Abbiamo raccolto l'errore numero ER: " << ER << endl;

          cerr << "\n (Puo seguire altre istruzioni sul tipo di errore\n e come risolvere il problema.) \n";

          }      catch (char * stringa)

   { cerr <<"\n----------------------------------------------"<< "\n\n con [ catch (char * stringa) ],   raccoglie \n il tipo di errore lanciato da throw,\n nell' [else throw \" NEW ERRORE funzione(0) NEW \"]  \n e raccogliamo l' errore \n "; cerr << stringa;    cerr << "\n Abbiamo raccolto il nuovo errore\n\n"<<"----------
----------------------------------\n"; }

} int main() {         cerr << " inizio delle eccezioni  \n";  funzione(3);
funzione(5.1);   funzione(0);     funzione('z');

   cout << "\n Per USCIRE usiamo [ system (\"pause\") ] con i comandi di try\n si esce direttamente come se ci fosse un exit(1) \n ";   system("pause");}
```

Possiamo raccogliere tutti i tipi di eccezioni senza dover avere per forza un specificato tipo, di solito l' utilizzo di questo tipo di raccolta di errori viene fatto per ultimo come se fosse il default del comando switch case per la raccolta usiamo il comando catch(…){blocco catch} proprio i 3 puntini acconsente

qualsiasi tipo di dato ecco un programma sviluppato come è possibile vedere con una sola istruzione catch(…) posso avere 4 tipi diversi di dati :

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

void funzione (int tER) { try

{ cerr << "\n Qui siamo dentro la funzione ora lanciamo una eccezione \n";

if (tER == 10) throw tER; if (tER == 20) throw 'E';

if (tER == 30) throw 1.20; if (tER == 40) throw 67; } //nota

catch (...) { cerr << " catch raccoglie: "; } }

int main() { cout << " inizio delle eccezioni \n";

funzione(10); cerr << "{ Errore N 10 = intero da variabile } \n";

funzione(20); cerr << "{ Errore N 20 = carattere } \n";

funzione(30); cerr << "{ Errore N 30 = double } \n";

funzione(40); cerr << "{ Errore N 40 = intero da numero }\n";

cout << "\n\n [ Per USCIRE (1 tasto+invio) ] "; char a; cin>>a ; }
```

Era possibile usare anche 2 istruzioni catch per raccogliere solo la prima e l' ultima ovvero la variabile tER (tipo_Errore) e l' ultima il numero intero tutto il resto poteva essere raccolto con la catch(…) basta inserire questa nuova catch(int) prima della catch(…) e alla fine del blocco try nel punto dove su ho scritto //nota ecco il testo da inserire vediamo che i numeri interi passeranno per questa nuova istruzione :

```
catch (int) { cerr << "\n ---> raccolta numero intero ---> \n"; }
```

Un' altro modo per usare le eccezioni è avere un elenco tipi dentro la funzione grazie alla definizione che può essere inserita dentro la stessa funzione dopo la normale funzione possiamo inserire throw(tipo1,tipo2, ecc..) ritorniamo alla nostra funzione che abbiamo sempre usato e la modifichiamo: void funzione (int tER) throw(int,char,double) cosi throw riconosce solo i tipi descritti dentro le parentesi però non è veloce e comodo vediamolo in dettaglio con il solito programma ad ogni chiamata di funzione corrisponde un suo try(funzione ()) e un suo catch(tipo){istruzioni} :

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

void funzione (int tER)throw(int,char,double )

{ cerr << "\n Qui siamo dentro la funzione ora lanciamo una eccezione \n";
```

```
        if (tER == 10) throw tER;    if (tER == 20) throw 'E';      if (tER ==
30) throw 1.20;      if (tER == 40) throw 67;

} int main() {      cout << " inizio delle eccezioni  \n";

        try { funzione(10);    }      catch (int i)

{  cerr << "\n  raccolta numero intero > " << i << "\n";

   cerr << "{ Errore N 10 = intero da variabile } \n";    }

        try { funzione(20);   }       catch (char c)

{  cerr << "\n  raccolta numero char > " << c << "\n";

   cerr << "{ Errore N 20 = carattere } \n";       }

        try { funzione(30); }       catch (double d)

{  cerr << "\n  raccolta numero double > " << d << "\n";

   cerr << "{ Errore N 30 = double } \n";   }

        try    {      funzione(40);}    catch (int i)

{  cerr << "\n  raccolta numero intero > " << i << "\n";

   cerr << "{ Errore N 40 = intero da numero }\n";     }

   cout << "\n\n [ Per USCIRE (1 tasto+invio) ] "; char a; cin>>a ;    }
```

sempre con throw possiamo rilanciare uno stesso errore basta utilizzare throw; vediamo dentro la funzione rilanciamo throw , poi nel main() richiamiamo l' eccezione tramite catch (int m); :

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

void funzione () {    cerr << "\n Qui siamo dentro la funzione \n lanciamo una
eccezione:";

   try    { throw 101; }    catch (int i) {  cout << " Errore n " << i << "\n";
   throw;  }    }

int main () {         cout << " Inizio dell' eccezioni \n";

   try    { funzione(); }     catch (int m) {cout << "\n Raccolta 2a eccezione in
main\n Errore n " <<m <<"\n";}

   cout<< "\n\n [ Per USCIRE (1 tasto+invio) ] "; char a; cin >> a;  }
```

La cosa migliore per usare le eccezioni è quella di usare le classi in modo che incassa, grazie all' incapsulamento delle stesse classi, tutte le informazioni che riguarda l' eccezione e ogni classe deve avere il proprio errore sviluppo un programma cattura errori, se il risultato di una divisione ci porta sotto lo 0.01

allora è errore perché è un numero negativo ecco il programma sviluppato se lo copiate direttamente ricordate per farlo funzionare bisogna spostare almeno using namespace std ; che va sotto l' #include<iostream> per il resto può rimare uguale altrimenti per ogni graffa aperta o chiusa e dopo ogni punto e virgola si va con la nuova riga, eccetto per le 2 funzioni costruttore inline ecco il rapporto :ho creato la classe classe_errore1 dove si costruisce il rapporto dell' errore, tipo e numero errore il numero viene catturato da int errore; e successivamente da err , mentre il tipo nome viene catturato dal array char array_errore[80]; poi successivamente sarà arr , poi abbiamo preso 3 variabili per creare la divisione double numero, num1, num2; , con il do {… } while (num1 != 0); ho creato il ciclo finche il primo numero immesso è diverso da 0 si gira , quando num1 è uguale a 0 si esce dentro il try si controlla che il numero ovvero il risultato della divisione non sia minore di 0.01 if (numero <= 0.01) in questo caso throw lancia l' errore chiamando la funzione costruttore posizionata dentro la classe throw classe_Errore1(numero , " <<< ERRORE questo numero e' negativo\n uguale o minore di 0.01 >>> "); la chiamata è per la classe_Errore1 con catch (classe_Errore1 oe) abbiamo catturato il tipo errore e con lui abbiamo creato l' oggetto della classe per le chiamate oe cerr << oe.array_Errore << " -" << oe.errore

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

class  classe_Errore1{  public:

    int errore;          char  array_Errore [80];

    classe_Errore1() { errore = 0; *array_Errore = 0; }

    classe_Errore1(int err , char *arr)  {     strcpy(array_Errore , arr);      errore = err;    }      };

int main ()  {  double numero, num1, num2;   do {

cout << " > Cattura errori che sono i numeri negativi \n i numeri positivi non sono errori <\n\n  { per numeri negativi si considera numeri sotto lo 0.01 } \n  " <<"\t   <[[per uscire dal ciclo immettere 0]]>\n";

            try {  cout << "\n Immettere due numeri positivi:\n" <<" immettere primo numero "; cin >> num1;

                    cout << " ... secondo numero "; cin >> num2;        system ("cls");

                    cout << "calcolo divisione... tra " <<num1 << " / "<<num2<<"\n" ;

                    numero = num1 / num2;      cout << "\n\n risultato [" << numero <<"]\n\n";

                    if (numero <= 0.01)                  { cout << endl;
```

```
    throw classe_Errore1(numero , " <<< ERRORE questo numero e' negativo\n uguale o minore di 0.01 >>> ");

                }            }     catch (classe_Errore1 oe)

          {      cerr << oe.array_Errore << " -" << oe.errore << "\n\n"; }


    }      while (num1 != 0);

    system("cls");     cout << "\n\n USCITA [premere un tasto+ invio] "; char zx; cin >> zx; }
```

Quando si usa eccezioni per classi derivate da classi base bisogna prima raccogliere le eccezioni per la classe derivata poi si deve raccogliere gli errori per la classe padre, conseguenza se si raccoglie prima le eccezioni tramite la funzione catch quella della classe padre poi il catch non leggerà più errori nella classe figlia perché in catch ogni classe padre o base risponde ad una classe figlia o derivata ogni catch della classe primaria raccoglie errori anche per la classe secondaria

Nel caso che una eccezione lanciata da throw non viene trovata ecco che il compilatore o il sistema stesso richiama le funzioni void unexpected(); e void terminate(); queste funzioni fanno parte della libreria std del C++tramite l'header <exception>, normalmente quando una funzione lancia un errore non consentito da throw il compilatore richiama la funzione unexpected(); che richiama la funzione terminate();. Quando il sistema non trova l' istruzione catch corrispondente a quel tipo di errore, oppure quando il programma rilancia errori che non ci sono, allora viene richiamata la funzione terminate (); che richiama la funzione abort(); che serve per uscire da un qualsiasi programma in qualsiasi circostanza di errore

14) Input e Output del C++ approfondimenti su stream cout ecc…

In C era possibile formattare il testo tramite i printf () ,scanf() mentre qui con cout e cin non possiamo fare nulla, invece si, scrivete nel compilatore cout . e esce fuori una bella lista di funzioni metodi ecc…, questa lista serve proprio per accedere alle formattazioni del testo ecc…. e non solo, la nuova libreria input e output è sostenuta dal file header iostream che opera dentro il namespace std; per prima cosa ricordiamo che cosa sono i stream , è il flusso dei dati che vanno e tornano, dati che vanno dal pc alla stampante , oppure dati che dalla tastiera vanno al pc, altri dati che dal disco fisso li passiamo alle chiavette usb ecc…., tutti questi dati sia file testo che file binari sono i stream e tutti questi hanno in comune che sono input e output dello stream e sono sotto il controllo

del preprocessore che è i file header iostream qui in C++, mentre prima era comandati nel vecchio C dal file header stdio.h

Già con l' inserimento del comando #include<iostream> ora e prima con <stdio.h > vengono aperti automaticamente 3 sistemi di stream che sono stdin, stdout, stderr , dove std sta per standard, invece in = input ingresso, err= error errori, out= output uscita , ma il vero significato della parola è produzione , in input abbiamo quello che entra ovvero dati che entrano verso il computer, quando leggiamo un cd o dvd o quando scriviamo sulla tastiera quello è input , quello che scriviamo dentro un cd o dvd oppure la stampa di un foglio o la stampa su video, ovvero tutto quello che va dal computer verso le periferiche quello è output, l' errori sono i messaggi che riceviamo e questi sono scritti da cerr<< , l'output lo scriviamo con cout<< e l' input lo scriviamo con cin<<, questi comandi e altri come leggere creare i file con fopen ecc.… sono tutti stream, tradotto stream vuol dire ruscello o corrente, infatti corrente luce o dati che corrono verso le periferiche come una corrente di un fiume grazie i bus che sono i cavi interni del computer che collega le varie periferiche al pc , per parlare di dati parliamo di flusso stream è il flusso di dati I/O che va e viene verso il microprocessore del pc

Sotto il preprocessore, che sono i file header, supporta queste operazioni di I/O specialmente nel nuovo header iostream che legge tutta la lista di tutti i file header, (vedete dentro l' esplora soluzioni del compilatore (ctrl+alt+L), dentro i file (dipendenze esterne) li vedete tutta la lista dei file header caricati dentro il programma), praticamente sono caricati tutti i file header , il nostro sistema I/O si basa su delle classi template la nostra classe che useremo di più deriva dalla classe principale basic_ios, e si chiama ios_base se aprite la guida del compilatore e nell' indice si scrive (ios_base) poi scegliete classe e non metodo li nella classe si vede tutta la lista delle funzioni e metodi adottati da questa classe con le rispettive descrizioni

Comunque esiste 3 tipi di classe principali le classe template che fa capo al (basic_nome) esempio basic_ios, poi c'è la classe per caratteri estesi che come inizio è sempre (wnome) wios inoltre c'è la classe per caratteri semplici con nome ios oppure come nella guida ios_base i nomi di queste classi I/O che nascono da ios_base sono : fstream, ifstream, ios, iostream, istream, istringstream ofstream, ostream, ostringstream, streambuf e stringstream, ecco una tabella della CNR di bologna, da base_ios nasce ios, da ios nasce istream e ostream, da istream abbiamo istringstream, ifstream e iostream, da ostream nasce iostream, ofstream e ostrnstream da iostream nasce fstream e stringstream come detto prima per accedere ad i stream in C++ abbiamo cin>> per la tastiera e per il video abbiamo cout<<,cerr<<, clog<< (simile a cerr<< ma con buffer completo) poi ci sono la versione caratteri estesi wide che inizia con w anziché con c pertanto abbiamo win, wout, werr,wlog per formattare i nostri testi dobbiamo usare i comandi allegati alla classe di ios_base per i stream cout<<,cerr<<, clog<< e cin>> per la lista completa dei membri della

classe come detto vedere la guida del compilatore (F1 in indice cercare ios_base, poi aprite su (classe) c'è tutta la lista e il significato di ogni parola chiave nelle parole chiave con il nome in italiano non è quella aprite il collegamento e vedete il vero nome come binaria è binary ma vi porta dentro openmode, ugualmente anche nella guida online in internet)

Ora vediamo come si accede alla classe ios per attivare le formattazioni che ci serve la funzione setf() che attiva le formattazioni , la sua sintassi è: stream.setf(classe::flag); lo stream come detto è cin, cout , ecc… setf sta per settare i flag la classe deve essere una delle classi dello stream fstream, ifstream, ios, iostream, istream, ofstream, ostream, streambuf , il flag è la lista che si apre quando usiamo l' operatore di scope (::), per attivare setf() dopo lo stream. ,dopo il punto si apre la lista delle classi e funzioni dove possiamo scegliere anche la funzione setf() un esempio per usare un numero scientifico usiamo prima la formattazione cout.setf (ios::scientific); poi facciamo seguire il numero che dobbiamo formattare cout<<12.33; abbiamo il risultato a schermo il numero scientifico 1.23300000e+01 se desideriamo formattare più cose possiamo farlo utilizzando l' OR | cosi possiamo scrivere 2 istruzioni o più con un solo cout.setf(ios::showpoint | ios::showpos | ios::ecc..); ora consideriamo che non mi serve il segno + positivo inserito da showpos d'avanti i numeri positivi, io allora posso eliminare questa istruzione con la parola unsetf() che provoca la disattivazione dei flag si scrive : cout.unsetf(ios::showpos); … poi segue il numero successivo cout <<70; cosi il simbolo + d'avanti il nuovo numero è disattivato

Tramite l' overload possiamo modificare la forma di setf () vediamo in dettaglio e guardare dopo la virgola ,classe::flagx dove flagx deve essere uno dei flag precedenti alla virgola, esempio se flagx è flag1 solo questo flag1 viene attivato , possiamo anche avere più flag anche qui divisi dall' OR | classe::flag2 in questo caso entrambi i flag sarebbero accettati :

stream.setf(classe::flag1 | classe::flag2 , classe::flagx | classe::flag2);

Un esempio pratico showpos inserisce un segno (+ più) verso i numeri positivi , mentre showpoint inserisce dei zeri dopo un numero reale con la virgola ecco un piccolo programma f sta per funzione fa, fb, ferase, fexit chi copia il mio programma deve inserire (using namespace sts;) per il resto può rimanere cosi oppure dopo ogni ; e } si va a capo in una nuova linea il programma scrive e inizializza 4 numeri 1 intero e 3 double, le 4 variabili sono inserite nella parte globale , poi li facciamo stampare su video e in seguito mettiamo i flag, poi ristampiamo i nuovi risultati, questo è uguale per le 2 funzioni fa(), fb() , con ferase() si annulla il settaggio dei flag di fa() in modo che fb() sia di nuovo con i numeri iniziali delle 4 variabili iniziali globali se non c'è la funzione ferase() è possibile vedere che i nuovi numeri sono stampati come all' uscita della funzione fa(), per togliere ferase() basta inserire il commento /* ferase()*/, fexit() serve per uscire dal programma, guardare la posizione / chiamata delle 4 funzioni

dentro il vero programma int main() { fa (); ferase (); fb (); fexit (); } , ecco il programma sviluppato nell' esecuzione confrontate i dati che sono dentro le righe :

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

int a = 12;    double b = 177.100, c = -100.12,  d = 1.8;

void fa (); void ferase (); void fexit (); void fb ();      int main ()  {  fa ();  ferase ();  fb (); fexit (); }

void fa () {

   cout << "-- A Numeri inseriti: " << "\n " << a << " // " << b << " // " << c << " // " << d << " risultato:\n" << " ----------";

   cout.setf (ios::showpoint | ios::showpos,ios::showpos | ios::showpoint);

   cout << "\n >>> " << a << " * " << b << " * " << c << " * " << d << " <<<\n"<<" ----------";

   cout << "\n prodotto dall'algoritmo:\n [cout.setf (ios::showpoint | ios::showpos \n  ,ios::showpos |ios::showpoint);]\n\n";       }

void ferase ()  {cout << "\n<> Cancelliamo l' algoritmi precedenti con l'istruzioni<>: \n      [cout.unsetf (ios::showpoint | ios::showpos);]\n\n" ;
   cout.unsetf (ios::showpoint | ios::showpos);   }

void fexit ()    {cout  << "\n ByMpt-Zorobabele\n Uscita (premere un tasto + invio): "; char zx; cin >> zx;    }

void fb () {

   cout << "-- B Numeri inseriti: " << "\n " << a << " // " << b << " // " << c << " // " << d << " risultato:\n" << " ----------";

   cout.setf (ios::showpoint | ios::showpos,ios::showpos);

   cout << "\n >>> " << a << " * " << b << " * " << c << " * " << d << " <<<\n ----------\n";

   cout << "\n prodotto dall'algoritmo:\n [cout.setf (ios::showpoint | ios::showpos,ios::showpos );]\n come possibile vedere e' attivo solo showpos \n con il + nei numeri positivi \n";          }
```

Vediamo alcuni flag e il loro utilizzo : adjustifield racchiude internal, left , right, dove left e right inserisce caratteri di riempimento left riempie la sinistra , right riempie la destra e internal riempie il centro del campo , basefiel racchiude dec, hex e oct rispettivamente inserisce o estrae i valori numerici in formato

decimale, esadecimale e ottale floatfield racchiude fixed e scientific inserisce i valori in virgola mobile , in formato virgola fissa o formato scientifico ovvero con o senza il campo dell' esponente boolalpha inserisce o estrae oggetti di tipo bool true o false anziche i numeri 0 negativo o altri positivi showbase inserisce i prefissi per i numeri ad esempio se si utilizza un numero esadecimale prima del numero inserisce 0x la base esadecimale showpoint inserisce in modo non condizionale un punto decimale in un campo a virgola mobile ovvero tratta i numeri in diversi modi 1 quintale è 100 chili ma è anche 0.1 tonnellate , sono anche 1000 ettogrammi showpos inserisce in un campo positivo il segno + skipws elimina i spazi vuoti iniziali , se skipws() è uguale a 0 i spazi vuoti non vengono eliminati unitbuf scarica l' output dopo ogni inserimento uppercase cambia le scritte minuscolo in maiuscolo n uppercase N

Se desideriamo conoscere la posizione o meglio situazione dei flag se sono attivi o meno senza modificare il loro stato, ios ci mette a disposizione una funzione che ci riconsegna un numero intero (long) esadecimale con la base corrente di ogni flag 1 attivo 0 non attivo oppure true o false , la funzione è flags() e il suo formato è fmtflags flags(); con ios::fmtflags f; in f abbiamo lo stato in cui si trova il flag che come detto può essere un numero di tipo long 0 o 1 il fmtflags è un typedef e riconosce solo i suoi "membri" che è la lista dei tipi dei flag copio una funzione per leggere questo stato di tutti i flag la funzione leggi_stato_flag{ con il comando ios::fmtflags riconosce i tipi di typedef e in f legge lo stato dei flag con f=(long)) cout.flags(); , con la variabile long i; si crea il ciclo per leggere tutti i flags for (i=0x4000; i; i=i >> 1) con >> 1 si presume che di parlare controlli di bit a bit infatti si usa il controllo if (i & f) cout<<" true 1 "; else cout<<" false 0 "; & and a bit , se i e f sono uguali scrivi true 1 altrimenti false 0, ecco la funzione prototipo :

```
leggi_stato_flag {( ios::fmtflags f; long i; f=(long) cout.flags();

for (i=0x4000; i; i=i >> 1) if (i & f ) cout<<" true 1 "; else cout<<" false 0 "; cout<<endl; }
```

Un altro modo di accedere nei flags potrebbe essere in modo indiretto per mezzo di una variabile (long f) la stessa variabile che abbiamo visto su dal comando ios::fmtflags f; anzi che chiamare con cout.setf(ios::nome_flag | ecc..) ; possibile chiamare in causa la variabile (long f) riconosciuta da fmtflags ecco una prototipo con la chiamata diretta tramite la funzione flags(f); cosi abbiamo attivato ugualmente i vari flags, importante la variabile f long e la chiamata tramite cout.flags(f) altrimenti non funziona nulla, controllate con la funzione prima espressa :

```
long f= ios::showpos | ios::showbase | ecc… ; cout.flags(f);
```

ecco il programma che racchiude le 2 forme di attivare i flags, poi il metodo per controllare i stessi flags attivi o meno alcuni saranno attivati all' inizio ma poi saranno disattivati :

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

void leggi_flags ();  int main ()  {

    cout << " Prima di attivare i flag solo 2 sono attivi, poi si \n disattivano perche' se si sceglie un controllo non e'\n possibile scegliere l'altro, come esempio si deve scegliere; \n un numero decimale esadecimale oppure ottale \n\n ";   leggi_flags ();

    long f = ios::showpos | ios::showbase | ios::oct | ios::right;      cout.flags (f);

    cout << "\n\n Dopo di aver attivato i primi 4 flag \n\n";    leggi_flags ();

    cout.setf ( ios::showpoint | ios::fixed);

    cout << "\n\n Dopo di aver attivato tutti 6 flag \n\n";       leggi_flags ();

    cout << "\n\n ByMpt-Zorobabele\n Per uscire [premere un tasto + invio]: ";
char zx; cin>> zx;   }

void leggi_flags ()    { ios::fmtflags f;  long i;    f = (long)cout.flags ();

    for(i = 0x4000; i; i = i >> 1)  if(i & f) cout << " 1 True  ";  else cout << " 0 False ";  cout << " \n"; }
```

Altri 3 comandi o meglio funzioni che servono per la formattazione del video output con width() si specifica un'ampiezza minima di un campo si usa con stream.width(n); lo stream è cout, (n) è il numero dell' ampiezza che si desidera se n è 12 e dobbiamo scrivere un numero 1 il numero 1 rimane nel 12mo carattere, se il numero era 500.000 l' ultimo 0 rimane nel 12mo spazio tutto quello prima del numero 5 rimane spazio vuoto nel caso che il nostro numero supera l' ampiezza il numero non viene troncato, con stream.fill('ch'); possiamo riempire con un carattere lo spazio vuoto cout.fill('*'); nei spazi vuoti creati da width () inseriamo il carattere specificato da ('ch') dove nel nostro caso nell' esempio è l' asterisco * , nel nostro caso il numero 1 con width(12); avremo 11 stellette poi il numero 1 , nel caso il numero era da 5 cifre avremo avuto 7 stellette più il numero stream.precision(n) questo comando Specifica il numero di cifre da visualizzare in un numero a virgola mobile cosi taglia (n) di cifre di solito la precisione standard è impostata a 6 cifre dopo il numero intero cosi di tutto quel numero può essere tagliato, con un numero tagliamo l' eccesso del numero su un numero di 6 cifre 100.203 se impostiamo (3) rimarremo solo con 100 e se il numero è più piccolo (2) avremo un numero di 1e+02 in altre parole , se il numero è fixed il (n) conta (n) dopo i decimali , con lo scientific con (n) viene tagliate anche le cifre significative questi comandi sono validi solo per il primo stream d' output che incontra, pertanto nel prossimo (cout<<) non ha effetto queste istruzioni fill('*'); width(n);precision(n); n sono sempre numeri interi

15) I Manipolatori

Ora parliamo dei manipolatori sono altre funzioni, queste serve per formattare in un' altro modo lo stream di I/O molti manipolatori sono uguali alle funzioni che abbiamo visto prima, essendo anche loro specifici per attivare o meno i flags ecco la lista di questi manipolatori I e O indica l' utilizzo se è per cout O o cin I o entrambi IO per utilizzare questi manipolatori serve inserire #include<iomanip> :

boolalpha - nobooalpha per IO attiva disattiva il flag true e false , dec hex oct per IO attiva il flag decimale o esadecimale o ottale, endl per O carattere di fine riga e svuota lo stream, ends per O un carattere nullo, fixed scientific per O attiva la modalità standard dei numeri o scientifici , flush O svuotamento dello stream , internal left right O attiva il flag internal, left , right, dove left e right inserisce caratteri di riempimento left riempie la sinistra , right riempie la destra e internal riempie il centro del campo noshobase - shobase O inserisce i prefissi per i numeri ad esempio se si utilizza un numero esadecimale prima del numero inserisce 0x la base esadecimale , noshowpoit - showpoint O inserisce in modo non condizionale un punto decimale in un campo a virgola mobile ovvero tratta i numeri in diversi modi 1 quintale è 100 chili ma è anche 0.1 tonnellate , sono anche 1000 ettogrammi , noshowpos - showpos O inserisce in un campo positivo il segno +, noskipws - skippws O elimina i spazi vuoti iniziali , se skipws() è uguale a 0 i spazi vuoti non vengono eliminati , nouppercase - uppercase O cambia le scritte minuscolo in maiuscolo n uppercase N, setiosflags - resetioflags(f) IO avvia o azzera i flags specificati in (f), setbase () O imposta la base numerica se i numeri sono dec, hex o oct, setfill() O imposta il carattere di riempimento , setprecision() O imposta il numero di cifre di precisione per i valori in virgola mobile , setw() O imposta l' ampiezza del campo , unitbuf O scarica l' output dopo ogni inserimento, ws I salta i spazi vuoti iniziali

Ora vediamo l' utilizzo di queste funzioni preparo un programma uguale scritto con il sistema visto qui e quello visto con l' attivazione dei flags con il primo sistema le differenze sono descritte evidenziate tra il giallo e il verde si vede che con l' utilizzo dei manipolatori si ha un codice più compatto con un solo stream posso inserire più ordini al compilatori se qualcuno utilizza questo programma deve inserire include e using un comando per riga il resto può rimanere tutto su una riga il compilatore non ha problemi :

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

int main () {       cout << " attivazione dei flag  nuovo stile : \n\n";

        cout << " scrivo il numero 125 in esadecimale: " << hex << 125 <<endl;

        cout << "\n\n scrivo il numero 12345.123  numero anticipato da setfill('x')\n abilitiamo 10 spazi vuoti da setw(10): \n\n ";          cout << setfill ('x') << setw (10) << 12345.123;

        cout << "\n\n Stesso lavoro ma attivo i flag nell'altro modo con ios:: -:\n\n";
```

```
cout << " scrivo il numero 125 in esadecimale: " ;   cout.setf
(ios::hex,ios::basefield);

        cout << 125 << endl;

cout << "\n\n scrivo il numero 12345.123  numero anticipato da setfill('x')\n
abilitiamo 10 spazi vuoti da setw(10): \n\n ";          cout.fill ('x');  cout.width (10);
cout << 12345.123;

cout << "\n\n\nByMpt-Zorobabele \n Per uscire premere un tasto + invio: ";
char zx; cin >> zx;  }
```

Il manipolatore setiosflags() , resetioflags() attiva o meno i manipolatori , setiosflag() effettua la stessa azione della funzione setf() , mentre il manipolatore boolalpha invece di portare un numero diverso da 0 =positivo, o uguale a 0 = negativo invece il manipolatore usa il true e il false vediamo l' uso di questi 2 manipolatori per usare setiosflags ricordiamo di inserire l' inclusione di <iomanip> inserire questo pezzo di programma nel programma scritto prima inserirlo al di sopra della chiusura ecco il listato da copiare :

```
cout << setiosflags (ios::showpos | ios::showbase);

cout <<" decimale = "<< 12345 << "\n esadecimale = " << hex << 12345
<<"\n octale = "<<oct<<12345;      cout << " \n\n uso di boolalpha ... \n\n ";
bool b = true;

cout << b << " <- prima di boolalpha..  \n dopo di boolalpha -> " <<
boolalpha << b << endl;

cout << "\n inserire un valore booleano [true/false]: ";     cin >> boolalpha
>> b;

if(b == true)  cout << "\n questo e' il valore immesso: " << b << " = 1\n ";

else   cout << "\n questo e' il valore immesso: " << b << " = 0\n ";
```

Abbiamo detto che lo stream è la funzione di chiamata cout << o cin>> proprio questi operatori (<< inseritore) e (>> estrattore) , questi operatori possiamo utilizzarli per leggere i dati tramite l' overloading, per creare una funzione per l' inseritore o l' estrattore si cambia solo la parola ostream in istream e invece di usare << si usa >> nome_classe è il nome della classe che usiamo oggetto è l' oggetto per accedere i membri della classe (o.x) in input questo oggetto va richiamato per la lettura della memoria con &o pertanto avremo nome_classe &oggetto vediamo la sua sintassi è dove usiamo o o i per ostream o istream dentro la funzione se si usa ostream si usa l' inseritore stream << mentre se si usa l' estrattore istream si usa stream>> importantissimo la raccolta dei dati il return stream restituisce tutti i dati della funzione della classe i/ostream normalmente questa funzione viene inserita al di

fuori della classe tutti i membri della classe devono essere pubblici, oppure viene dichiarata friend in modo che può accedere i membri della classe cosi il prototipo della funzione può essere inserita dentro la classe ecco i 2 tipi di funzione per ostream e per istream mentre il corpo della funzione è uguale cambia solo l' uso dello stream (<<) o (>>) :

friend ostream &operator<<(ostream &stream, nome_classe oggetto)

friend istream &operator<<(i stream &stream, nome_classe &oggetto)

{ corpo della funzione inseritore stream << (come cout) o estrattore stream >>(come cin) return stream ;}

Effettivamente questa nuova funzione essendo un overloading degli operatori o<< i>> non possono essere membri della classe su cui lavora nome_classe oggetto , ma se viene dichiarata friend amica allora deve essere inserita dentro la stessa classe

16) Operazioni input e output

Nel mio testo si parla ancora di stream per aprire i file e tanto altro, nel nostro compilatore alcuni comandi non sono più gestiti e danno errore di compilazione , ecco un elenco di istruzioni che non vengono più impiegate dal nostro compilatore, https://msdn.microsoft.com/library/windows/apps/jj606124.aspx dove non possiamo utilizzare altri comandi alternativi, cosi alcuni capitoli del libro non saranno presi in considerazione come esempi , nello stesso modo vedremo tutti i comandi o funzioni in uso

Per eseguire l' operazioni di Input e Output si utilizza il file Header < fstream > e queste classi derivano dalla classe padre IOS come visto i stream sono 3 input, output, input/output dove ogni stream va dichiarato con la sua classe , dove input ifstream, output ostream , fstream serve per le operazioni di input/output queste classi serve per l' apertura e la chiusura dei file , se desideriamo lavorare in un file con operazioni su descritte possiamo fare :

ifstream in; // input ofstream out; // output fstream io; // input/output

Ora vediamo open() che è una funzione dei sopracitati stream dove il suo prototipo è :

void ifstream::open(const char* nomefile, ios::openmode modo ios::in);

void ofstream::open(const char* nomefile, ios::openmode modo ios::out | ios::trunc);

void fstream::open(const char* nomefile, ios::openmode modo ios::in | ios::out);

Il nomefile è il nome che useremo per il file che utilizzeremo, è possibile includere anche il percorso di tale file , il modo è come verrà aperto il file e le modalità sono valori definiti da openmode attraverso la classe ios_base i modi sono:

ios::app ios::ate ios::binary ios::in ios::out ios::trunc

Per usare o combinare più di queste classi si utilizza l' operatore OR | , vediamo cosa serve ogni modalità che abbiamo citato : ios::app = append usato solo da output , l' output viene aggiunto alla fine del file. ios::ate usato per operazioni di input e output posiziona il puntatore alla fine del file aperto . ios::binary apre file in modalità binaria se il file viene aperto in modalità testo può avvenire traduzioni dei caratteri come il ritorno fine linea o fine riga , in modalità binaria non viene eseguite nessuna traduzione , i file sia binari che di testo possono essere aperti sia in un modo che nell' altro l' unica differenza è la traduzione dei caratteri. ios::in usato per operazioni d' input dati per operazioni d'input . ios::out usato per operazioni d' output , dati per operazioni d'output . ios::trunc , con questo comando si tronca a zero i file preesistenti con la conseguenza della distruzione del contenuto del file precedente

Per aprire i file con i stream la prima cosa da è controllare che il file stesso sia aperto con successo per far questo si usa una funzione booleanea is_open utilizzando , se il file è aperto ci restituisce true altrimenti false :

if (!nomefile.is_open()){ cout << " file non aperto \n ";

per chiudere lo stesso file prima aperto si usa la funzione close() :
nomefile.close;

Per leggere e scrivere dati non formattati è possibile usare le funzioni che legge e scrive un carattere get() legge e put() scrive e questi file devono essere aperti in modalità binaria altro modo per leggere i dati binari sono le funzioni read () legge e write() scrive queste funzioni legge e scrive dei buffer di caratteri

La funzione getline () anzi che leggere un buffer di caratteri legge solo una linea di caratteri che è il buffer ma la differenza tra la funzione get e getline , getline legge e elimina il delimitatore dello stream d' input

Nella lettura di un file si deve vedere la fine dello stesso file per questo si utilizza la funzione eof() il prototipo è

bool eof(); questa funzione restituisce un valore booleano true quando raggiunge la fine del file , caso contrario rimane con false . Un'altra funzione membro è ignore() serve di leggere e ignorare num di caratteri presenti nello stream d' input (il valore normale rimane 1) con delim è il carattere che permette di usare EOF :

istream &ignore(streamsize num=1 , int_type delim= EOF);

La funzione peek() permette di conoscere il prossimo carattere presente nello stream , la funzione putback() inserisce di nuovo l' ultimo carattere che aveva letto . La funzione flush() questa funzione serve in output per far scrivere o scaricare il buffer , di norma il buffer si scarica quando la sua dimensione è piena con la possibilità di perdere i dati in caso di precarietà del sistema usando ostream &flush(); possiamo scaricare prima il buffer e mettere in salvo i nostri dati lo svotamento del buffer può essere fatto con la chiusura del file e questo sarebbe il lavoro naturale come con la chiusura normale dello stesso programma

Altre funzioni che lavorano con i stream sono istream seekg() ; e ostream seekp (); queste funzioni permette di accedere a un file in modo non sequenziale …. Ci sono altre funzioni da vedere , come già detto nel nostro nuovo compilatore non funziona

Ciao grazie da ByMpt-Zorobabele

Testo PDF aggiornato con questo ultimo capitolo inserito lo trovate :

http://www.freeforumzone.com/d/11266858/Elenco-figure-dei-comandi-della-programmazione-C-TESTO-PDF/discussione.aspx

Anche qui di queste discussioni che inserisco faccio un testo PDF da poter scaricare tutto il mio
materiale è da utilizzare come desiderate per il testo PDF vedi cartella dedicata , ogni volta cambia ubicazione del testo allora cambio il file e il testo lo inserisco dentro la cartella < file aggiornato ad oggi >

http://www.freeforumzone.com/d/11266858/Elenco-figure-dei-comandi-della-programmazione-C-TESTO-PDF/discussione.aspx