19. L'istruzione iterativa *while-do*
Sintassi: *while (exp) istruzione;*
dove *exp* è un'espressione Booleana e *istruzione* può essere un'istruzione complessa.

L'esecuzione dell'istruzione *while* comporta:
- il calcolo dell'espressione *exp*
- se *exp* è vera, l'esecuzione di istruzione e la ripetizione dell'esecuzione dell'istruzione *while*

Se il risultato della valutazione di *exp* è false (cioè 0) l'istruzione non viene eseguita. In caso contrario viene eseguita ogni qualvolta *exp* viene valutata *true*.
L'istruzione potrebbe non venire mai eseguita oppure potrebbe venire eseguita all'infinito (creano dei loop) se *exp* è sempre vera.

I loop infiniti sono da evitare, ma possono essere utili nella realizzazione di alcuni servizi.

! *exp* tipicamente contiene almeno una variabile (variabile di controllo del ciclo), che viene modificata in istruzione per far convergere *exp* verso uno stato in cui diventi falsa. Si può mettere qualsiasi espressione al posto di *exp* (posso scrivere *true* e in questo modo l'istruzione del ciclo verrà eseguita all'infinito).

19.1 Qualche esempio di istruzione iterativa *while-do*.
- Eseguire l'istruzione 10 volte (avendo inizializzato *i=0,* oppure con *i=1* si mette come *exp* *i<=10*)
```
while (i<10){
   istruzione;
     i++;
  }
```
- Esegue l'istruzione all'infinito
```
while (true)
   istruzione;
```

- Esegue l'istruzione all'infinito (in realtà non è proprio all'infinito ma finché non raggiunge l'overflow; dopo aver inizializzato *i=1)*
```
while (i>0) {
   istruzione;
   i++;
}
```

19.2 L'istruzione iterativa *do-while*
Sintassi: *do { istruzioni; } while (exp);*
dove *exp* è un'espressione Booleana e *istruzione* può essere un'istruzione complessa.

L'esecuzione dell'istruzione *do* comporta
- l'esecuzione di istruzione
- il calcolo dell'espressione *exp*
- se *exp* è vera, la ripetizione dell'esecuzione dell'istruzione *do*

Un'importante differenza rispetto al ciclo *while* è che in quest'iterazione l'istruzione viene eseguita almeno una volta.

19.2.1 Un'importante uso del ciclo *do-while*: i menu'
int main (){

char scelta;

do {

istruzioni;

cout << "Vuoi rieseguire il programma? Premi 's' " << endl;
cin >> scelta;

} while (scelta == 's');

return 0;
}

Mettendo tutto ciò che fa il nostro programma all'interno di un ciclo *do-while* riusciamo successivamente a permettere all'utente di rieseguirlo quante volte vuole.

! È importante che se il nostro programma stampa ad esempio un numero di iterazioni prima di rieseguirlo esse vengano rinizializzate al numero standard.

19.2.2 Reinserimento di valori sbagliati

do{
cout << "Inserisci un numero che sia maggiore di 0";
cin >> x;
(x<=0) ? cout << "Errore. Numero minore di 0";
} while (x<=0);

Se l'utente dovesse sbagliare e inserire un valore minore di 0, non c'è problema! Basterà dargli la possibilità di reinserire il valore con un ciclo *do-while*.

19.3 L'istruzione iterativa *for*

Sintassi: *for (init; exp; agg) istruzione;*
dove:

- *init* è un'istruzione di inizializzazione delle variabili di controllo
- *exp* è un'espressione Booleana
- *istruzione* può essere un'istruzione complessa
- *agg* è un'istruzione di aggiornamento delle variabili di controllo

L'esecuzione dell'istruzione *for* comporta:

- l'esecuzione di *init*
- il calcolo dell'espressione *exp*
- se *exp* è vera, viene eseguita istruzione, poi *agg*, e si ricomincia dal passo 2.

! L'iterazione *for c*onsente di separare le istruzioni di controllo del ciclo e concentrarle tutte in un'unica riga, garantendo una miglior praticità e leggibilità del codice.

19.3.1 Esempi del ciclo *for*

1.Si possono definire variabili di controllo interne al ciclo:
for (int i = 0; i < MAXDIM; i++) //si svolge il ciclo MAXDIM volte e *i* occorre solo qua

2.Si possono anche non mettere *init* e *agg* (che chiaramente devono essere rispettivamente specificate prima e dentro al ciclo)
for (; i < 10;)

3.Si può fare un ciclo dentro a un ciclo (le variabili *i* e *j* devono essere definite in precedenza)

```
for (i=1;i<=12;i++){
        for (j=1; j<=10; j++){
                cout << i*j;
        }
}
```

Il programma stamperà:
1*1, 1*2, … , 1*12
Poi aggiornerà le variabili:
2*1, 2*2, … , 2*12
…

4. L'utente può volendo inserire un valore e il nostro ciclo svolgerà le istruzioni n volte quante lo ha inserito l'utente. Ad esempio chiedendo il numero massimo di iterazioni all'utente, noi faremo l'operazione seguente per n volte.

```
cout << "Numero massimo di iterazioni?: ";
cin >> N;

for (int i=0;i<=N;i++)
   somma+=1.0/(pow(2.0,i));          //tende a zero
```

5. Condizione di uscita dal ciclo quando si raggiunge una data precisione di ciò che viene calcolato.

```
cout << "Valore della precisione richiesta?: ";
cin >> Eps;
double  ultimasomma=0.0, somma=1.0;
 for (i=1;fabs((somma-ultimasomma))>=fabs(Eps*somma);i++)
```

```
 {
  ultimasomma=somma;
  somma+=1.0/(pow(2.0,i));
 }
 cout << "La somma e' = " << somma << endl;
 cout << "calcolata  dopo " << i-1 << " passi\n";
```

fabs((somma-ultimasomma))>=fabs(Eps*somma): con questa espressione si calcola il valore assoluto della differenza tra somma e ultima somma e quando questo valore è minore del prodotto tra la precisione richiesta e la somma stessa si esce dal ciclo.

6. Doppia inizializzazione e doppia condizione di uscita.
Nei cicli come *init*, *exp* e *agg* si possono mettere delle istruzioni complesse.

```
for (i=0,somma=0.0,ultimasomma= -1.0;                    //init
     fabs((somma-ultimasomma))>=fabs(Eps*somma) && i<=N;  //exp
     i++)                                                //agg
```

19.3.2 Equivalenza dei cicli

I tre cicli sopra descritti sono equivalenti. Uno può essere usato al posto di un altro e viceversa.

for (init; exp; agg) istruzione;

equivale a:

```
init;
while ( exp ){
istruzione;
agg; };
```

20. Iterazioni e cicli importanti

Come sommo n numeri inseriti dall'utente senza prima chiedergli quanti numeri vuole inserire? Di norma bisognerebbe chiedere quanti numeri l'utente vuole inserire per poi creare un ciclo che gli permetta di inserire *n* numeri e poi ne faccia la somma.

Ricordiamo che come condizione *exp* può essere scritta qualsiasi espressione anche questa:
cin >> x;
Definito il tipo della variabile *x* l'espressione risulterà *true* (e quindi verrà eseguito il ciclo) se l'utente ha inserito un tipo conforme con *x*. In caso contrario lo stream andrà in errore e non si eseguirà il ciclo.

! È importante ricordare che si esce dal ciclo grazie allo stream che va in stato di errore. Bisogna quindi effettuare l'operazione di pulizia dello stream prima di fare altre operazioni di IO.

Esempio

```
cout << "Inserisci quanti numeri vuoi. Poi premi un carattere non numerico qualsiasi per terminare";
while (cin >> x) {
   sum = sum + x;
  }

cin.clear();      //due costrutti importanti per pulire lo stream
cin.ignore();
```

Questo esempio equivale a scrivere questo:

```
 cin >> x;
while (!cin.fail()) {
   sum = sum + x;
   cin >> x;
  }
```

cin.fail() verifica se lo stream è in stato d'errore (ritornando 1 se c'è un'errore, 0 in caso contrario). Se viene inserito un numero *cin.fail()* restituirà *0* perché lo stream non è in errore. Ricordando che *0 = false* in questo caso il ciclo non verrebbe eseguito. Per questo bisogna mettere il punto esclamativo *!* davanti a *cin.fail()*. Quando verrà inserito un carattere non numerico *exp* varrà *0* e quindi si uscirà dal ciclo.

21. Gli invarianti di un ciclo (Loop invariant)
Tecnica per la verifica di correttezza dei cicli (proprietà P).

Idea: suddividere la proprietà desiderata P (che garantisce la correttezza del ciclo) in una sequenza di affermazioni P0, P1, ... Pn, in modo che:

1. P0 sia vera immediatamente prima che il ciclo inizi (dopo l'inizializzazione!).
2. per ogni indice di ciclo i $\in$ {1, ..., n}: se Pi−1 è vera prima dell'inizio del ciclo i-esimo (ed è verificata la condizione di permanenza del ciclo), allora Pi è vera alla fine del ciclo i-esimo (e quindi immediatamente prima dell'inizio del ciclo (i + 1)-esimo). Tipicamente è il passo più critico.
3. Alla fine dell'ultimo ciclo (n-esimo), Pn (e la negazione della condizione di permanenza) implica la proprietà P.

Le Pi a volte sono ovvie, a volte molto complesse. Talvolta sono necessarie delle variabili ausiliarie addizionali e si adottano convenzioni per gestire il caso i = 0: (la somma di 0 elementi è 0, il prodotto di 0 elementi è 1, …).

21.1 Esempio fattoriale
i = 1; fact = 1;
*while (i<=n) { fact *= i; i++; }*

Proprietà P: dopo il ciclo, *fact* vale il prodotto dei primi n numeri.
Invariante Pi : *fact* vale il prodotto dei primi i numeri.

1. P0 è vera immediatamente prima che il ciclo inizia: *fact* vale il prodotto dei primi 0 numeri, cioè 1.
2. Prima dell'i-esimo ciclo *fact* vale il prodotto dei primi i-1 numeri. Ad esempio prima del terzo ciclo, *fact* vale il prodotto dei primi 2 numeri. PRIMO CICLO: fact=1, SECONDO: fact=2 (alla fine del secondo *i=3*), TERZO: *fact=6 (i=4)*. Dopo l'i-esimo ciclo (*i* è incrementato di 1) *fact* vale il prodotto dei primi *i* numeri.
3. Alla fine dell'ultimo ciclo (n-esimo), Pn (più la negazione della condizione di permanenza del ciclo) implica la proprietà P.

21.2 Esempio divisibilità per 2 (il programma serve a dire quante volte un numero è divisibile per due)
ndiv2=0; tmp=num; // "tmp" ausiliaria
while (tmp%2 == 0) { ndiv2++; tmp/=2; }

Proprietà P: dopo il ciclo, *tmp%2!=0* e *tmp*(2^ndiv2)==num* (cioè *tmp*2(tanti due quante sono le volte che tmp è stato diviso per 2)==num*).
Invariante Pi : *tmp*(2^ndiv2)==num.*

1. Prima del ciclo, *tmp*(2^0)==num*
2. Prima dell'i-esimo ciclo *tmp* è divisibile per due e *tmp*(2^ndiv2)==num.* Dopo l'i-esimo ciclo *tmp*(2^ndiv2)==num* (infatti *(tmp/2)*(2^(ndiv2+1))==num*).
3. Alla fine dell'ultimo ciclo (n-esimo), Pn (più la negazione della condizione del ciclo) implica la proprietà P : *tmp*(2^ndiv2)==num* e *tmp%2!=0*.

22. Le istruzioni di salto
Sono di 4 tipi: ***break, continue, return*** e **goto**.
Vanno sempre se possibile evitate (si possono evitare modificando le condizioni). Soprattutto **goto: <u>mai usare goto!!!!</u>**

22.1 L'istruzione *break*
L'istruzione *break* termina direttamente tutto il ciclo, indipendentemente dalla condizione di uscita.

```
while (...) {
istr;
break;
istruzion2;
}
istruzione3;    //a causa del break tutto ciò che viene dopo nel ciclo non viene eseguito. Si riparte da
                istruzione3
```

22.2 L'istruzione *continue*
L'istruzione *continue* termina il ciclo attualmente in esecuzione e passa al successivo. Significa che si ritorna a verificare la condizione *exp* del ciclo saltando tutte le operazioni tra *continue* e la fine del ciclo.
!Non si esce dal ciclo ma si torna all'inizio, saltando tutto ciò che c'è dopo.

Nel caso di ciclo *for* viene saltata l'istruzione di aggiornamento.

L'uso di un continue può essere evitato con un *if*.

```
while (…){
istr;
continue;       //non viene eseguita istruzione2 e si torna alla condizione exp del ciclo
istruzione2;
}
istruzione3;
```

22.3 L'istruzione *return*
L'istruzione *return* termina direttamente il ciclo (e l'intera funzione).

Se siamo nella funzione *main* e usiamo un *return* all'interno di un ciclo. La funzione *main* finirà senza eseguire più nulla del nostro programma dopo il *return*.

```
int main () {
istr;

while (...) {
istruzione2;
return 0;       //l'esecuzione del programma si ferma a istruzione2. Tutto ciò che viene dopo non
                viene eseguito
istruzione3;
}
…
istruzione4;
return 0;       //fine main
}
```

22.4 L’istruzione *goto*

goto LABEL_EX1; //da qua andiamo a LABEL EX1

cout << "Non verrò mai visualizzato" << endl;

LABEL_EX1: //si esegue ciò che dice la LABEL EX1 e poi si riprende linearmente
cout << "Sono dopo la label" << endl;