Seminario di C

Cosa e' il C

Il C è un linguaggio di programmazione… Come il Pascal, Visual Basic, ecc…

Pregi e Difetti

Il C viene preferito dai professionisti perché permette di lavorare sia ad alto che a basso livello.
Ad alto livello significa che possiamo usare delle funzioni senza dover necessariamente pensare a come sono fatte.
A basso livello invece significa che ci permette di lavorare molto vicino al linguaggio macchina vero e proprio, il che porta ad un indubbio incremento della velocità di esecuzione.
D'altro canto il fatto che si possa lavorare a basso livello permette anche di fare cose che non dovrebbero essere fatte. Come ad esempio scrivere in zone di memoria che non ci appartengono mandando in questo modo in crash il sistema o il PC.

Cosa mi serve per fare un programma in C ?

Un programma in C non è altro che uno o più file di testo, che hanno normalmente estensione C, CPP, H, HPP (torneremo più avanti sul significato di queste estensioni). Quindi per fare un programma serve un normalissimo **editor di testi** (Ad esempio notepad, pico, joe, ecc..).

Per fare un programma in C è necessario un **compilatore**, cioè un programma che "traduca" le linee di codice scritte con l'editor di testo nel linguaggio interpretabile dalla macchina.
Se avete una macchina linux o unix è molto probabile che abbiate almeno il **gcc**. Cioè il compilatore C++ sotto licenza GNU.

Il primo programma in C

```
#include <stdio.h>

int main()
{
        printf("Ciao mondo!!!\n");
}
```

Questo esempio è quello usato sempre come primo approccio al linguaggio, questo perché contiene al suo interno tutte le caratteristiche fondamentali del linguaggio.
Analizziamolo insieme…

#include <stdio.h>

Questa è detta direttiva di preprocessore, ed indica al compilatore che il file "stdio.h" va letto ed interpretato prima della compilazione del "modulo". Un modulo è un singolo sorgente o file di un programma C. In questo caso si tratta di una libreria di sistema che contiene diverse funzioni.

int main()

Come tutto ciò che si fa in C main è una funzione. In realtà è qualcosa di più è la funzione più importante. E' il punto di accesso al programma, cioè la prima funzione che viene eseguita.

{ …. }

Tra parentesi graffe si trova un blocco di codice, in questo caso il codice della funzione. E' obbligatorio dopo la definizione della funzione mettere le parentesi graffe.

printf("Ciao mondo!!!\n");

Questa è una chiamata a funzione, la funzione si chiama **printf**, e il parametro della funzione è la scritta **Ciao mondo!!!.** Il carattere **\n** è un carattere speciale che dice che a quel punto della stringa il cursore deve andare a capo.

Da notare che ogni riga in C deve essere chiusa con un punto e virgola (**;**) perché l'interprete ingora gli spazi, le tabulazioni e gli a capo e considera una riga fino al punto e virgola.
Per intenderci sarebbe stato uguale scrivere

```
printf(
"
Ciao mon
do!!!\n"
)
;
```

anche se sarebbe stato molto meno leggibile.

Compilazione

Ora che abbiamo fatto il nostro programma dobbiamo imparare a compilarlo. Per fa questo abbiamo bisogno del compilatore. Nel nostro caso farò gli esempi con il compilatore gcc perché è gratuito e disponibile in rete per tutta una serie di sistemi operativi, ma per fare le prove potete utilizzarne uno qualunque (Borland, Microsozz, ecc..).
NOTA : Una cosa importante di cui tenere conto è il fatto che **il sorgente va compilato sulla macchina** o almeno sul sistema operativo **su cui dovrà "girare"**. Questo perché il codice non viene "interpretato" come nel caso di perl o di altri linguaggi di scripting, ma viene convertito nel linguaggio macchina per essere eseguito.

Salvando il nostro sorgente con il nome **primo.c** la linea di comando da utilizzare è :

gcc primo.c –o primo

L'opzione **–o** indica al compilatore il nome del file eseguibile che dovrà essere creato. Nel caso del gcc altrimenti verrà creato **a.out**.

A questo punto se tutto è scritto correttamente abbiamo appena creato il nostro primo programma che normalmente è già eseguibile.

I tipi di dati

In C le variabili che si utilizzeranno nel programma vanno obbligatoriamente definite con nome e tipo. Ad esempio :

int i;

indica che voglio dichiarare una variabile che si chiama **i** e che è di tipo intero, ovvero un numero reale intero.

I tipi di dati più comuni sono :

short (intero corto)
long (intero lungo)
float (reale corto)
double (reale lungo)
char (carattere) ad es. ‘a’

Un esempio di dichiarazione può essere:

short s;
s = 10000

long l;
l = 450000;

float f;
f = 12.34;

double d;
d = 13.3455

char c;
c = ‘a’;

Di solito un **float** rappresenta un numero a **32 bit** e un **double** rappresenta un numero a **64 bit**. Questo implica che un double ha necessariamente una precisione superiore ad un float ma occupa più spazio in memoria. Può anche capitare che il float sia a 16 bit e il long a 32. Questo dipendentemente dal tipo di macchina su cui stiamo compilando.
La stessa differenza la riscontriamo tra il tipo **short** che è a **16 bit** e il tipo **long** che è a **32 bit** (anche questi possono essere dipendenti dal tipo di macchina).

Operazioni tra i dati

Ora che conosciamo i dati che possiamo utilizzare vediamo quali operazioni possiamo farci.

*** => moltiplicazione**
4*5=20 Moltiplica i numeri inseriti
+ => addizione
2+10=12 Somma i numeri inseriti
- => sottrazione
3-2=1 Sottrae e numeri inseriti
/ => divisione
5/4=1 Divide e come risultato abbiamo il n° di volte divisibili senza resto
% => divisione con modulo
10%7=3 Divide e come risultato abbiamo il resto
++ =>incremento
a++; Assegna e Incrementa a di uno
++a; Incrementa a di uno e Assegna
-- =>decremento
a--; Assegna e Decrementa a di uno
-- a; Decrementa a di uno e Assegna

Vedremo più avanti come vengono trattate le differenze di tipi durante le operazioni. Per ora basta sapere che normalmente viene convertito il numero a minore precisione in uno a maggiore precisione.

Input/Output

A questo punto abbiamo bisogno di poter comunicare con l'utente.
Per scrivere qualcosa a video la funzione più comunemente utilizzata è **printf** che abbiamo già visto nell'esempio.

La sintassi è : printf("**formato**", **parametro1**, **parametro2**, **parametro3**,ecc.);
Il testo scritto in **formato** indica alla funzione ciò che deve cercare nei parametri successivi. I parametri da sostituire iniziano sempre per il simbolo della percentuale (**%**).
Ad esempio per stampare un intero si usa **%d**.

Ad esempio la funzione : **printf("il numero da scrivere è %d\n",12);**
darà come risultato : **Il numero da scrivere è 12**.
In pratica ogni volta che viene incontrato un parametro con il % viene cercato il parametro corrispondente. Se ci sono più parametri l'ordine con cui vengono incontrati i % deve essere lo stesso dei parametri.

Ad esempio :
printf("Sono le ore %d e %d minuti\n", 13, 26);
dà come risultato :
Sono le ore 13 e 26 minuti.

I formati che possono essere usati nella printf sono :

Table: Printf/scanf format characters

Formato spec. (%)	**Tipo**	**Risultato**
c	Char	singolo carattere
i,d	Int	numero decimale
o	Int	number ottale
x,X	Int	numbero hexadecimal notatione maiuscolo/minuscolo
u	Int	unsigned int
s	char *	stampa una stringa
f	double/float	formato -m.ddd...
e,E	double/float	Formato Scientifico -1.23e002
g,G	double/float	e or f quello che è più compatto
%	-	stampa il carattere %

Ciò vuol dire che per esempio che per stampare un numero in virgola mobile devo digitare :

printf("Il tavolo è lungo %f cm\n", 133.26);

che dà come risultato :

Il tavolo è lungo 133.26 cm

Gli array

Un array non è altro che una sequenza di valori. Ad esempio le temperature degli ultimi giorni, una lista di nomi,ecc.

10	11	12	13	41	0	12	20

Un array in C viene definito con le parentesi quadre [] e il valore tra parentesi indica la dimensione dell'array.

La dichiarazione si effettua ad esempio

int a[10];

E' un'array che può contenere **10** valori. Gli indici dei valori vanno da 0 a 9.
In questo caso per accedere ad un dato valore indico tra parentesi semplicemente l'indice del valore che ci interessa.

a[2]=100;

Assegna al **terzo** elemento dell'array (gli array partono sempre da 0) il valore 100;
Oppure :

printf("L'elemento numero 3 è %d", a[2]);

per stampare il valore.
Nello stesso modo possiamo utilizzare gli array con tutti gli altri tipi di dati (long, short, ecc.).
In realtà un discorso a parte lo merita **l'array di char**. In C non esistono le stringhe, per cui un testo non è altro che un array di caratteri. Parleremo più dettagliatamente della cosa più avanti.

I Puntatori

Un puntatore non è altro che l'indicatore di un indirizzo di memoria. Ogni variabile infatti viene "fisicamente" memorizzata nella memoria del computer.
Il modo canonico per indicare un puntatore in C è il seguente.

int *pi;

che in questo caso indica che la variabile **pi** è un puntatore ad un intero cioè come già detto indica la locazione di memoria della variabile.
Per accedere alla variabile "puntata" dal puntatore si utilizza il carattere asterisco(*) prima del nome. Ad esempio :

****pi =5;***

Le stringhe

L'uso più comune dei puntatori è nella gestione delle stringhe. Come abbiamo già detto le stringhe non esistono nel linguaggio C. Una stringa non è altro che un array di caratteri terminata da un "**\0**".
Cosa significa terminata da un \0 ? Che indipendentemente da quello che può essere la grandezza dell'array di caratteri che contiene la stringa questa viene stampata solo fino a che non viene incontrato in carattere \0 che equivale codice carattere 0.
Ad esempio :

char a[8] = "Ciao";

C	**i**	**a**	**o**	**\0**	**.**	**.**	**.**

Nella dichiarazione precedente i caratteri vengono memorizzati come nella tabella e viene aggiunto il \0 come **terminatore di stringa**.
Per accedere alla stringa a questo punto possiamo utilizzare il puntatore alla stringa. Avendola dichiarata in questo modo la variabile è già un puntatore a char (char *) e quindi posso scrivere:

printf(" La parola memorizzata è %s\n", a);

che produrrà come output :

La parola memorizzata è Ciao

Generalmente però non sappiamo a priori che cosa mettere nella variabile. Per far questo abbiamo bisogno delle funzioni per le stringhe che sono nella libreria <**string.h**>.
Per assegnare una variabile in c è necessario usare la funzione di copia stringa sempre a causa del fatto che non esiste il tipo stringa:

strcpy(destinazione, sorgente);

copia la stringa (d'ora in poi la chiameremo così per praticità) sorgente in quella destinazione.

Lettura dallo standard input

Per comunicare con l'utente è necessario in ogni caso poter leggere dallo standard input (tastiera o file). Per far questo possiamo utilizzare la funzione scanf.

```
int i;
scanf("%d", &i);
```

Legge un dato di tipo intero e lo memorizza nella variabile i.
Vediamo in questo esempio una cosa nuova. Il carattere **&** davanti alla variabile i è una modalità particolare di passare i parametri. Il questo caso significa che della variabile debba essere passato il puntatore alla funzione. Per intenderci in questo caso la dichiarazione della funzione scanf potrebbe essere :

```
void scanf(char *tipo, int *var);
```

I tipi di variabili più comuni che possiamo leggere sono :

Sintassi da utilizzare	Descrizione
%d	Dati di tipo int
%lf	Dati di tipo double
%c	Dati di tipo char
%s	Dati di tipo stringa

Espressione condizionale con il costrutto if

Le espressioni condizionali in C sono molto simili a quelle di altri linguaggi. L'esecuzione condizionata viene fatta con il costrutto if.

```
if (condizione)
        codice1;
else
        codice2;
```

In questo caso se la condizione risulta verificata viene eseguito il codice1 altrimenti viene eseguito il codice2. L'else non è obbligatorio, ma se il codice1 o il codice2 è composto da più di una istruzione è necessario racchiuderlo tra parentesi graffe (**{}**).
Ad esempio con una sola istruzione:

```
if(a < 25)
        a++;
else
        a=0;
```

mentre con più istruzioni :

```
if(b < 25)
{
        a++;
        b- -;
}
else
        a=0;
```

Il codice è ovviamente puramente esplicativo.

Operatori di confronto

Per confrontare le variabili si utilizzano i seguenti operatori di confronto

Tipo di espressione	Descrizione
X= =Y	Testa se il valore di x è uguale a y
X >Y	Testa se x è maggiore di y
X >=Y	Testa se x è maggiore uguale di y
X<Y	Testa se x è minore di y
X<=Y	Testa se x è minore uguale di y
X !=Y	Testa se x è diverso da y

Nota Bene : Il C non riconosce come errore se mettete in un confronto l'operatore = al posto di quello = = per cui se sbagliate e scrivete

if (a =1) invece di **if (a = = 1)**

il programma non eseguirà il confronto ma un assegnamento (a diventa 1) e utilizzerà il valore di a per valutare l'espressione. Essendo a diverso da 0 verrà considerato come **vero** e valutato di conseguenza. In C infatti un valore da valutare viene considerato falso se è 0 e vero se diverso da 0.
Infatti se utilizzo un operatore di confronto per fare una assegnamento il valore memorizzato sarà 0 se l'espressione è falsa e sarà diverso da 0 se l'espressione è vera.
Ad esempio :

```
c= (a>b);
```

Se a è maggiore di b, c varrà 1 (o comunque un valore diverso da zero) mentre in caso contrario varrà 0.

Operatori logici

Nel caso dovessimo valutare più di una condizione abbiamo a disposizione gli operatori logici per collegare le varie condizioni.
I più comuni sono quello di **AND (&&)** e **OR(||)**. Se qualcuno non conosce i principi di logica possiamo brevemente dire che l'AND torna vero **solo se entrambe** le condizioni risultano vere, mentre l'OR torna vero **se almeno una** delle due condizioni risulta vera.
Esiste anche l'operatore **NOT (!)** che nega (cioè se è falso diventa vero e viceversa) il valore prima del quale è messo.
Ad esempio nelle istruzioni:

```
a = 10;
b = 20;
if ((a > 15) && (b > 12))
```

essendo la prima condizione falsa tutta la condizione globale risulterà falsa, mentre se avessimo messo

if ((a > 15) || (b > 12))

essendo la seconda condizione vera la condizione globale risulterà vera.
E nella condizione :

if(!(a>15))

se a è maggiore di 15 risulta falso (perché è negato) e viceversa.

Conversione di tipo (cast)

Quando in una espressione sono utilizzati tipi di dati differenti, il compilatore, per poter calcolare l'espressione è costretto a trasformare i numeri in modo che siano dello stesso tipo. Questo viene fatto trasformando sempre il numero che occupa meno memoria in quello che occupa più memoria.
Ad esempio :

```
int i;
double d;
a = d * i;
```

In questa espressione il compilatore converte l'int in double e poi esegue l'operazione.
Possiamo forzare questa conversione facendo un "**cast**". Per fare un cast è sufficiente mettere il tipo che vogliamo usare tra parentesi tonde prima dell'operazione. Ad esempio :

```
d = (float)i / 2;
```

In questo caso il compilatore forza la conversione della variabile i(intera) in un float e quindi fa una operazione tra float, mentre senza il cast avrebbe fatto una operazione tra interi e poi assegnato il risultato (senza decimali) al double.

Cicli

A volte possiamo avere bisogno di ripetere più volte la stessa operazione. Per fare questo ci serviremo delle istruzioni per fare dei cicli.
I tre tipi di cicli che possiamo utilizzare in C sono :

- **for**
- **while**
- **do…while**

Ciclo for

Il **ciclo for** è il più comune e ci permette di ripetere ciò che segue finché non si verifica una determinata condizione. La sintassi generica è :

for(a ; b ;c)

a = impostazione iniziale variabili
b = condizione da verificare
c = operazioni da effettuare al termine del ciclo

Con un esempio pratico possiamo dire che l'istruzione

```
for (i = 0; i < 100; i++)
        codice;
```

Inizializzerà la variabile i a 0 ed eseguirà il codice che segue l'istruzione finché i sarà minore di 100 e in caso che la condizione non fosse verificata incrementerà di 1 la variabile i e ripeterà il ciclo di nuovo.

Ciclo while

La struttura del ciclo while invece è :

```
while(condizione)
{
        codice;
}
```

In questo caso viene valutata la condizione e finché rimane valida viene eseguito il codice. Siccome la condizione deve variare all'interno del ciclo nella maggior parte dei casi avremo più istruzioni che andranno quindi racchiuse all'interno delle parentesi graffe.
Ad esempio :

```
i = 0;
while( i < 0 || i > 10)
{
        printf("Inserisci un numero tra 0 e 10\n");
        scanf("%d", &i);
}
```

In questo caso il ciclo viene ripetuto finché il valore inserito dall'utente non è un numero compreso tra 0 e 10.

Ciclo do … while

Il ciclo do… while è essenzialmente identico come procedura al precedente, ma controlla il verificarsi della condizione solo al termine del codice.
Il **do** identifica l'inizio del blocco di codice da ripetere e il **while** la fine. Tra di loro è necessario mettere le parentesi graffe.

```
do
{
        ….
}while(a <10);
```

I commenti

I commenti sono una cosa a dir poco importante quando fate un programma. Essi sono pezzi di testo che non vengono tenuti in considerazione dal compilatore. Essenzialmente servono per rendere più leggibile e comprensibile il programma stesso.

In C un commento viene racchiuso tra la sequenza slash - asterisco (/*) e asterisco - slash (*/).
Tutto il testo contenuto all'interno viene come abbiamo già detto ignorato.
Facciamo un esempio :

```
/*
Primo programma in C

E' un programma di esempio per il seminario di C
*/

#include <stdio.h>
.
```

In questo caso abbiamo usato il commento per indicare la descrizione (Primo programma in C) e cosa fa il programma (programma di esempio).
Alcuni tipi di compilatore permettono di fare i "**nested comments**", cioè i commenti nidificati.
Ad esempio :

```
/* esempio di commento

/*
Commento nidificato
*/

*/
```

Se il compilatore è predisposto tiene conto in pratica del numero di commenti aperti e considera il commento fino alla sua chiusura. Questo può capitare se "***commentiamo***" un pezzo di sorgente all'interno del quale sono presenti già dei commenti.
Siccome non tutti i compilatori lo possono fare cerchiamo di evitarlo.

Strutture

Le strutture sono un modo di raggruppare logicamente più dati in un unico oggetto. In C esse sono identificate dalla parola chiave **struct**.
Poniamo ad esempio che volessimo creare una struttura che contenesse i dati identificativi di un punto. Un punto è identificato dalla sua posizione x e dalla sua posizione y.
La struttura corrispondente sarà :

```
struct punto {
        int x;
        int y;
};
```

A questo punto abbiamo un nuovo tipo di variabile detto definito dall'utente. Per dichiararlo dobbiamo scrivere :

struct punto p;

A questo punto vorremmo accedere ai dati contenuti nella variabile di tipo punto p. Per far questo si usa l'**operatore punto** (.).

p.x = 15;
p.y = 5;

In pratica con questa sintassi accediamo al dato di tipo intero che si trova all'interno della struttura.

Tipi di dati utente

Ora che abbiamo conosciuto le strutture però ci farebbe comodo poterle usare in modo un po' più pratico. Tramite la parola chiave **typedef** possiamo definire proprio un nuovo tipo di dati.
Se scriviamo ad esempio :

```
typedef double costi;
```

Dopo la dichiarazione potremo usare **costi** come tipo di variabile in una nuova dichiarazione.

```
int rate;
costi costototale;
```

Allo stesso modo potremo usare il typedef, che poi è il modo più utilizzato, per creare un tipo definito con una struttura. Ad esempio :

```
typedef struct punto{
        int x;
        int y;
};

punto p;
```

L'operatore ->

Nel caso in cui al posto di una struttura avessimo un puntatore a struttura l'accesso ai dati della stessa si può fare in due modi. Con l'operatore astrerisco come abbiamo già detto :

```
punto *p;
punto p2;

p = &p2;
(*p).x = 10;
```

o con un nuovo tipo di operatore che è studiato appositamente per questo tipo di accesso.
Ad esempio :

```
p->x = 10;
```

La sintassi potrebbe sembrare inutile, ma quando si hanno puntatori a strutture (che è un caso abbastanza frequente nelle funzioni), è consigliabile e più comodo. Inoltre vedremo un suo utilizzo più massiccio nel C++.

Le direttive di preprocessore

Le direttive di preprocessore, come già accennato sono comandi eseguiti prima della compilazione vera e propria. Tutte le direttive di preprocessore sono precedute dal simbolo cancelletto o diesis (#).

Alcune delle direttive di preprocessore più diffuse sono :

#define
#include
#ifdef -#else #endif
#if
#elif
#undef

In pratica quando inseriamo delle direttive di preprocessore, la compilazione viene eseguita sul sorgente generato dal preprocessore stesso.
Facciamo un esempio pratico.

```
#define PIGRECO    3.1415

int main()
{
        printf("%f", PIGRECO);
}
```

Viene convertito in :

```
.
        printf("%f", 3.1415);
.
```

Il vantaggio rispetto all'utilizzo delle variabili standard sta essenzialmente nella velocità. Il programma non va a cercare in memoria il valore da passare ma usa un valore ben definito come se avessimo scritto il numero stesso nel sorgente.

La direttiva ***#include*** , che abbiamo già incontrato invece inserisce un file direttamente al'interno del sorgente da compilare. Se utilizziamo le virgolette il file verrà cercato nella directory corrente e in quelle della **path,** mentre se scriviamo il nome in mezzo ai simboli minore e maggiore (<>) il file verrà cercato nelle directory specificate nella cartella include (normalmente sono le librerie di sistema).

Direttiva #ifdef

La direttiva ifdef serve per fare la cosiddetta compilazione condizionale. Se la variabile specificata dopo la direttiva è definita .il sorgente fino all'#else o all'#endif (se l'else non è presente) viene compilato, in caso contrario se è definito l'#else viene compilato quello all'interno dell'#else altrimenti viene semplicemente saltato.
In C viene usato ad esempio per fare un sorgente unico per più compilatori o per sistemi operativi diversi. In questo modo chiunque compila il sorgente non ha bisogno di trovarne uno appositamente scritto per lui. Oppure può essere comodo per realizzare delle versioni di debug che includano all'interno del sorgente codice che non dovrà apparire all'utente finale.
Ad esempio :

```
#define DEBUG 1

#ifdef DEBUG
        prinf("debug attivo")
#endif
```

Operazioni sui bit

Gli operatori orientati al bit, bitwise secondo la terminologia anglosassone, sono i seguenti

NOT	~	(unario)
AND	&	(binario)
XOR	^	(binario)
OR	|	(binario)

L'operatore NOT (~) compie il complemento logico bit per bit, quindi se il bit è 0, questo viene alzato, ovvero portato a 1.
L'operatore AND (&) dà come risultato il bit a 1 se il corrispondente bit di entrambi gli operandi è 1, altrimenti il bit risultante è 0.
L'operatore OR (|) dà come risultato il bit a 1 se il corrispondente bit di almeno uno degli operandi è 1, solo quando entrambi i bit sono 0 il bit risultante è 0.

Per avere maggiori chiarimenti potremmo guardare un qualsiasi tabella di corrispondenza logica.

NOT

Bit	Risultato
0	1
1	0

AND

Bit 1	Bit 2	Risultato
0	0	0
0	1	0
1	0	0
1	1	1

OR

Bit 1	Bit 2	Risultato
0	0	0
0	1	1
1	0	1
1	1	1

XOR

Bit 1	Bit 2	Risultato
0	0	0
0	1	1
1	0	1
1	1	0

Un altro operatore che opera sui bit è il cosiddetto bit shifter. a sinistra (<<) o a destra (>>).
In pratica questi operatori fanno spostare tutti i bit all'interno del dato di un bit a destra o a sinistra.
Ad esempio :

```
char a;
a = 32;
```

Qui a vale (visto bit a bit) 00100000. Lo spostamento a sinistra di una posizione del numero è :

```
a = a << 1;
```

quindi il numero sarà 01000000. Con uno spostamento a destra di tre diverrà invece:

```
a = a >> 3; /* il numero ora è 00001000 */
```

Funzioni

Le funzioni servono per separare blocchi di codice, più specificatamente per "dividere" in blocchi più piccoli le operazioni che devono fare.
Una funzione viene dichiarata secondo il seguente schema :

```
variabile1 NomeFunzione(tipoparametro1 parametro1, tipoparametro2 parametro2, ecc.)
{
        corpo della funzione
}
```

Una funzione in C permette di "ritornare" un solo valore, mentre può avere un qualsiasi numero di parametri. In C non esiste una distinzione tra subroutine e funzioni.
L'equivalente di una subroutine in C è una funzione che torna un parametro vuoto (**void**).
Void è infatti un tipo di variabile, ma che in realtà non può essere associato ad una variabile ma solo usato per definire un tipo di parametro.

```
void Subroutine(int x, int y)
{ }
```

oppure

```
int Funzione (void)
{ }
```

Nella seconda funzione avemmo potuto anche omettere che il tipo di parametro da passare è nullo, mentre per il parametro da ritornare è sempre buona norma specificarlo perché compilatori diversi potrebbero considerare il tipo di default in modo diverso.

Per tornare un valore è necessario usare la parola chiave **return** seguita dal valore o dalla variabile che si vuole restituire. Ad esempio in una funzione che faccia la somma di due numeri interi :

```
int SommaInteri(int numero1, int numero2)
{
        return (numero1 + numero2);
}
```

Nella funzione main il return è il valore che viene ritornato dal programma come codice di errore.

Prototipi di funzioni

I prototipi di funzione servono principalmente per far sapere al compilatore che una determinata funzione sarà presente altrove nei sorgenti e che verrà dichiarata in un certo modo.
In pratica se ad esempio poniamo le funzioni alla fine del nostro sorgente o addirittura in un altro file (vedremo più avanti perché) il compilatore ha bisogno di sapere se queste ci sono o no per poterci avvertire che non può chiamare la funzione perché non c'è.
Un prototipo è fatto con la stessa struttura della dichiarazione di funzione, ma ha in realtà bisogno solo dei tipi di parametri.
Il prototipo della funzione vista precedentemente sarà quindi :

```
int SommaInteri(int, int);
```

Avremmo anche potuto lasciare i nomi delle variabili, che non danno fastidio al compilatore e che anzi possono essere d'aiuto al programmatore.
I prototipi normalmente se il sorgente è unico e breve si mettono in testa in modo che siano visibili da tutto il sorgente, altrimenti si mettono in un file "**.h**" detto file header.

Visibilità delle variabili

Le variabili possono avere diversi "ambienti" di visibilità. Ciò vuol dire che le variabili sono "visibili", cioè riconosciute dal programma a seconda del punto in cui vengono dichiarate.
Secondo la visibilità abbiamo diversi tipi di variabili.

Variabili globali (che sono visibili da tutto il programma)
Variabili locali di una funzione (visibili solo all'interno della funzione)
Variabili locali di pezzo di codice.

In realtà esistono anche altri tipi come le variabili locali di un modulo, che rimangono visibili solo all'interno di un modulo.

Una variabile è detta globale quando viene dichiarata all'esterno di tutte le funzioni del programma.
Ad esempio :

```
#include <stdio.h>

int globale; /* questa variabile è visibile da tutti i punti del programma */

int main()
{
        printf("%d", globale);
}
```

Se invece la dichiaro in altri punti :

```
int main()
{
        int locale_di_funzione; /* questa variabile è visibile solo all'interno del main */
        int i;

        printf("%d", globale);
        for(i = 0; i < 10; i++)
        {
                int locale_di_blocco; /* variabile visibile solo all'interno del ciclo for*/

                locale_di_blocco = i;
        }
        locale_di_blocco = locale_di_funzione; /* errore (1) */
}
```

Nel codice la variabile locale_di_blocco(1) non è visibile all'esterno del ciclo for e quindi il compilatore darà un errore.
Quando si fa un sorgente bisogna tenere conto di questo fatto, perché se si dichiarano due variabili con lo stesso nome in due ambienti (**scope**) diversi solo una delle due verrà utilizzata.

Altre espressioni condizionali

L'if - else non è l'unica espressione condizionale che possiamo utilizzare in c.
Le altre sono i costrutti **if - else if,** il costrutto **switch** e l'operatore punto interrogativo (**?**).

Il costrutto if ... else if

Questo costrutto si usa se si ha la necessità di verificare una serie di condizioni.
Diciamo che è una specie di if else con più di un else. Ad esempio :

```
if (a = =1) {
        .....
}else if (b = = 2 && a = = 3) {
        .....
}else if (c = = 1 && b = = 2) {
        .....
}else if (c = = 1 && a = = 2) {
        .....
}else {
        .....
}
```

Il costrutto switch

Il costrutto switch permette invece di operare una serie di scelte a seconda del valore di un'espressione. Il suo formato è :

```
switch (espressione da valutare)
{
        case valore1:
                ...
                break;
        case valore2:
                ...
                break;
        case valore3:
                ...
                break;
        default:
                ...
}
```

In pratica in base al valore dell'espressione viene eseguito il codice presente subito dopo il **case** corrispondente al valore e fino al **break**.
La parola chiave case identifica infatti il caso del valore. Se nessuno dei valori corrisponde al valore dell'espressione valutata viene eseguito il codice dopo la parola chiave **default**. La parte del default è opzionale, cioè se non è presente non viene eseguito alcun codice.
Se avessimo invece bisogno di eseguire lo stesso codice per più di un valore sarà sufficiente mettere un case sotto l'altro.

```
switch (a)
{
        case 1:
        case 2:
                ....
                break;
}
```

L'operatore ?

L'operatore punto interrogativo è invece un metodo compatto per fare valutazioni condizionali. In pratica è una versione compressa dell'if. Si usa nel modo seguente :

```
risultato = (a = = 8 ? 15 : 5);
```

In questo caso viene valutata l'espressione a = = 8, se risulta vera (**?**) viene assegnato il valore 15 altrimenti (**:**) viene assegnato il valore 5.

Accesso ai file

Vediamo ora come accedere ai file. Le funzioni di accesso ai file sono nella libreria di sistema "stdio.h" che abbiamo già conosciuto con la funzione printf. Questa funzione infatti non fa altro che inviare un buffer di caratteri allo standard output.
Un file è identificato da un particolare tipo di variabile : il puntatore a file (**FILE ***). Dichiariamo un nuovo puntatore a file.

FILE *fp;

Ora che abbiamo un riferimento possiamo passare alla apertura del file che ci interessa. Per far questo usiamo la funzione di libreria **fopen**.

fp = fopen("file.txt", "w");

Questa funzione imposta il valore del puntatore a file del file **file.txt** in modalità scrittura (**w**). La funzione torna una valore valido se è possibile aprire il file, in caso contrario torna **NULL**.
Normalmente viene infatti usato con il controllo di errore.

```
if ((fp = fopen("file.txt", "w")) == NULL) {
        printf("Errore nell'apertura del file !!!");
}else{
        ......
}
```

A questo punto, dopo aver aperto il file sarà necessario scriverci. Per questo c'è l'equivalente della funzione printf per i file : **fprintf**.

fprintf(fp, "Stringa da mettere nel file");

I parametri come possiamo vedere sono gli stessi della printf con l'eccezione che prima del primo parametro è stato inserito il file pointer (puntatore a file).
Scritto ciò che volevamo è necessario chiudere il file con l'istruzione **fclose.**

fclose(fp);

Modalità di accesso ai file

I file possono essere aperti in diverse modalità a seconda dell'utilizzo per cui vengono creati.

r	Apre un file di testo per la lettura
w	Apre un file di testo per la scrittura
a	Aggiunge dati a un file di testo

Aggiungendo a questi la lettera **b** (rb, wb, ab) il file non sarà più testo ma binario.
Se volessimo aprire il file in lettura e scrittura è invece necessario aggiungere il segno + (r+, r+b, ecc.).
Ci sono altre funzioni di diagnostica per sapere ad esempio se si è raggiunta la fine del file (**feof**) o per sapere se si è verificato un errore (**ferror**), ma è meglio dare un'occhiata più approfondita ad un qualsiasi manuale di c per rendersi conto delle possibilità.

Ancora sugli array

Abbiamo visto che gli array non sono altro che dei contenitori di dati. Essi in realtà possono essere anche a più di una dimensione. Per dicharare un array a due dimensioni ad esempio :

int schermo[80][25];

Abbiamo dichiarato un array di interi di 80*25. E per accedere ad un dato :

schermo[3][6] = 15;

Attenzione !!! : In C non viene effettuato un controllo sulla validità dei dati forniti per le coordinate in un array. Quindi se andate a scrivere o leggere al di fuori della definizione di un array potrebbero verificarsi dei seri problemi.

E' possibile anche fare logicamente array di puntatori.

char *a[10];

Dichiara un array di puntatori a carattere. **Attenzione che non è una stringa !!!!** Potrebbe essere usata per memorizzare una sequenza di stringhe. Ad esempio :

```
char nome[10], nome2[10];
strcpy(nome, "pippo");
strcpy(nome2, "pluto");
a[0] = nome;
a[1] = nome2;
```

Lettura dei parametri dalla riga di comando

A volte capita di voler chiamare un programma passandogli dei parametri. In C si usano gli argomenti (facoltativi) della funzione main. Questa può infatti diventare :

```
int main(int argc, char *argv[])
{ ..... }
```

La variabile **argc** contiene il numero di parametri che sono stati passati al programma. E' almeno 1 perché il primo parametro è il nome del file completo di percorso.
L'array **argv** è un array di puntatori a stringhe che sono i parametri passati (dalla posizione 1).

Enumerazione di variabili (enum)

Il tipo di dato enumerativo consiste in una serie di costanti intere dotate di nome che specifica quali valori sono ammissibili per una variabile di quel tipo. Ad esempio :

```
enum colore { rosso, verde, bianco, nero };
enum colore colore_macchina;
```

I valori di rosso, verde, ecc. sono stati assegnati automaticamente partendo da 0 ed incrementando il valore per ogni nuovo nome.
La cosa rende più leggibile il sorgente in alcuni casi. Ad esempio :

if (coloremacchina = = bianco)

è molto più leggibile di

if (coloremacchina = = 2)

Campi di bit

Essendo un linguaggio più vicino alla macchina di altri il C ha istruzioni che permettano di lavorare direttamente con i bit. I campi di bit permettono di risparmiare memoria per dati.

```
struct semaforo {
        unsigned rosso : 1;
        unsigned giallo : 1;
        unsigned verde : 1;
};
```

I numeri che seguono i due punti indicano il numero di bit necessari per la memorizzazione del dato. In questo caso serve un bit per ognuno dei tre colori per definire lo stato della luce. Tutta la struttura non occupa in realtà neanche un byte (in realtà ne occupa uno e non usa gli ultimi bit).

Unioni di dati

Una unione di dati (**union**) ci permette di unire più di un dato nello stesso spazio di memoria e quindi di risparmiare spazio. Ad esempio :

```
union prova_union {
        int intero;
        char carattere;
};
```

La definizione come possiamo vedere è molto simile a quella di una struttura. La differenza principale è che le due variabili condividono la stessa memoria, il che vuol dire che non possono essere utilizzate contemporaneamente. L'occupazione in memoria sarà necessariamente quello del dato più grande, nel nostro caso l'intero.
L'assegnamento avviene esattamente come per le strutture.

```
union prova_union u;
u.intero = 20;
```

Logicamente se assegno la parte intera della unione e poi vado a leggere la parte carattere otterrò dei dati che non sono congruenti.

Funzioni di utilizzo più comune

Altre funzioni di cui non abbiamo parlato ma che sono tra le più utilizzate sono :

Per i file (stdio.h)

fgetc	Legge un carattere dal file e lo torna (EOF se fine file)
fputc	Scrive il carattere specificato nel file alla posizione corrente
fread	Legge un certo numero di record di grandezza definita dal file
fwrite	Scrive un certo numero di record di grandezza definita dal file
fscanf	Stessa funzione di scanf ma per i file
fseek	Si posiziona alla posizione indicata all'interno del file
remove	Cancella il file specificato
rename	Rinomina il file specificato

Per l'I/O (stdio.h)

putchar	Scrive nello stdout un carattere
getch	legge un carattere dallo stdin
gets	Legge una stringa dallo stdin fino a che non incotra \n o EOF

Per le stringhe (string.h)

strcat	Accoda una stringa ad un'altra stringa
strchr	Cerca la posizione di un carattere all'interno della stringa
strcmp	Confronta due stringhe
strcpy	Copia una sringa in un'altre

Funzioni matematiche (math.h)

abs	Numero assoluto
cos	Coseno del numero
sin	Seno del numero
tan	Tabgente del numero
exp	Restituisce e elevato alla potenza specificata
log	Logaritmo natura di x
pow	Restituisce x elevato ad y
sqrt	Radice quadrata del numero

Altre funzioni le potete trovare negli header

time.h	Funzione per la gestione di data e ora
stdlib.h	Varie funzioni di utilità generale

Ho espressamente omesso tutte le funzioni che seppur comuni non sono ansi C o dipendenti dal sistema operativo su cui state sviluppando. Il mio consiglio rimane sempre lo stesso fatevi un giro negli aiuti (help) o nei manuali del c che utilizzate per sapere cosa avete a disposizione.

Introduzione al C++

Il C++ è la naturale evoluzione del linguaggio C come dice il nome stesso. Infatti tutte le funzioni base del C continuano a funzionare perfettamente nel nuovo linguaggio.
In compenso le migliorie e le funzioni lo rendono tale da essere considerato quasi un linguaggio diverso. Il concetto più innovativo del linguaggio è quello di **oggetto**.
Un oggetto viene creato sulla base di una **classe** che è il contenitore base dell'oggetto. Vedremo cosa vuol dire.
Per programmare in C++ dobbiamo infatti passare dalla programmazione strutturata (propria del C) a quella ad oggetti (propria del C++).
Mentre in C si fa tutto con le funzioni in C++ si preferisce (che è anche un metodo più pulito) lavorare con gli oggetti.
Facciamo un esempio. Creiamo un oggetto di tipo punto che fa parte di un oggetto di tipo linea che fa parte di un oggetto di tipo poligono che a sua volta può far parte di altri oggetti sempre più complessi. In questo modo il livello di astrazione durante la programmazione ci permette di lavorare ad un livello più alto.

Il primo programma in C++

L'equivalente di ciò che abbiamo visto in C per il C++ è :

```
#include <iostream>  // aggiunge la libreria iostream

int main()
{
        std::cout << "Ciao mondo\n";
}
```

In questo caso **std** è un'oggetto predefinito della libreria iostream che contiene al suo interno il metodo **cout**. Quest'ultimo si occupa di mandare il flusso (<<) verso l'esterno.
Aggiungendo all'inizio del programma :

```
using namespace std;
```

avremmo potuto omettere **std::** perché diciamo che lo spazio dei nomi (cioè il default) è std quindi la riga sarebbe diventata :

```
cout << "Ciao mondo\n";
```

La funzione equivalente in input come flusso è **cin** ad esempio :

```
int valore;
cin >> valore; // legge un valore dallo standard input e lo mette in valore
```

Concetto di classe

La classe per coloro che provengono dal C può essere considerata (anche se è un po' riduttivo) una estensione della struttura. Un classe è un contenitore di variabili e funzioni che a suo volta può essere contenuta in altri oggetti.
Si dice che la classe permette di "incapsulare" i dati e le funzioni, cioè di raggruppare le caratteristiche di un oggetto e il suo comportamento.

Un programma che manipola dati simili all'ambiente sarà non solo più facile da comprendere e mantenere, ma anche più coinciso. Ed inoltre sarà più affidabile perché verrà impedita qualsiasi accesso non consentito in fase di compilazione.

Dichiarazione di una classe

Un classe è strutturata in questo modo :

```
class Nome_Classe {
        int variabili;
                ….
        Nome_Classe(); // costruttore
        ~Nome_Classe(); // distruttore

        int Funzione1(short parametro): // Funzione di classe (membro)
                ….
};
```

Come possiamo vedere la struttura assomiglia molto alle **struct** con l'eccezione che in questo caso abbiamo delle funzioni (che d'ora in poi chiameremo membri) all'interno della classe.
L'accesso alle variabili e ai membri della classe avviene come per la struttura con il punto (***Nome_Classe.Funzione1(1);***).

Costruttore e Distruttore

Il costruttore di una classe è una funzione che dobbiamo **obbligatoriamente** dichiarare. Esso ha sempre il nome della classe stessa e non torna nulla. I parametri da passare al costruttore per inizializzare gli elementi della classe sono facoltativi.
Questa è la funzione che viene chiamata quando viene **creato** un nuovo oggetto della classe, quindi al suo interno ci va l'inizializzazione, cioè l'impostazione iniziale di tutti le variabili (dati membro) di proprietà della classe.

Il distruttore al contrario viene chiamato quando la classe viene cancellata dalla memoria (deallocata). Non può avere nessun parametro e si chiama sempre con il nome della classe preceduto dal tilde (~). Se non ce l'avete sulla tastiera si fa con ALT + 126.

Concetto di privato e pubblico

Un altro concetto introdotto nel C++ è quello di privato e pubblico.
All'interno di una classe non è sempre necessario che le variabili o i metodi che utilizziamo siano visibili dall'esterno, anzi è buona norma che non lo siano, se non nei casi espressamente previsti.
Il costruttore e il distruttore stanno sempre nella parte pubblica della classe.
Per definire quali parti sono pubbliche e quali parti sono private si usano le parole **public** e **private**.
Ad esempio.

```
class Punto   {
        private:
                int x;
                int y;

        public:
                Punto();
                ~Punto();
                SetX(int value);
        private:
                CalcPosizione();
};
```

In questo esempio le variabili x e y non sono accessibili dall'esterno della classe, ma la funzione SetX sì. Si tende ad utilizzare funzioni per accedere alle variabili all'interno di una classe perché la rende indipendente dai cambiamenti interni. Per esempio se cambiamo nome o tipo alla variabile x.
Logicamente potrebbero anche servirci dei membri non accessibili che non nostro caso è la CalcPosizione.
Qual è il vantaggio principale di questo modo di lavorare ? Che dall'esterno ho la possibilità di usare solo ciò che ho deciso, e che quindi all'interno della classe posso cambiare praticamente tutto esclusa la dichiarazione delle funzioni pubbliche senza che il resto del programma se ne accorga.

Costruttori Multipli

Un'altra miglioria del C++ rispetto al C è la possibilità di avere più funzioni con lo stesso nome ma parametri diversi. Questo mi permette di creare ad esempio una serie di funzioni che lavorino su un tipo di parametro diverso (int, long, double, ecc.) e di ricordare un nome solo senza preoccuparmi del tipo di varibile che gli viene passato.
Questa miglioria si riversa appunto oltre che nelle funzioni standard anche sui costruttori. A volte può essere infatti comodo utilizzare più costruttori con parametri diversi che inizializzino in modo diverso le strutture della classe.

Argomenti di default nelle funzioni

Nelle dichiarazioni delle funzioni è inoltre possibile decidere che alcuni parametri che devono però essere sempre gli ultimi, non siano obbligatori. E' sufficiente scrivere :

```
int Somma(int a, int b, int c=0, int d=0);
{ return(a + b + c + d); }
```

Per avere una funzione che ha un numero di parametri variabile da un minimo di due ad un massimo di quattro. Nel caso in cui uno degli ultimi due parametri non ci sia viene inizializzato a 0.

Utilizzo di una classe

Come già detto una classe si utilizza come una struttura, cioè accedendo ai membri con il punto. IN realtà esistono due differenti modi di creare una classe :

```
Linea l;
l.Disegna();

Linea *l;
l = new Linea();
l->Disegna();
delete l;
```

Il primo metodo crea la classe automaticamente e la distruggerà nel momento in cui esco dalla funzione, mentre nel secondo caso ho creato un puntatore alla classe e poi **creato dinamicamente** l'oggetto della classe con l'operatore **new**. Questo operatore si occupa di allocare la memoria per l'oggetto e poi chiama il costruttore dell'oggetto.
Si dice che crea una **istanza** dell'oggetto stesso.
Nel secondo caso inoltre, visto che ho allocato io la memoria è anche necessario che la liberi per evitare che rimanga occupata. Questo si fa con l'operatore **delete**.

Da notare come nel primo caso abbiamo utilizzato l'operatore punto per accedere al membro Disegna della classe, mentre nel secondo caso, dato che ***l*** era un puntatore abbiamo utilizzato il puntatore freccia (->).

Scrittura del codice della classe

Logicamente dopo aver definito una classe è necessario scrivere il suo codice. Per farlo si scrive semplicemente il nome della classe seguito da due due punti (::) prima del nome della funzione.
Ad esempio :

```
int Linea::Disegna()
{
        ……
}
```

Organizzazione del programma (divisione su più file)

Per migliorare la manutenzione e la gestione di un programma, si usa dividere il sorgente stesso in più file o **moduli**. I moduli in C e in C++ hanno normalmente delle convenzioni per le estensioni.
In C i moduli del programma stesso, dove cioè si trovano le funzioni hanno estensioni **.c**, mentre i file che contengono le dichiarazioni delle funzioni e delle variabili (header) hanno estensione **.h**.
In C++ normalmente invece i moduli hanno estensione **cpp** e gli header **hpp**.
La divisione in moduli consente di migliorare i tempi di compilazione perché vengono compilati solo i moduli modificati e linkato il tutto. Si tende inoltre a raggruppare i moduli per funzioni che operino nello stesso campo.
Il programma che si occupa di controllare quali sono i moduli da ricompilare, se non avete un ambiente integrato come il visual c++, è il **make**. Questo programma normalmente sempre presente interpreta un file di testo che gli dice quali moduli vanno compilati se sono stati modificati altri moduli e come linkare il tutto. Permette inoltre di effettuare delle compilazioni condizionali.

Ultima modifica : 02/10/2002 20.21

Sommario