Introduzione al linguaggio C

Per i sistemi embedded

Il linguaggio C si è affermato come il linguaggio standard per lo sviluppo per dispositivi embedded. Questo perché è un linguaggio relativamente semplice, non ad alto livello (che permette la manipolazione diretta della memoria) e che è anche ampiamente diffuso in altri ambiti.

L'implementazione di un linguaggio originalmente pensato per la programmazione su computer ad uso generale a dispostivi molto più limitati non è immediata e in molti casi architetture più semplici non dispongono di un compilatore che supporta pienamente il linguaggio C, mentre tutti i microcontrollori moderni sono pensati per poter essere utilizzati con un compilatore C.

Esistono varie versioni del linguaggio C, la versione più comunemente implementata per le architetture embedded è la versione C99 (alcuni compilatori però si limitano al supporto per C89).

Vediamo ora le principali caratteristiche del linguaggio C.

Variabili e tipi dati

Tutti i programmi hanno la funzione di elaborare un qualche tipo di informazione, immagazzinata nelle variabili. Una variabile in C è definita dichiarando che un identificatore (o nome) deve essere trattato come un particolare tipo di dato (i tipi sono generalmente predefiniti).

```
<tipo> <identificatore>;
```

Il linguaggio C è un linguaggio *case-sensitive* quindi distingue tra lettere maiuscole e minuscole per gli identificatori. Vediamo alcuni dei tipi dati più utilizzati:

Numeri interi (con o senza segno): `char`, `short`, `int`, `long` (in ordine di lunghezza). La lunghezza dei tipi non è definita dallo standard e dipende dal compilatore utilizzato. Di seguito viene riportata la tabella con la lunghezza delle variabili per il compilatore Microchip MPLAB XC8 (che verrà utilizzato per la prima parte di questo corso)

TABLE 4-3: INTEGER DATA TYPES

Type	Size (bits)	Arithmetic Type
`__bit`	1	Unsigned integer
`signed char`	8	Signed integer
`unsigned char`	8	Unsigned integer
`signed short`	16	Signed integer
`unsigned short`	16	Unsigned integer
`signed int`	16	Signed integer
`unsigned int`	16	Unsigned integer
`__int24`	24	Signed integer
`__uint24`	24	Unsigned integer
`signed long`	32	Signed integer
`unsigned long`	32	Unsigned integer
`signed long long`	32/64	Signed integer
`unsigned long long`	32/64	Unsigned integer

Numeri in virgola mobile: `float` (virgola mobile a precisione singola). I processori utilizzati in questo corso non sono dotati di un'unità di calcolo in virgola mobile quindi le operazioni su queste variabili sono piuttosto lente.

Array

È possibile definire una lista con indice di un qualsiasi tipo dati, ovvero un *array*, semplicemente aggiungendo il numero di elementi tra parentesi quadre al termine della dichiarazione:

```
<tipo> <identificatore>[<dimensione>];
```

È possibile accedere ad un elemento di un array con l'espressione `<identificatore>[<indice>]`. Si ricorda che in C l'indicizzazione degli array inizia da 0 quindi il primo elemento di un array è `<identificatore>[0]` e l'ultimo è `<identificatore>[<dimensione> - 1]`.

Esiste un metodo più flessibile per accedere agli elementi di un array (o in generale a qualsiasi variabile o parte della memora) tramite l'uso dei puntatori che verrà illustrato in seguito

Operatori

Una volta che le variabili sono definite vogliamo manipolarle in qualche modo: questo viene fatto usando gli *operatori*.

L'operatore più semplice è l'*operatore di assegnamento* (=): il valore a destra dell'uguale viene assegnato alla variabile a sinistra. Le assegnazioni possono essere concatenate, ad esempio scrivendo `a = b = 1`

Sono presenti gli operatori aritmetici elementari come, addizione (+), sottrazione (-), moltiplicazione (*), divisione (/) e modulo (%) [definito solo per i numeri interi].

Sono presenti i seguenti operatori logici che restituiscono un valore booleano (vero o falso)[1]: maggiore/minore (e maggiore/minore uguale): (<, <=, >, >=), uguale (==), diverso (!=), AND logico (&&), OR logico (||), NOT logico (!).

Inoltre, ci sanno molto utili gli operatori *bit a bit* (bitwise) che agiscono sui bit di una variabile:

Operatore	Espressione C	Funzione
AND	`a & b`	1 nelle posizioni in cui sia a che b ha bit a 1, 0 altrimenti
OR	`a | b`	1 nelle posizioni in cui almeno uno tra a e b ha bit a 1, 0 altrimenti
XOR	`a ^ b`	1 nelle posizioni in cui solo uno tra a e b ha bit a 1, 0 altrimenti
NOT	`~ a`	Inverte il valore di ogni bit della variabile a
Shift a sinistra	`a << b`	Scala i bit di a a sinistra di un numero b di posizioni
Shift a destra	`a >> b`	Scala i bit di a a destra di un numero b di posizioni

È possibile combinare gli operatori con l'operazione di assegnamento: ad esempio `x = x + y` si può scrivere come `x += y`.

Precedenza tra operatori

Il linguaggio C ha una serie di regole di precedenza tra gli operatori. Nel caso l'ordine non andasse bene o per rendere più esplicita la funzione di una data operazione è possibile usare le parentesi `()` per cambiare l'ordine: le espressioni nelle parentesi vengono valutate prima.

Espressioni ed istruzioni

`a = rupd(x, y)` è una espressione, mentre `a = rupd(x, y);` è un istruzione. Il segno `;` viene usato per indicare al compilatore che no vogliamo fare altro con il risultato di una data espressione. Ogni istruzione C deve terminare con `;`. Un insieme di istruzioni può essere racchiuso con delle parentesi graffe `{}` per formare un blocco, sintatticamente equivalente ad una singola istruzione

[1] In C tradizionalmente non esiste un tipo dedicato per i valori booleani: si una un qualsiasi tipo numerico considerando 0 come valore falso e qualsiasi altro numero come valore vero.

Controllo del flusso di esecuzione

Il linguaggio C supporta le seguenti strutture per controllare il flusso di esecuzione di un programma

Esecuzione condizionale `if` – `else`

Questa struttura è utilizzata per eseguire o meno un blocco in base al risultato di un'espressione. Ha la seguente struttura:

```
if(<espressione>) {
      istruzione1; //eseguita se <espressione> è valutata true
} else {
      Istruzione2; //eseguita se <espressione> è valutata false
}
```

È possibile concatenare più condizioni con la seguente struttura

```
if(<espressione1>) {
      istruzione1; //eseguita se <espressione1> è valutata true
} else if(<espressione2>) {
      istruzione1; //eseguita se <espressione1> è valutata false e <espressione2> è valutata true
} else {
      istruzione2; //eseguita se <espressione1> ed <espressione2> sono valutate false
}
```

Cicli `while`, `do` – `while` e `for`

Il linguaggio C ha tre strutture che permettono di ripetere per un numero fisso o variabile di volte un blocco di codice.

Il ciclo `while` ripete il blocco fino a che l'espressione indicata viene valutata `false`. La decisione se eseguire il blocco viene fatta prima dell'avvio del ciclo (controllo in testa), quindi è possibile che il blocco non venga mai eseguito:

```
while(<espressione) {
      istruzioni;
}
```

Il ciclo do – while è molto simile solo che il controllo viene fatto alla fine del ciclo, quindi il bocco viene eseguito almeno una volta:

```
do {
      istruzioni;
} while(<espressione>);
```

Una versione più avanzata del ciclo `while` è data dal ciclo `for` che permette di esprimere in un singolo costrutto un'istruzione di inizializzazione, la condizione del ciclo `while` ed un'istruzione da eseguire alla fine di ogni ciclo.

```
for(<inizializzazione>, <condizione>, <azione>) {
      istruzioni;
}
```

Il ciclo for viene spesso usato per iterare sopra gli elementi di un array:

```
for(int i = 0; i < <lunghezza>; i++) {
      array[i] = 0;
}
```

Funzioni

Ogni programma C è costituito da una o più funzioni: una funzione permette di racchiudere una parte del programma rendendola riutilizzabile. Il programma c prende avvio da una funzione chiamata `main()`.

Una funzione C può restituire un valore oppure no e qualsiasi funzione può chiamare qualsiasi funzione (una funzione può anche chiamare sé stessa[2]) o essere chiamata da una qualsiasi funzione.

Una funzione è definita dal tipo di funzione (che può essere qualsiasi tipo oppure `void` per le funzioni che non restituiscono un valore), un nome e tra parentesi una lista opzionale di argomenti. Gli argomenti sono i parametri che vengono passati alla funzione, vengono dichiarati allo stesso modo delle variabili. Se la funzione restituisce un valore questo viene indicato con l'istruzione `return`.

```
<tipo> <nome>(<lista argomenti) {
      istruzioni;
}
```

Un esempio di funzione potrebbe essere:

```
int mac(int a, int b, int c) {
      return a + (b * c)
}
```

Se è necessario restituire più di un valore da una funzione allora bisogna utilizzare i puntatori oppure inserire i risultati in variabili globali.

Variabili globali e locali

Una variabile **globale** è definita al di fuori di tutte le funzioni tipicamente all'inizio del programma (dopo le istruzioni al preprocessore) e può essere utilizzata da tutte le funzioni nel file.

Una variabile **locale** (o automatica) è definita all'interno di una funzione ed è solo utilizzabile all'interno di tale funzione. Una variabile può essere locale ad un blocco di codice: ogni volta che si aprono le parentesi graffe cambia lo scopo e le variabili dichiate all'interno di quel blocco hanno esistenza solo in quel blocco:

```
void test() {
      int a;
      for(int i = 0; i < 10; i++) {
            int b = 20 - i;
      }
      a = b; //Errore: la variabile b non esiste più qui
}
```

Puntatori

Ogni variabile dichiarata avrà una certa posizione in memoria identificata da un indirizzo. Grazie al meccanismo dei puntatori sarà possibile accedere direttamente alla memoria tramite gli indirizzi.

Nel linguaggio C è possibile ottenere l'indirizzo di memoria di una qualsiasi variabile grazie all'operatore & (*indirizzo di*). Ad esempio, possiamo scrivere `int a; p = &a`. Ora p contiene l'indirizzo di a. La varabile p che contiene un indirizzo di memoria si chiama **puntatore**.

Per dichiarare un puntatore si usa la seguente sintassi: `<tipo> *<nome puntatore>;` ad esempio, `int *p;` Il puntatore deve essere dello stesso tipo della varabile di cui deve contenere l'indirizzo ma è anche

[2] Questo non è sempre vero: per ciò che sia possibile senza effetti collaterali una funzione deve essere scritta in modo che sia rientrante. Questo richiede degli accorgimenti particolari, specialmente con alcuni compilatori per architetture molto semplici che non si comportano come un compilatore C standard per computer ad uso generale.

possibile dichiarare un puntatore di tipo `void` che può contenere l'indirizzo di una variabile di qualsiasi tipo.

Per accedere al contenuto dell'indirizzo di memoria si usa l'operatore di *dereferenziazione* *: anteponendo * al nome del puntatore si può accedere al contenuto dell'indirizzo di memoria associato al puntatore. Ad esempio: `int a; int *p = &a; *p = 7` al termine di questa sequenza di istruzione la variabile a ha il valore 7.

Questo tipo di scrittura potrebbe non sembrare utile[3] (ed in effetti abbiamo illustrato un caso banale) ma i puntatori di memoria sono alla base di buona parte delle strutture più complesse che sono comunemente utilizzate. L'implementazione degli array e delle stringhe in C si basa sui puntatori di memoria.

Preprocessore

Il linguaggio C include un preprocessore che permette di effettuare operazioni sui file di codice prima della loro inclusione. Le direttive per il preprocessore sono indicate con #.

Una delle direttive più usate è la direttiva `#include` che permette di includere file di intestazione che consentono l'accesso a funzioni definite in altri file .c. Altrettanto utile, specialmente per la programmazione embedded, è la direttiva `#define` che permette di definire costanti ed espressioni che vengono valutate durante la compilazione.

Il preprocessore è uno strumento molto potente ed un programma che usa molto le sue funzionalità può risultare difficile da leggere e da mantenere.

Tipi interi standard C99

All'inizio abbiamo visto che i tipi interi non hanno una lunghezza ben definita, dato che dipende dal compilatore utilizzato. Per rendere più esplicita la lunghezza dei tipi e rendere il codice più facilmente portabile tra vari compilatori lo standard C99 ha introdotto nuovi tipi interi.

Per poter accedere a questi tipi è necessario includere il file di intestazione stdint.h con la direttiva `#include <stdint.h>`

Sono definiti sia tipi con segno che senza segno e prendono la seguente forma: per i tipi senza segno `uint_XXt` dove XX è la lunghezza in bit del tipo, per quelli con segno `int_XXt`. Sono definiti per le seguenti lunghezze: 8, 16, 32 e 64 bit.

[3] Qui introduciamo solo i puntatori per una spiegazione più completa vedere http://www.cplusplus.com/doc/tutorial/pointers/ (attenzione però è riferito al linguaggio C++ che presenta alcune piccole differenze)