UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C – Concetti base

Ultimo aggiornamento: 15/10/2020

Il linguaggio C

- È un linguaggio di programmazione general-purpose progettato inizialmente da Dennis Ritchie dei Bell Laboratories e implementato nel 1972. Inventato per sopperire ai limiti del linguaggio B e BCPL
- Nel 1983 il comitato X3J11 dell'American National Standard Institute (ANSI) sviluppò il cosiddetto ANSI C o C standard
- Lo standard è stato poi aggiornato nel 1999 con l'introduzione di alcune varianti e chiarificazioni. In generale non faremo riferimento a queste varianti (soprattutto i variable length array). Ulteriore aggiornamento nel 2011 con il cosiddetto C11.
- Il linguaggio è di alto livello perché non ha una traduzione «diretta», ovvero 1 a 1, ovvero biunivoca, in linguaggio macchina, a differenza dell'assembly.
- È però anche di basso livello perché lo si può utilizzare come una specie di assembly poco più evoluto. Questo modo di usarlo è però poco lungimirante e decisamente sconsigliato.

Un programma C

- Un programma in linguaggio C è composto da alcune parti che possono o meno essere presenti e più o meno ripetute e alternate tra loro:
  - dichiarazioni/definizioni (di variabili e funzioni)
  - direttive per il preprocessore
- Ignoriamo per il momento le dichiarazioni che non siano definizioni e preprocessore e procediamo ad analizzare le definizioni.
- Abbiamo già visto che cos'è una variabile, ovvero una o più celle di memoria al cui indirizzo viene dato un nome simbolico.
- In C ogni variabile può occupare uno o più byte. A differenza di quello che facevamo con l'Assembly ADE8, nel momento in cui diamo un nome alla locazione di memoria, diciamo anche quanti byte occupa e soprattutto quale è il suo **tipo**, ovvero che cosa c'è dentro e come viene codificato.

I tipi di dati numerici interi

- Esistono 8 tipi di dati interi. La loro definizione da standard è molto flessibile e consente ai diversi produttori di fare quello che vogliono. Per chiarezza però noi faremo riferimento sempre a queste definizioni:

Nome	**Dimensione**	**Descrizione**
`char`	1 byte	intero con segno a 8 bit
`unsigned char`	1 byte	intero senza segno a 8 bit
`short`	2 byte	intero con segno a 16 bit
`unsigned short`	2 byte	intero senza segno a 16 bit
`int`	4 byte	intero con segno a 32 bit
`unsigned int`	4 byte	intero senza segno a 32 bit
`long long`	8 byte	intero con segno a 64 bit
`unsigned long long`	8 byte	intero senza segno a 64 bit

- In realtà le regole sui nomi dei tipi interi sono leggermente più complesse, ma le approfondiremo nel seguito, eventualmente.

I tipi di dati numerici in virgola mobile

- Esistono 2 tipi di dati numerici in virgola mobile:

Nome	Dimensione	Descrizione
`float`	4 byte	numero in virgola mobile a 32 bit
`double`	8 byte	numero in virgola mobile a 64 bit

- Questi valori numerici sono codificati secondo lo standard IEEE 754-1985, ovvero quello che abbiamo utilizzato durante le lezioni iniziali del corso.

Definizione di variabili

- Con *definizione di variabile*, si intende il modo con cui in un file che segue la sintassi del linguaggio C viene richiesto di riservare memoria per contenere un certo dato e gli viene assegnato un nome simbolico.
- La sintassi per definire una variabile in C è:

```
<tipo> <nome variabile> ;
```

- Notate che prima di tutto si esprime il tipo del dato che richiediamo, poi gli assegniamo un nome, poi chiudiamo la definizione con un punto e virgola.
- In C il punto e virgola viene utilizzato in diversi punti del linguaggio per indicare la fine della «cosa» che si sta facendo. In questo caso la fine della definizione.
- Il tipo può essere una delle parole riservate indicate precedentemente. Vedremo in seguito altri tipi più complessi.

Esempi di definizioni

- Vediamo alcune definizioni di variabili:

```
char c;
short s;
int i;
long long numero;
float f;
double radice;
```

- Come vedete, deciso il tipo, bisogna scegliere un nome per fare riferimento alla variabile.
- Quali sono i nomi validi?

Identificatori

- In C gli identificatori utilizzabili per dare un «nome» a qualcosa possono contenere qualsiasi combinazione di:
  - lettere maiuscole e minuscole
  - numeri
  - il carattere «_», detto *underscore* o *sottotratto*.
- L'unico ulteriore vincolo è che non possono cominciare con un numero.
- Infatti le sequenze che iniziano con un numero sono *costanti* o *letterali* numeriche. Le costanti numeriche possono essere:
  - decimali: cominciano con una cifra da 1 a 9 e proseguono con altre cifre da 0 a 9.
  - ottali: cominciano con 0 e proseguono con altre cifre da 0 a 7.
  - esadecimali: cominciano con «0x» o «0X» e proseguono con altre cifre da 0 a 9 e con le lettere (maiuscole o minuscole) da «A» a «F».
- Identificatori corretti: test, mia_var, var001, var002, _pippo, a_3f4_5.
- Valori corretti: 23, 45, 065 (questo è il numero 53), 0xff (questo è il numero 255), 0X32 (questo è il numero 50).

Letterali numerici

- Col termine *letterale* si intende un valore costante del C.
- Per i tipi numerici esistono i letterali di tipo int che abbiamo appena visto:
  - Decimali: 123, 245681, ecc…
  - Ottali: 0123, 02456, ecc…
  - Esadecimali: 0x123, 0x245abc, ecc…
- Ma con il suffisso «u» è possibile specificare che il loro tipo è unsigned int, quindi 123u è di tipo unsigned int.
- Esistono inoltre i letterali di tipo double, definiti dalla presenza di un *punto decimale*:
  - 123. è un double, come anche 123.0 e come 123.345
- È inoltre possibile utilizzare la notazione esponenziale o scientifica e aggiungere al numero una «E» o «e», seguita dall'esponente:
  - 1.23e2, oppure 123.e-2 oppure 12e7
- Se si vuole un letterale di tipo float, è necessario usare il suffisso «f»:
  - 1.23e2f, oppure 12.f, oppure 0.12345f

Altri letterali

- Come già visto nel processore ADE8, i caratteri della tabella ASCII sono scomodi da utilizzare col loro valore numerico.
- Anche il C permette di inserire quei numeri come simboli digitati direttamente da tastiera includendoli in apici singoli:
  - Il valore 0x30 si può scrivere '0',
  - il valore 0x41 si può scrivere 'A',
  - oppure il valore 0x61 si può scrivere 'a'.
- Tutte le espressioni tra **'** (singolo apice) sono ancora di tipo int (**non sono char!**).
- E se voglio inserire un carattere non presente sulla tastiera? Oppure un "a capo"?
- Bisogna utilizzare un *codice di escape* ovvero un carattere speciale che indica un caso particolare.
- Nei letterali tra singoli apici il codice di escape è il carattere **** (backslash), che viene poi seguito da codici specifici.
- Vediamo quali sono questi codici.

Sequenze di escape

- Ecco le principali sequenze di escape:

Sequenza di escape	Valore	Significato
`\t`	0x09	Tabulazione
`\n`	0x0A	A capo (LF)
`\r`	0x0D	Torna a inizio riga (CR)
`\"`	0x22	Doppie virgolette
`\'`	0x27	Singolo apice
`\\`	0x5C	Backslash
`\`*num*	Qualsiasi	Il numero *num* interpretato come **ottale**
`\x`*num*	Qualsiasi	Il numero *num* interpretato come esadecimale

- È interessante notare ad esempio che il valore 10 può essere scritto come 10, 012, 0x0a, '\n', '\12', o anche '\x0a'.
- Il char che vale 0 (NUL nella tabella ASCII) può essere scritto come 0, 0x0, 0x00, 0x000, …, 0x00000000, '\0', '\00', '\000', ‘\0x0', ‘\0x00', …

Inizializzazione di variabili

- Di default, le variabili non hanno un valore predefinito, quindi bisognerà successivamente eseguire comandi per assegnargli qualcosa.
- È possibile anche inizializzare una variabile, in modo che all'avvio del programma (o meglio all'inizio della loro «vita») abbia un valore preciso:

```
<tipo> <nome-varibile> = <espressione> ;
```

- Notate la differenza concettuale con quello che accade nell'assembly di ADE8: qui la definizione riserva uno spazio di memoria ed è opzionale inizializzare con un valore.
- In ADE8 invece il numero veniva messo in memoria e quindi aveva un suo indirizzo ed era opzionale dargli un nome.

- Che cosa si intende con *espressione* in C?

Espressioni del linguaggio C

- Uno degli elementi fondamentali del C sono le sue espressioni. Ne esistono diverse, ma le più semplici possono essere descritte con poche regole:
  - Un `<letterale>` è un'espressione.
  - Un `<identificatore>` è un'espressione.
  - La combinazione
    `<espressione> <operatore-binario> <espressione>`
    è un'espressione.
  - La combinazione
    `<operatore-unario-prefisso> <espressione>`
    è un'espressione.
  - La combinazione
    `<espressione> <operatore-unario-postfisso>`
    è un'espressione.
  - La combinazione `(<espressione>)` è un'espressione.
- Queste regole non sono esaustive, ma ci consentono di proseguire e di analizzare diverse combinazioni.

Tipo e valore di un'espressione

- Le espressioni possono avere un valore e in questo caso hanno anche un tipo associato. Quindi ci si può chiedere: «Quanto **vale** questa espressione?»
- Le espressioni possono anche avere un effetto, ovvero ci si può chiedere «Che cosa **fa** questa espressione?».
- (Purtroppo) il C consente di avere tutte le combinazioni. Partiamo dal caso semplice, ovvero espressioni che valgono qualcosa e non fanno niente. Le tre espressioni seguenti:

```
21          numero          3.7f
```

hanno un valore (e un tipo), ma non «fanno» nulla. La prima è di tipo `int`, la seconda (facendo riferimento alla definizione data a pag. 7) è di tipo `long long`, mentre la terza è di tipo `float`.

Esempi di definizioni con inizializzazione

- Vediamo di aggiungere una inizializzazione alle definizioni precedenti:

```
char c = 21;
short s = 0xffff;
int i = '7';
long long numero = 1234567890123;
float f = 3.141592f;
double radice = 1.41421356237309504880168872420969;
```

- In questo modo la memoria allocata per la variabile viene anche riempita con un valore iniziale durante la sua creazione (detta anche allocazione).
- Notate che per ora abbiamo utilizzato solo espressioni composte da letterali.

Variabili di sola lettura

- È possibile specificare che la variabile in memoria potrà essere solo letta, ovvero che è una variabile *read-only*, tramite la parola riservata **const**.

const *<tipo> <nome-varibile>* **=** *<espressione>* **;**

- Questa variabile è identica alle altre (ha un tipo, una locazione di memoria e viene inizializzata), se non per il fatto che nessun comando successivo potrà modificarne il valore.
- Tecnicamente la parola chiave const potrebbe anche essere posta dopo il tipo, ma è molto inusuale e sconsigliabile per gli studenti.
- Attenzione! Questa **non è una costante** (nonostante il nome lo ricordi chiaramente). Non può essere usata dove è indispensabile avere una espressione costante. Semplicemente non è possibile assegnare a questa variabile un valore diverso durante l'esecuzione del programma.
- Ad esempio:

```
const double pi_greco = 3.14159265358979323846264383;
```

Definizione di più variabili nella stessa linea

- È possibile anche definire più variabili assieme per avere una scrittura più compatta. La sintassi è:

<tipo> <nome-var-1>, <nome-var-2>, … , <nome-var-N> ;

- In pratica, si scrive il tipo una sola volta e si elencano di seguito le variabili che si intende definire. È inoltre possibile aggiungere o meno una inizializzazione ad ogni variabile.
- Vediamo qualche esempio:

```
int i, j, k;
char a = 9, b = 23, c = 12;
short s1, s2 = 33, s3;
```

Le funzioni

- Un programma C è costituito di tanti sottoprogrammi detti *funzioni*. Una funzione è l'unico «posto» in cui è possibile mettere comandi in linguaggio C.
- La forma più generale di una definizione di funzione è:

```
<tipo-di-ritorno> <nome-funzione> ( <parameteri> ) {
        <comandi>
}
```

- In questa definizione è importante notare due elementi: le parentesi tonde aperte e chiuse e le parentesi graffe aperte e chiuse.
- Le parentesi tonde permettono di distinguere tra la definizione di una funzione e quella di una variabile: appena in una definizione si vede aprire una parentesi tonda, sappiamo che questa è una funzione.
- Le parentesi graffe invece segnalano la presenza di un blocco di comandi. Devono essere obbligatoriamente presenti.

Le funzioni

- Con *parametri* si intendono delle variabili temporanee che nascono con la funzione e servono per fornire informazioni alla funzione. I parametri dovranno poi essere inizializzati esternamente per utilizzare la funzione.

- La più semplice (e raramente utilizzata) definizione di una funzione è la seguente:

```
void nomeFunzione ( void ) {
        <comandi>
}
```

- La parola riservata *void* indica una «assenza» e in particolare l'assenza di un tipo di ritorno (cioè la funzione non restituisce nulla) e l'assenza di parametri (la funzione non riceve informazioni dall'esterno).

La funzione *main*

- Noi lavoreremo sempre con programmi scritti in linguaggio C in un ambiente «ospitato» (*hosted*), ovvero con un Sistema Operativo che si occupa di gestire il caricamento del nostro programma.
- Per questo motivo è importante che sia chiaro al Sistema Operativo quale è la prima funzione da eseguire all'interno di un programma C.
- La funzione si chiama *main* e **deve** avere la seguente forma:

```
int main(void)
{
        <comandi>
}
```

- Esiste anche un'altra forma standard del *main* che consente di ricevere i parametri dalla linea di comando, ma la vedremo successivamente.
- All'interno di questa funzione, come nelle altre, è possibile utilizzare dei *comandi C*, in inglese *statement*.

Il return statement

- Come visto, una funzione può restituire un valore (del tipo specificato prima del nome) oppure no (in questo caso si indica void).
- Il primo statement (il primo comando C) che vediamo è return.
- La sintassi è

```
return <espressione> ;
```

- Se la funzione restituisce un valore, questo deve essere passato come parametro del comando *return* (l'espressione indicata). Se la funzione non restituisce nulla (void), l'espressione può essere omessa:

```
return ;
```

- Quando viene eseguito il comando *return*, la funzione termina e ritorna al chiamante (analogo all'istruzione RET di ADE8).
- Se una funzione raggiunge la } esegue un *return* implicito senza parametri. Ovviamente questo ha senso solo nelle funzioni che hanno tipo di ritorno void.

Il primo programma in C

- Ecco il nostro primo programma, che è anche il programma minimo che si possa realizzare in C:

```
int main(void)
{
    return 0;
}
```

- Questo programma non fa ovviamente nulla, ma non ha errori: contiene la funzione main, questa è correttamente definita (non accetta parametri e ritorna un int), contiene il comando return che imposta il valore di ritorno (in questo caso a 0, un valore classico per dire «programma terminato correttamente»).
- Notate che non è possibile terminare la funzione senza che venga impostato il valore di ritorno.
- Iniziamo ora a vedere altri comandi per fare qualcosa di utile!

Expression Statement

- Esistono numerosi comandi in C e li introdurremo progressivamente durante il corso.
- Lo statement più comune è quello costituito da una espressione seguita da un punto e virgola, che ne indica la fine.

```
<espressione> ;
```

- Dice che l'espressione deve essere **valutata**, ovvero che bisogna eseguire quanto indicato.
- Sintatticamente l'espressione può essere qualsiasi, ma di norma, perché questo comando abbia senso, deve essere un'espressione con un effetto sulle variabili coinvolte (*side effect*), altrimenti si esegue un calcolo il cui risultato viene buttato.
- Per fare questo, iniziamo a vedere alcuni operatori, che ci consentono di comporre espressioni che oltre a valere qualcosa, facciano qualcosa.

L'assegnamento

- L'operatore più utilizzato è certamente l'*assegnamento*, ovvero un operatore binario che prende il valore a destra e lo scrive in memoria in quello di sinistra.
- La forma più generale di assegnamento è

```
<lvalue> = <rvalue>
```

- *rvalue* e *lvalue* sono due *espressioni*, ovvero due sequenze di simboli che rispettano la grammatica del C e che hanno un significato logico all'interno del programma.
- La sintassi del C prevede diverse regole per la scrittura di espressioni, ma quello che dobbiamo intanto distinguere è che alcune espressioni hanno uno spazio di memoria associato, mentre altre no. Le espressioni che hanno uno spazio di memoria associato sono dette *lvalue*. Ogni *lvalue* può essere utilizzato dove è previsto un *rvalue*, dato che da una espressione in memoria si può leggere il corrispondente valore.

Esempi elementari di assegnamento

- Ecco alcuni esempi elementari di assegnamento utilizzati in un programma C:

```
char x;
char y;
char z;

int main(void) {
        x = 3;
        y = x;
        z = 0xf5;
        x = z;
        return 0;
}
```

Tre definizioni di variabili

Definizione di una funzione

Questo è un *expression statement*: l'espressione è «x = 3», seguita dal «;» che indica la fine dello statement. L'assegnamento è un operatore che ha un effetto su x, infatti ne cambia il valore.

x è un *lvalue*, possiede infatti un indirizzo in memoria in cui possiamo inserire valori.
3 invece è un *rvalue* perché vale 3, ma non corrisponde ad una cella di memoria.
Sarebbe un grave errore scrivere «3 = x», perché tenteremmo di leggere il contenuto di x e metterlo in 3!?

Questo è un *return statement*.

Operatori

- Il C è particolarmente ricco di operatori, ovvero di simboli che vengono tradotti in istruzioni del processore per cui realizziamo il codice macchina, ovvero il nostro programma.
- Vediamo quelli aritmetici elementari che potremo usare sui tipi di dato interi e in virgola mobile:
- Gli operatori aritmetici binari fondamentali sono:
  - Somma: **+**
  - Sottrazione: **-**
  - Prodotto: *****
  - Quoziente: **/**
  - Resto o Modulo: **%** (solo per numeri interi)
- Ci sono due operatori unari (o monadici) aritmetici:
  - Segno positivo: **+** (implicito e sostanzialmente inutile)
  - Segno negativo: **-**

Esempi di assegnamento ed espressioni

- Ecco alcuni esempi utilizzati in un programma C:

```
int x;
int y;
int z;
int t = 5;

int main (void) {
        x = 3;
        y = x + 5;
        z = y * 3 + 2;
        x = 2 + y * 3;
        y = x / -t;
        z = x % -t;
        return 0;
}
```

Precedenza degli operatori

- Nelle espressioni precedenti, abbiamo visto due espressioni equivalenti:

```
y * 3 + 2
2 + y * 3
```

- Entrambe eseguono prima la moltiplicazione del valore di y per 3 e poi la somma con due. Questo non ci sorprende, dato che è anche quello che avviene nella usuale notazione matematica utilizzata in analisi.
- La motivazione è che l'operatore * ha precedenza più alta rispetto al +.
- È possibile fare una tabella delle precedenze che consenta di risolvere eventuali incertezze:

Precedenza	Operatori
1	+ - (unari)
2	* / %
3	+ - (binari)

- Per questo motivo ha senso anche l'espressione `x/-t` che indica di effettuare prima il negato di t e poi la divisione tra x e questo valore.

Le parentesi nelle espressioni

- Non sempre le precedenze di default sono quello che vogliamo. Si può certamente spezzare una sequenza di operazioni in più passi, ma a volte è più chiaro lasciare indicata una espressione unica e usare le parentesi per indicare che una sotto espressione deve essere svolta prima del resto:

```
x = (2 + y) * 3;
```

- In questo caso indichiamo che x deve assumere il valore ottenuto aggiungendo 2 a y e poi moltiplicato per 3.
- Le parentesi non fanno altro quindi che modificare le precedenze. È sempre possibile aggiungere parentesi anche inutili, ma non sempre questo aiuta la leggibilità:

```
(x) = (((2)+(y))*(3));
```

Associatività

- In Matematica le operazioni binarie associative possono essere scritte senza l'uso di parentesi, tanto l'ordine di calcolo non è importante.
- Per le operazioni non associative più comuni è definita una associatività di default: ad esempio $x - y - z$ significa che si deve svolgere prima $x - y$ e poi togliere al risultato z. Lo stesso, anche se meno comune, vale per la divisione: x / y / z significa (x/y)/z. Per evitare ambiguità però si utilizzano linee di lunghezza differente: $\frac{\frac{x}{y}}{z}$. Questa viene detta *associatività a sinistra*.
- Non è però sempre così, dato che x^{y^z} non significa $(x^y)^z$. Infatti questa operazione verrebbe scritta, per le note proprietà dell'elevamento a potenza, come x^{yz}. Quindi la ripetizione dell'elevamento a potenza significa che si vuole eseguire l'operazione $x^{(y^z)}$, evidentemente molto diversa dalla prima. Questa viene detta *associatività a destra*.
- Tutte le operazioni binarie che abbiamo elencato utilizzano la regola dell'associatività a sinistra.
- Ricordate che **in C non esiste un operatore di elevamento a potenza!**

Tipo delle espressioni

- In C ogni dato ha un tipo associato. Anche i risultati delle espressioni hanno un tipo e questo dipende dai dati che sono coinvolti nell'espressione. Ad esempio:

```
int x = 7;
int y = 2;
int z;
...
z = x + y;
```

- In questo caso l'espressione x+y ha un risultato di tipo int. Per cui va tutto bene, ovvero stiamo assegnando un int ad una variabile di tipo int.
- Attenzione quindi alla divisione:

```
z = x / y;
```

- perché in questo caso z assume il valore 3. Infatti la divisione tra numeri interi restituisce solo la parte intera (il resto viene restituito dall'operatore %).

Conversioni aritmetiche consuete

- Che cosa succede quando si mischiano i tipi di dato nelle espressioni?
- Le regole complete dello standard C sono molto complesse e sostanzialmente poco chiare. Limitiamoci a descrivere gli elementi fondamentali.
- La prima cosa da ricordare è che, quando usati in una qualsiasi espressione, char e short vengono promossi a int, mentre unsigned char e unsigned short vengono promossi a unsigned int. Questo comportamento si chiama *integer promotion*.
- Se uno degli argomenti di un operatore è double, l'altro viene convertito in double. Altrimenti se è un float, l'altro viene convertito in float. Infine se è un long long l'altro viene convertito in long long.
- Quando si assegna un dato intero ad uno più grande (in termini di bit) la conversione è quella opportuna (con o senza segno). Quando lo si assegna ad uno più piccolo il risultato viene memorizzato tenendo solo la parte meno significativa.
- Quando si assegna un double o un float ad un intero il risultato perde la parte decimale: 3.7 → 3, -3.7 → -3.

Conversione di tipo

- Consideriamo questo esempio:

```
int x = 5;
int y = 2;
float z;
int main (void) {
        z = x / y;
        return 0;
}
```

- Quanto vale z?
- In questo caso z assume il valore 2.0. Come possiamo forzare la conversione di tipo?
- Una prima soluzione sarebbe quella di usare l'assegnamento:

```
int main (void) {
        z = x;
        z = z / y;
        return 0;
}
```

Conversione di tipo esplicita (cast)

- È però anche possibile forzare la conversione con l'operatore di casting. Questo ha la sintassi seguente:

```
( <tipo-desiderato> ) <dato-da-convertire>
```

- L'operatore è dato da una coppia di parentesi tonde aperta e chiusa con un nome di tipo all'interno. Questo è un operatore unario e viene messo prima del dato da convertire.
- Attenzione! La conversione di tipo ha precedenza uguale a quella degli operatori unari + e -, quindi superiore alle altre operazioni aritmetiche.
- Il caso precedente può quindi essere scritto anche in uno dei seguenti modi:

```
z = ((float)x) / ((float)y);
z = (float)x / (float)y;
z = (float)x / y;
z = x / (float)y;
```

Definire nuovi tipi di dato

- In C è possibile definire un nuovo nome per aggiungere semantica ai tipi di dato. La sintassi è analoga a quella di definizione delle variabili:

typedef *<tipo-di-dato> <nome-del-nuovo-tipo>*;

- Ad esempio potremmo definire un nuovo tipo «anni» che diventi un sinonimo di unsigned char:

```
typedef unsigned char anni;
```

- Si può scrivere allora:

```
anni n;
```

Definire nuovi tipi di dato

- Una volta definiti, i tipi di dato sono disponibili ovunque, anche nelle definizioni di nuovi tipi.

- Oltre ad aggiungere semantica, possono essere utili per scrivere meno. Nella libreria OpenCV per esempio si definisce:

```
typedef unsigned char uchar;
```

- La definizione di nuovi tipi di dato può essere un utile strumento per rendere più generale e riutilizzabile il codice: se scrivo un programma che usa un float e poi voglio passare a double, mi tocca sostituire dappertutto, mentre se avessi definito un tipo «numero», sarebbe bastato cambiarne la definizione.

Parametri delle funzioni

- Abbiamo già detto che le funzioni in C possono avere dei parametri.
- La loro definizione avviene con una sintassi molto simile alla definizione di variabili, ma ogni parametro viene separato dal precedente da una virgola invece che da un punto e virgola, in analogia con la notazione matematica.
- Ad esempio:

```
int funz(int par1, int par2) {
      return par1 + par2;
}
```

- Questa è la definizione di una funzione chiamata *funz* che richiede in ingresso due parametri di tipo *int* che verranno chiamati internamente alla funzione *par1* e *par2*. La funzione restituisce un valore di tipo *int*, in particolare la somma dei valori dei due parametri.

Invocare le funzioni

- Una funzione per essere di qualche utilità deve poter essere utilizzata e questo può essere fatto passando il controllo alla stessa. Questa operazione è detta invocazione di una funzione o più comunemente *chiamata a funzione* (la CALL di ADE8).
- **La chiamata a funzione è un'espressione** ed ha la seguente sintassi:

```
<nome-funzione> ( <argomenti> )
```

- Come vedete, la chiamata a funzione viene effettuata specificandone il nome seguito dalle parentesi tonde aperta e chiusa. Se la funzione ha dei parametri, si possono fornire tra le parentesi degli *argomenti* (espressioni) che verranno usati per inizializzare i parametri, altrimenti non si scrive nulla tra le parentesi, che però **devono** essere indicate.
- Di che tipo è l'espressione «chiamata a funzione»? Del tipo di ritorno della funzione. Il suo valore sarà quello impostato con un return statement nella funzione.
- Vediamo ora qualche esempio di definizione e chiamata a funzione.

Esempio

```
void f(void) {
}

int main(void) {
        f();
        return 0;
}
```

Questa riga che comando C è?

È un «expression statement». L'espressione è composta di un solo operatore, chiamata a funzione.

- In questo programma viene definita la funzione *f* che non riceve parametri e non ritorna nulla. Il main invoca (chiama) la funzione e ritorna.
- Notate che, limitandoci a quello che abbiamo visto finora, le funzioni devono essere definite prima della loro invocazione.
- Precisamente è necessario che una funzione sia *dichiarata* prima di poterla usare. Siccome quando definiamo una funzione ne forniamo anche la dichiarazione, possiamo limitarci a questa modalità, per ora.

Esempio

```
int laRisposta (void) {
      return 42;
}

int x;

int main (void) {
      x = laRisposta();
      return 0;
}
```

- Qui viene chiamata la funzione *laRisposta*, sempre senza parametri, e il valore di ritorno viene assegnato alla variabile *x*.

Esempio

```
int x;

int fai_piu_per(int a, int b, int c) {
    return a + b * c;
}

int main(void) {
    x = fai_piu_per(3, 2, 5);
    return 0;
}
```

- Qui la funzione `main` chiama `fai_piu_per`, con 3 argomenti (`3`, `2`, e `5`) che inizializzano rispettivamente i parametri `a`, `b` e `c`. Il valore di ritorno viene assegnato alla variabile `x`.

Parametri e variabili globali

- Consideriamo di voler realizzare una funzione che raddoppia il valore della variabile passata come parametro.
- Ingenuamente si potrebbe pensare di scrivere:

```
int y = 7;

void raddoppia (int x) {
    x = x*2;
}

int main(void) {
    raddoppia(y);
    return 0;
}
```

← Programma errato di esempio

- Purtroppo dopo l'esecuzione di `raddoppia(y);` la variabile y vale ancora 7. Infatti i parametri sono sostanzialmente delle nuove variabili che «nascono» al momento della chiamata della funzione e «muoiono» alla fine. Quindi la funzione raddoppia correttamente il valore di x, ma questo è una *copia* di y, o meglio una nuova variabile inizializzata con il suo valore.

Parametri e variabili globali

- In C si dice che i parametri vengono *passati per copia*, cioè che non sono riferimenti all'indirizzo della variabile originale, ma una nuova variabile con lo stesso valore.
- Sempre con poco ragionamento si potrebbe pensare di «correggere» il programma così:

```
int y = 7;

void raddoppia (int y) {
    y = y*2;
}

int main(void) {
    raddoppia(y);
    return 0;
}
```

← Programma errato di esempio

- Eseguendo il programma però ci si accorge che nulla è cambiato. Infatti il programma è esattamente quello di prima. Un parametro, anche se ha lo stesso nome di una variabile globale, è un'altra variabile indipendente.

Visibilità (*scope*)

- I parametri quindi sono nuove variabili la cui *visibilità* o *scope* è limitato alla funzione in cui sono definiti. Questo significa che il loro nome può essere usato solo all'interno della funzione e che se esiste una variabile definita esternamente (globale) con lo stesso nome, viene utilizzata quella definita più «vicino».
- Come si può quindi scrivere correttamente il programma desiderato?

```
int y = 7;

int raddoppia (int x) {
    return x*2;
}

int main(void) {
    y = raddoppia(y);
    return 0;
}
```

- Le funzioni devono essere trattate come «scatole nere» in cui entrano dati e da cui esce un risultato.

Assegnamenti composti

- Introduciamo alcuni ulteriori operatori. Questi non sono fondamentali e si può programmare una vita senza mai usarli, ma è importante conoscerli per saper interpretare codice scritto da altri.
- Espressioni tipiche che si incontrano sono nella forma seguente:

```
<variabile> = <variabile> <operatore> <espressione>
```

- Esempi:

```
x = x + 5
y = y * 2
a = a / (x-1)
```

- Vista la frequenza con cui queste espressioni vengono utilizzate, in C sono disponibili operatori compatti per realizzarle.

Assegnamenti composti

- Facendo riferimento alle espressioni precedenti, ecco la loro versione «compatta»:

```
x = x + 5          →    x += 5
y = y * 2          →    y *= 2
a = a / (x-1)      →    a /= x-1
```

- Sono quindi disponibili gli operatori:

```
+=    -=    *=    /=    %=
```

- Queste sono esattamente sostituibili con la versione estesa che utilizza due volte la variabile.

Commenti

- Come in Assembly, anche in C è fondamentale inserire dei commenti nel codice per chiarire il senso delle operazioni che si stanno eseguendo.
- Il modo classico è quello di utilizzare una sequenza di inizio commento `/*` e una di fine `*/`
- Ecco un esempio di codice con commenti:

```
/* La mia variabile */
int y = 7;

/* Funzione che restituisce il doppio
del numero passato come parametro */
int raddoppia (int x) {
    return x*2;
}

/* La prima funzione del programma */
int main(void) {
    y = raddoppia(y);
    return 0;
}
/* Il programma è finito */
```

- Con questi commenti, è possibile scrivere anche lunghe porzioni di testo in cui si va a capo.

Commenti

- Dal C99 in poi, è stato introdotto anche il commento a linea singola, preceduto dalla sequenza di inizio commento `//`
- Il commento termina alla fine della linea:

```
// La mia variabile
int y = 7;

// Funzione che restituisce il doppio
// del numero passato come parametro
int raddoppia (int x) {
    return x*2;
}

// La prima funzione del programma
int main(void) {
    y = raddoppia(y);
    return 0;
}
// Il programma è finito
```

- Con questi commenti, non è possibile andare a capo. O meglio, ogni riga deve essere preceduta dal simbolo di inizio commento.

Operatori di confronto

- Per ora abbiamo visto solo come seguire una sequenza di comandi.
- Per poter invece effettuare delle scelte è necessario saper confrontare tra loro due espressioni. In C questo viene fatto con gli operatori di confronto:

Maggiore	`>`
Maggiore o uguale	`>=`
Minore	`<`
Minore o uguale	`<=`
Uguale	`==`
Diverso	`!=`

- Queste espressioni **sono di tipo int** e valgono 1 se la condizione è verificata, 0 altrimenti.

Ancora sulle espressioni

- Attenzione a non confondere l'operatore di confronto == con l'operatore di assegnamento =. Infatti le due espressioni seguenti

```
x == 5
x = 5
```

  hanno significato molto diverso.

- La prima non *fa* niente, ma confronta x con 5 e *vale* 1 se la variabile x contiene il valore 5, 0 altrimenti
- La seconda invece *fa* qualcosa: mette 5 nella variabile x. Inoltre *vale* 5.
- Come «vale 5»? In C anche gli assegnamenti sono espressioni e possono essere combinati con le regole che valgono per gli altri operatori. Ad esempio il comando:

```
x = 5 * (y+=3);
```

  è valida e significa: somma 3 ad y, poi moltiplica 5 per il valore appena assegnato ad y, poi inserisci il risultato in x.

Associatività e precedenze degli assegnamenti

- Come per le altre espressioni anche gli assegnamenti hanno una loro precedenza e associatività. Ad esempio l'espressione

```
x = y += 3;
```

- potrebbe essere interpretata in due modi:

```
(x = y) += 3;
x = (y += 3);
```

- **L'interpretazione corretta è la seconda**. Infatti gli operatori di assegnamento sono associativi a destra (si svolge prima l'assegnamento più a destra). Questo ha poi senso anche perché un assegnamento è un espressione che vale il valore assegnato, quindi, se y valesse 6, la prima forma verrebbe interpretata come 6 += 3, che ovviamente non ha alcun senso. Per la precisione un assegnamento è un rvalue, quindi non può essere messo a sinistra di un altro assegnamento.

Associatività e precedenze degli assegnamenti

- È però possibile utilizzare una espressione come questa:

```
x = y += 3;
```

- infatti significa: incrementa y di 3 e assegna il risultato a x. Oppure, molto comunemente, si azzerano più variabili assieme con una espressione del tipo:

```
x = y = z = 0;
```

- In questo caso tutte le variabili vengono azzerate.
- Anche se gli assegnamenti possono essere utilizzati in espressioni, è sconsigliabile farlo, dato che non è poi facilmente interpretabile il significato che questa espressione assume.
- Riassumendo, non scrivete espressioni, anche corrette, tipo:

```
x = g-5/(y+=7<z)*(z/=8-k);
```

← NO!!!

Il trasferimento di controllo

- In C esiste un sostituto diretto per il JMP che abbiamo visto in assembly. È il comando:

```
goto <etichetta>;
```

- Il goto trasferisce il controllo alla posizione indicata da una label. Queste si definiscono esattamente come in assembly con un identificatore seguito dal «due punti». Ad esempio:

```
int x, y, z;

int main(void) {
    x = 5;
    goto Matteo;
Marco:
    z = y / x;
    goto Luca;
Matteo:
    y = x + 5;
    goto Marco;
Luca:
    x = y - z;
    return 0;
}
```

La selezione

- Come in assembly, l'utilizzo di salti non ha molta utilità se non è possibile combinarli con delle condizioni. In C esiste un comando che consente proprio questo:

```
if ( <espressione> ) <comando>
```

- Il comando viene eseguito se l'espressione è **diversa da zero**.
- Con quello che abbiamo visto, con if e goto, è già possibile riprodurre le funzionalità dell'assembly molto facilmente.
- Consideriamo ad esempio il problema del calcolo del massimo comun divisore (M.C.D.) tramite l'algoritmo di Euclide che utilizza la sottrazione.
- La logica è abbastanza semplice: dati due numeri naturali non nulli *m* e *n*, se sono uguali l'MCD è il loro valore, altrimenti se m<n vanno scambiati, altrimenti l'MCD è uguale a quello tra m-n e n, quindi basta sostituire a m il valore m-n.
- Utilizzando l'assembly del processore ADE8 si può realizzare come segue:

MCD - ADE8

```
        jmp     main

m:      15
n:      6
t:      0
mcd:    0

main:
        ld      m
        cmpv    0
        je      Store
        ld      n
        cmpv    0
        je      Store
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

MCD - verso il C

```
        jmp     main

m:      15
n:      6
t:      0
mcd:    0

main:
        ld      m
        cmpv    0
        je      Store
        ld      n
        cmpv    0
        je      Store
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

MCD - verso il C

```
m:      15
n:      6
t:      0
mcd:    0
```

```
unsigned int m=15, n=6, t, mcd;
```

```
main:
        ld      m
        cmpv    0
        je      Store
        ld      n
        cmpv    0
        je      Store
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo
Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai
Store:
        st      mcd
Fine:   jmp     Fine
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

main:                          int main(void) {
        ld      m
        cmpv    0
        je      Store0
        ld      n
        cmpv    0
        je      Store
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
        ld      m
        cmpv    0
        je      Store
        ld      n
        cmpv    0
        je      Store
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
```

```
if (m==0)
    goto Store;
```

```
if (n==0)
    goto Store;
```

```
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
    if (m==0)
        goto Store;
    if (n==0)
        goto Store;
Ciclo:
        ld      m
        cmp     n
        je      Store
        jlu     Scambia
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

```
if (m==n)
    goto Store;
```

```
if (m<n)
    goto Scambia;
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
    if (m==0)
        goto Store;
    if (n==0)
        goto Store;
Ciclo:
    if (m==n)
        goto Store;
    if (m<n)
        goto Scambia;
Sottrai:
        ld      m
        sub     n
        st      m
        jmp     Ciclo

Scambia:
        ld      m
        st      t
        ld      n
        st      m
        ld      t
        st      n
        jmp     Sottrai

Store:
        st      mcd

Fine:   jmp     Fine
```

```
m = m - n;
goto Ciclo;
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
    if (m==0)
        goto Store;
    if (n==0)
        goto Store;
Ciclo:
    if (m==n)
        goto Store;
    if (m<n)
        goto Scambia;
Sottrai:
    m = m - n;
    goto Ciclo;
Scambia:
```

```
ld      m
st      t
ld      n
st      m
ld      t
st      n
jmp     Sottrai
```

```
t = m;
m = n;
n = t;
goto Sottrai;
```

```
Store:
        st      mcd

Fine:   jmp     Fine
```

MCD - verso il C

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
    if (m==0)
        goto Store;
    if (n==0)
        goto Store;
Ciclo:
    if (m==n)
        goto Store;
    if (m<n)
        goto Scambia;
Sottrai:
    m = m - n;
    goto Ciclo;

Scambia:
    t = m;
    m = n;
    n = t;
    goto Sottrai;

Store:
```

```
        st      mcd
Fine:   jmp     Fine
```

```
    mcd = m;
    return 0;
}
```

MCD - versione C (brutta!)

```
unsigned int m=15, n=6, t, mcd;

int main(void) {
    if (m==0)
        goto Store;
    if (n==0)
        goto Store;
Ciclo:
    if (m==n)
        goto Store;
    if (m<n)
        goto Scambia;
Sottrai:
    m = m - n;
    goto Ciclo;

Scambia:
    t = m;
    m = n;
    n = t;
    goto Sottrai;

Store:
    mcd = m;
    return 0;
}
```

La programmazione strutturata

- Se questo fosse il modo di programmare in C, non ci sarebbero troppi vantaggi rispetto all'assembly.
- Uno dei problemi principali nella comprensione di questi programmi è questo continuo saltare da una parte all'altra.
- Già negli anni 60/70 apparve un approccio alla programmazione che aveva l'obiettivo di «nascondere» i salti.
- Questo modo di programmare, contrapponendosi alla «sregolatezza» del *goto*, volle imporre una certa struttura al programma e quindi trovare soluzioni standardizzate in grado di sopperire alla necessità di un *goto* esplicito. Semplificando, la conclusione è che il goto non dovrebbe mai essere utilizzato.
- È una regola assoluta? No. Ci sono casi in C in cui un *goto* è meglio che altri accrocchi e quindi, un programmatore che sa quello che sta facendo, può usarlo con ottimi risultati.
- Però dal punto di vista didattico bisogna puntare alla massima semplicità, quindi: **in questo corso è vietato l'uso del goto**.

Eliminare il *goto*: blocco di comandi

- Analizzando il programma che abbiamo scritto, vediamo che l'uso del goto è fatto per **due scopi diversi**. Il primo riguarda l'etichetta Scambia. Siamo costretti a saltare per poter eseguire i tre comandi solo se m<n. Poi saltiamo indietro per continuare con l'esecuzione.
- Quindi per eliminare questa coppia di salti sarebbe necessario poter eseguire in modo condizionato non un solo comando, ma più di uno.
- Per questo in C si introduce il *blocco di comandi* o *comando composto*, ovvero:

```
{ <comando> <comando> ... }
```

- che consente di sostituire ad un solo comando tutti quelli che si vogliono inserire.
- Sfruttando questo comando il programma può diventare:

MCD - meno brutto, ma ancora brutto

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
Ciclo:
    if (m == n)
        goto Fine;

    if (m < n)
        goto Scambia;

Sottrai:
    m = m - n;
    goto Ciclo;

Scambia:
    t = m;
    m = n;
    n = t;
    goto Sottrai;

Fine:
    mcd = m;
    return 0;
}
```

→

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
Ciclo:
    if (m == n)
        goto Fine;

    if (m < n) {
        t = m;
        m = n;
        n = t;
    }

Sottrai:
    m = m - n;
    goto Ciclo;

Fine:
    mcd = m;
    return 0;
}
```

MCD - meno brutto, ma ancora brutto

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
Ciclo:
    if (m == n)
        goto Fine;

    if (m < n)
        goto Scambia;

Sottrai:
    m = m - n;
    goto Ciclo;

Scambia:
    t = m;
    m = n;
    n = t;
    goto Sottrai;

Fine:
    mcd = m;
    return 0;
}
```

→

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
Ciclo:
    if (m == n)
        goto Fine;

    if (m < n) {
        t = m;
        m = n;
        n = t;
    }

Sottrai:
    m = m - n;
    goto Ciclo;

Fine:
    mcd = m;
    return 0;
}
```

Nessuno fa più riferimento all'etichetta Sottrai, quindi possiamo eliminarla.

Eliminare il *goto*: iterazione o ciclo

- Con l'uso del blocco di comadi abbiamo eliminato due necessità di salto esplicito.
- L'altro utilizzo dei salti è quello di ripetere più volte una sequenza di comandi. Per questo scopo, la selezione non è più sufficiente e dobbiamo introdurre l'**iterazione**. Nella sua forma più semplice si scrive:

```
while ( <espressione> ) <comando>
```

- Il comando (o a questo punto il blocco di comandi) continua ad essere eseguito per tutto il tempo in cui l'espressione è **diversa da zero**.
- Utilizzando il *while*, è possibile quindi ripetere un blocco di comandi effettuando una iterazione su un certo insieme di comandi, per tutto il tempo in cui una certa condizione si mantiene vera.
- Bisogna prestare attenzione all'espressione, che rispetto a prima non è la condizioni da verificare per uscire, bensì la condizione da verificare per restare all'interno del ciclo.

MCD

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
Ciclo:
    if (m == n)
        goto Fine;

    if (m < n) {
        t = m;
        m = n;
        n = t;
    }

    m = m - n;
    goto Ciclo;

Fine:
    mcd = m;
    return 0;
}
```

→

```
unsigned int m = 15, n = 6, t, mcd;

int main(void) {
    while (m != n) {
        if (m < n) {
            t = m;
            m = n;
            n = t;
        }
        m = m - n;
    }
    mcd = m;
    return 0;
}
```

Ulteriori elementi di programmazione strutturata

- Grazie all'introduzione del concetto di blocco di comandi e del comando *while*, è possibile realizzare interessanti funzionalità molto complesse, utilizzando blocchi elementari.
- Vediamo ad esempio come potremmo incapsulare la funzionalità appena vista in una vera e propria funzione.
- Quello che si deve fare è dare un nome alla funzione, specificare quali sono i parametri di input (m e n) e qual è l'output.
- In particolare possiamo osservare che i valori sono naturali, quindi utilizziamo `unsigned int` sia per i parametri che per il tipo di ritorno. Inoltre se m o n sono nulli, ritorniamo zero.

Ulteriori elementi di programmazione strutturata

```
unsigned int m = 15, n = 6, t, mcd;

unsigned int MCD (unsigned int m, unsigned int n) {
    if (m == 0)
        return 0;
    if (n == 0)
        return 0;
    while (m != n) {
        if (m < n) {
            t = m;
            m = n;
            n = t;
        }
        m -= n;
    }
    return m;
}

int main(void) {
    mcd = MCD(m, n);
    return 0;
}
```

- In questa funzione, effettivamente tutto è piuttosto contenuto, ma rimane la necessità di definire esternamente una variabile t per consentire lo scambio delle variabili.
- Per isolare effettivamente la funzione, questa variabile temporanea dovrebbe essere «interna» alla funzione.
- In C per questo scopo si utilizzano le *variabili locali*. Ogni blocco di comandi infatti consente di dichiarare variabili oltre che comandi.

Variabili locali

- All'inizio di un qualsiasi blocco di comandi (quelli utilizzati nella definizione di funzione o in un if o in un while) è possibile inserire definizioni di variabili.
- Per il compilatore la variabile «nasce» all'inizio del blocco, è *visibile* solo all'interno del blocco e termina di esistere (la memoria può essere utilizzata per altri scopi) alla fine del blocco.
- Il blocco di comandi viene quindi esteso così:

```
{
        <definizione>
        <definizione>
        ...

        <comando>
        <comando>
        ...
}
```

Nello standard conosciuto come C89 (il primo ANSI C), le definizioni devono essere obbligatoriamente **prima** di qualsiasi comando!
Dal C99 questo vincolo è scomparso.

Uso di variabili locali

```
unsigned int MCD (unsigned int m, unsigned int n) {
    if (m == 0)
        return 0;
    if (n == 0)
        return 0;
    while (m != n) {
        if (m < n) {
            unsigned int t = m;
            m = n;
            n = t;
        }
        m -= n;
    }
    return m;
}

int main(void) {
    unsigned int m = 15;
    unsigned int n = 6;
    unsigned int mcd;

    mcd = MCD(m, n);

    return 0;
}
```

- È importante notare che ora le variabili locali possono sostituire completamente quelle globali, che rimangono solo per un utilizzo sporadico e molto particolare.
- È sempre possibile fare a meno di variabili globali, ma in alcuni casi possono essere comode per evitare eccessivi passaggi di parametri.
- Notate che la variabile t è stata definita solo all'interno del blocco in cui serviva.

Espressioni logiche

- Nell'esempio precedente per ritornare 0 in caso di m==0 o di n==0 abbiamo dovuto utilizzare due comandi if. In realtà è possibile in C costruire espressioni logiche grazie agli *operatori logici*.
- Gli operatori logici sono i seguenti:

`<ex1> && <ex2>`	`AND logico`	restituisce 1 se entrambe le espressioni sono diverse da 0
`<ex1> || <ex2>`	`OR logico`	restituisce 1 se almeno una delle espressioni è diversa da 0
`!<ex>`	`NOT logico`	restituisce 1 se l'espressione è uguale a 0

- Negli altri casi restituiscono 0.
- Questi operatori hanno una priorità molto bassa, certamente più bassa di tutti gli operatori di confronto e degli operatori aritmetici, quindi le espressioni «semplici» sono interpretate come ci aspetteremmo anche in assenza di parentesi.

Espressioni logiche

```
unsigned int MCD (unsigned int m, unsigned int n)
{
    if (m == 0 || n == 0)
        return 0;
    while (m != n) {
        if (m < n) {
            unsigned int t = m;
            m = n;
            n = t;
        }
        m -= n;
    }
    return m;
}

int main(void)
{
    unsigned int m = 15;
    unsigned int n = 6;
    unsigned int mcd;

    mcd = MCD(m, n);
    return 0;
}
```

- Ecco come si potrebbe scrivere l'espressione precedente utilizzando l'operatore logico OR.
- Attenzione che gli operatori logici AND e OR sono scritti con due simboli, mentre ne esistono altri che utilizzano una sola «&» e una sola «|». **Non confondetevi!**

Ancora sulla selezione

- Parlando del comando if, abbiamo visto come sia possibile utilizzarla per decidere se eseguire una parte di codice al verificarsi di una condizione.
- Spesso però è utile poter specificare che cosa fare in caso contrario, ovvero se la condizione indicata non è vera, o meglio **se l'espressione di controllo vale 0**.
- La versione completa dell'if è quindi la seguente:

```
if ( <espressione> )
        <comando1>
else
        <comando2>
```

- comando2 viene eseguito se l'espressione vale zero. Per quanto detto, sia comando1, sia comando2 possono essere blocchi di comandi.
- Possiamo realizzare allora una versione diversa dell'MCD: se m>n sostituiamo a m il valore di m-n, altrimenti sostituiamo a n il valore di n-m.

Ancora sulla selezione: l'else

```
unsigned int MCD (unsigned int m, unsigned int n)
{
    if (m == 0 || n == 0)
        return 0;
    while (m != n) {
        if (m < n)
            n -= m;
        else
            m -= n;
    }
    return m;
}

int main(void)
{
    unsigned int m = 15;
    unsigned int n = 6;
    unsigned int mcd;

    mcd = MCD(m, n);
    return 0;
}
```

- Ecco come si potrebbe scrivere il programma che calcola l'MCD utilizzando l'else.

Altri costrutti per l'iterazione

- In generale è possibile descrivere una iterazione come la composizione di quattro diverse fasi:
  - Init: inizializzazione di variabili di controllo
  - Check: controllo della condizione di ripetizione del ciclo
  - Body: insieme dei comandi da ripetere
  - Update: aggiornamento delle variabili di controllo del ciclo
- Il ciclo while è tipicamente costruito così:

```
Init
while ( Check ) {
        Body
        Update
}
```

- Esistono casi in cui si vuole eseguire il ciclo almeno una volta e poi verificare se è necessario ripeterne l'esecuzione.

do...while

- In questi casi si può sfruttare il comando do...while:

```
Init
do {
        Body
        Update
} while ( Check );
```

- Notate che a differenza del ciclo precedente il while deve essere seguito da un punto e virgola.
- Questo comando non è frequentissimo, ma può servire quando è necessario leggere dati da una sorgente di cui non si conosce il contenuto, come ad esempio un file o la tastiera.

Esempio di do...while

- Consideriamo di voler calcolare la radice quadrata di un numero utilizzando l'algoritmo di Newton per gli zeri di una funzione.
- Si vuole cioè calcolare $x = \sqrt{a}$ trovando lo zero di $f(x) = x^2 - a$.
- Per fare questo si può partire con l'approssimazione $x_0 = a$ e poi avvicinarsi alla soluzione con $x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right)$.
- Per fare questo memorizziamo il valore precedente della *x* in una variabile temporanea *t* e poi aggiorniamo il valore di *x* usando *t*.
- Possiamo terminare (grazie alle proprietà di questo algoritmo) quando *x* è uguale a *t*.
- Dato che all'inizio non esiste un valore precedente di *x*, la variabile *t* dovrebbe essere inizializzata a caso con un valore impossibile da ottenere inizialmente.
- Una soluzione più elegante è quella di utilizzare un ciclo do...while.

Radice quadrata

```
double sqrt (double a)
{
    double t, x = a;

    if (x <= 0.)
        return 0.;

    do {
        t = x;
        x = 0.5 * (t + a / t);
    } while (x != t);

    return x;
}

int main(void)
{
    double d = sqrt(137.0);

    return 0;
}
```

- La funzione è molto semplice, nonostante l'algoritmo sia matematicamente molto sofisticato!
- Dopo un primo controllo sul fatto che l'argomento della funzione debba essere positivo l'iterazione procede salvando il valore di x e calcolandone il successivo.
- Questa funzione gestisce in modo semplicistico gli errori dovuti ad argomenti negativi, ritornando zero.
- Notate anche la costante «0.» per indicare lo zero in double.

Somma di numeri

- Supponiamo di voler calcolare il valore $x = \sum_{i=1}^{n} i$, ovvero la somma dei primi *n* numeri naturali positivi.
- Ignoriamo il fatto che si possa risolvere il problema facilmente con la sola espressione $x = \frac{n(n+1)}{2}$.
- Si può affrontare il problema scrivendo:

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;
    unsigned int i;

    i = 1;
    while (i <= n) {
        x = x + i;
        i = i + 1;
    }

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);
    return 0;
}
```

- È molto facile vedere in questo programma tutti gli elementi classici dei cicli, ovvero Init, Check, Body e Update.
- Il C per queste tipologie di cicli prevede una sintassi specifica detta *ciclo for*.

Il ciclo for

- La sintassi del comando for è la seguente:

```
for ( Init ; Check ; Update )
      Body
```

- Come per tutti i costrutti precedenti è possibile utilizzare o meno le parentesi graffe per il body a seconda che sia un solo o più comandi.
- Init, Check e Update sono tre espressioni e possono o meno essere indicate, ma i due punti e virgola all'interno delle parentesi sono obbligatori e non possono essere omessi in alcun caso!
- Il pregio del ciclo for è che è più difficile dimenticarsi di effettuare l'inizializzazione della variabile o di aggiornarne il valore.
- Vediamo come diventerebbe il ciclo precedente utilizzando il for.

Somma di numeri col for

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;
    unsigned int i;

    for (i = 1; i <= n; i = i + 1)
        x = x + i;

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Con il ciclo for è possibile realizzare cicli molto compatti, in cui tutta la parte di controllo è concentrata nell'intestazione del ciclo ed è facilmente analizzabile senza dover saltare nel codice all'inizio e alla fine del ciclo.
- Notate come, per ora, non sia possibile assegnare più di una variabile nella parte di Init del ciclo e come l'espressione di update sia molto prolissa.
- Vediamo come compattare ulteriormente queste espressioni.

Somma di numeri col for (più compatto)

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;
    unsigned int i;

    for (i = 1; i <= n; i += 1)
        x += i;

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Utilizzando l'operatore += abbiamo reso ancora più breve la funzione precedente.
- L'operazione +=1 (incremento) è però molto comune e gli ideatori del C hanno ritenuto opportuno introdurre operatori appositi.
- Questi (come anche quelli di decremento) sono molto sofisticati, soprattutto se utilizzati all'interno di altre espressioni.
- Vediamo la loro sintassi.

Incremento e decremento

- Esistono 4 operatori unari di incremento e decremento di variabili. Consideriamo un lvalue generico *x*; le espressioni sono le seguenti:

`++x`	`preincremento`	x viene incrementato e l'espressione vale x+1.
`x++`	`postincremento`	x viene incrementato e l'espressione vale x.
`--x`	`predecremento`	x viene decrementato e l'espressione vale x-1.
`x--`	`postdecremento`	x viene decrementato e l'espressione vale x.

- Attenzione al significato di queste piccole differenze. Consideriamo ad esempio questa espressione (con y=4 e z=5):

```
x = ++y*z++;
```

- Qui bisogna stare attenti, perché ++y vale 5, z++ vale 5 e quindi alla fine x=25, y=5, z=6.
- In generale l'utilizzo di preincrementi e postincrementi in espressioni complesse è fonte di confusione.

Undefined behavior

- Oltre a generare confusione, a volte per smania di essere cool o geek, qualcuno si lancia nella composizione di espressioni complicatissime, arrivando a scrivere cose tipo:

```
a=b%c-(--a);
```

- Che cosa vale a? Non è importante sapere i valori di a, b o c, tanto non si può sapere comunque. Infatti modificare il valore di *a* nella stessa espressione più volte è un comportamento che nello standard del C non è definito.
- Quindi un compilatore potrebbe produrre codice che fa 42, un altro invece formattarvi l'hard disk (improbabile e decisamente poco saggio da parte di chi ha scritto il compilatore, ma non vietato dallo standard).
- Morale: non modificate **MAI** il valore di una variabile più volte nella stessa espressione.

Somma di numeri col for

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;
    unsigned int i;

    for (i = 1; i <= n; ++i)
        x += i;

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Possiamo allora sostituire l'operazione +=1 con un incremento.
- Invece che ++i, avremmo potuto utilizzare i++ in modo indifferente.

- Quante volte viene eseguito il *corpo* del for (il comando x+=i;)?

- Quante volte viene verificata la condizione i<=n?

- Quanto vale i alla fine del ciclo?

Somma di numeri col for

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;
    unsigned int i;

    for (i = 1; i <= n; ++i)
        x += i;

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Possiamo allora sostituire l'operazione +=1 con un incremento.
- Invece che ++i, avremmo potuto utilizzare i++ in modo indifferente.

- Quante volte viene eseguito il *corpo* del for (il comando x+=i;)?
  n volte
- Quante volte viene verificata la condizione i<=n?
  n+1 volte
- Quanto vale i alla fine del ciclo?
  n+1

Un for più flessibile

- Il C99 ha introdotto una ulteriore variante del for, che consente di avere nella sezione Init del for una definizione di variabile, oltre che una espressione.
- Cosa vuol dire? Consideriamo il codice seguente:

```
unsigned int i;
for (i = 1; i <= n; ++i)
    x += i;
```

- Se la variabile i ci serve solo per effettuare il for, potremmo scrivere:

```
for (unsigned int i = 1; i <= n; ++i)
    x += i;
```

- In questo caso la variabile i è visibile solo all'interno del ciclo for, come se avessimo scritto una cosa di questo tipo:

```
{
    unsigned int i;
    for (i = 1; i <= n; ++i)
        x += i;
}
```

Somma di numeri col for (finale)

```
unsigned int somma_primi (unsigned int n)
{
    unsigned int x = 0;

    for (unsigned int i = 1; i <= n; ++i)
        x += i;

    return x;
}

int main(void)
{
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Ecco allora la versione finale della funzione che somma i primi n numeri interi.
- La variabile i è definita solo per il tempo necessario al suo utilizzo.

Ancora operatori

- Vediamo alcuni operatori non fondamentali, ma che devono essere conosciuti per saper interpretare il codice scritto da altri.
- Una struttura molto comune è la seguente:

```
if (<A>)
      <azione X che usa> <B>
else
      <azione X che usa> <C>
```

- Notate che in base alla condizione <A> viene eseguita una certa azione X (la stessa) in un caso con <B>, nell'altro con <C>. Quindi se avessimo una espressione che può valere <B> o <C> a seconda di <A> potremmo scrivere:

```
<azione X che usa> [<B> o <C> a seconda di <A>]
```

Operatore condizionale

- L'espressione è ottenuta con l'operatore ternario condizionale:

```
<A> ? <B> : <C>
```

- Un esempio ne chiarisce l'uso:

```
if (x>0)
      v = b+1;      →      v = x>0 ? b+1 : f/3 ;
else
      v = f/3;
```

- Fondamentalmente questo operatore serve a scrivere espressioni molto complesse in poco spazio, ma un consiglio personale è quello di evitarlo, perché un if è più chiaro e non ha nulla di meno.

Operatore virgola

- L'operatore a più bassa priorità del C (anche più bassa dell'assegnamento) è l'operatore virgola:

```
<A> , <B>
```

- Questa espressione esegue <A>, poi <B> e vale <B>. Ma a che cosa serve? Nella pratica solo per consentire di inizializzare o incrementare più di una variabile nel for.
- Anche qui il consiglio è di non usare questo operatore, perché è possibile scrivere cose difficilmente comprensibili come:

```
x = y , z;
```

- Questo comando è composto di un assegnamento e del valore di z. In pratica l'assegnamento avviene prima della virgola, a x viene assegnato il valore di y e la z viene letta e non usata! Vediamo un caso più sensato:

Somma di numeri col for

```
unsigned int somma_primi (unsigned int n) {
    unsigned int x, i;

    for (x = 0, i = 1; i <= n; ++i)
        x += i;

    return x;
}

int main(void) {
    unsigned int x;
    x = somma_primi(10);

    return 0;
}
```

- Notate che abbiamo inizializzato x a 0 e y a 1 nella parte di inizializzazione del for.
- Senza l'operatore virgola questo non sarebbe stato possibile.
- Ma si poteva vivere e programmare benissimo anche senza!

- In particolare bisogna evitare di scrivere cose orribili come:

```
unsigned int somma_primi (unsigned int n) {
    unsigned int x, i;
    for (x = i = 0; x += i, i++ < n; );
    return x;
}
```

- NOOOO!!!!!

Operatori per lavorare con i bit

- Molto spesso si rende necessario lavorare bit a bit e per questo servono le usuali operazioni tra bit: AND, OR, XOR e NOT. Gli operatori corrispondenti in C sono:
  - AND: **&**
  - OR: **|**
  - XOR: **^** ← Notate che non è l'elevamento a potenza!!!
  - NOT: **~**
- Dei primi tre esiste anche la forma dell'assegnamento composto:
  - AND: **&=**
  - OR: **|=**
  - XOR: **^=**
- Infine è possibile spostare tutti i bit dei tipi interi verso destra o verso sinistra, quindi dividere o moltiplicare per una potenza del 2 con gli operatori di shift a destra (**>>**) di shift a sinistra (**<<**) ed i corrispondenti assegnamenti composti: **>>=** e **<<=**.

Esempi di operazioni bit a bit

```
int main(void)
{
     unsigned char x = 0x3a;
     unsigned char y = 0xc8;

     unsigned char a = x & y;
     unsigned char b = x | y;
     unsigned char c = x ^ y;
     unsigned char d = ~x   ;

     x &= 0xf0;
     y |= 0xf0;
     a ^= 0xaa;

     b = x >> 4;
     c = y << 4;

     x >>= 1;
     y <<= 2;

     return 0;
}
```

- Provate a svolgere queste operazioni e verificate nella pagina seguente il risultato.

Esempi di operazioni bit a bit

```
int main(void)
{
     unsigned char x = 0x3a;  /* 0011.1010 */
     unsigned char y = 0xc8;  /* 1100.1000 */

     unsigned char a = x & y; /* 0000.1000 -> 0x08 */
     unsigned char b = x | y; /* 1111.1010 -> 0xfa */
     unsigned char c = x ^ y; /* 1111.0010 -> 0xf2 */
     unsigned char d = ~x   ; /* 1100.0101 -> 0xc5 */

     x &= 0xf0; /* 0011.0000 -> 0x30 */
     y |= 0xf0; /* 1111.1000 -> 0xf8 */
     a ^= 0xaa; /* 1010.0010 -> 0xa2 */

     b = x >> 4; /* 0000.0011 -> 0x03 */
     c = y << 4; /* 1000.0000 -> 0x80 */

     x >>= 1; /* 0001.1000 -> 0x18 */
     y <<= 2; /* 1110.0000 -> 0xe0 */

     return 0;
}
```

- Notate che abbiamo scelto di utilizzare un tipo unsigned, dato che non è sempre chiaro che cosa debba succedere con gli shift quando i dati sono con segno.
- Gli operatori binari, in pratica, vanno utilizzati solo per dati unsigned.
- Nel caso di dati signed, lo standard dice che il risultato dipende dall'implementazione dei numeri negativi.