Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Le funzioni

Catia Prandi - catia.prandi2@unibo.it

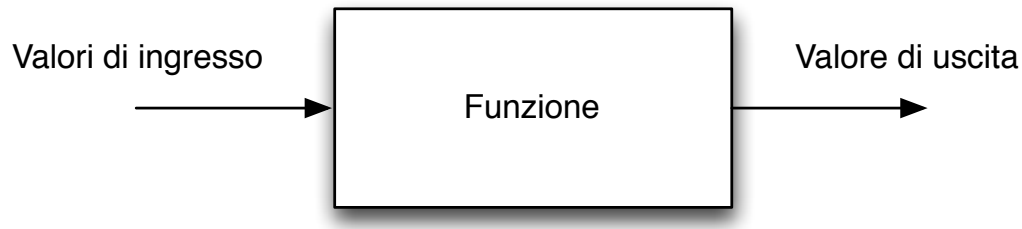
Credit: **Pietro Di Lena**

```c
extern int home;

int man() {
  goto pub;
  pub: goto drink;
  drink: goto bathroom;
  bathroom: goto drink;
  return home;
}
```

Le funzioni

- Una **funzione** è un particolare costrutto sintattico che permette di definire un blocco di codice che assolve un compito ben preciso.
- Sinonimi del termine funzione: procedura, routine, subroutine, sottoprogramma.
- Possiamo vedere una funzione come una **black box**, che prende in ingresso alcuni valori e produce in output un valore.



- Esistono funzioni che non prendono nessun valore in ingesso e/o che non producono alcun valore di uscita.
- Le funzioni permettono di **scomporre** un problema complesso in moduli più semplici.
- Se implementate correttamente, possono essere **riutilizzate** per la risoluzione di problemi differenti.
- Rendono più chiaro il codice e ne facilitano la **manutenzione**.

Le funzioni in C

- Un programma in C è costituito da un insieme di funzioni.
- Il `main()` è una funzione.
  - Ogni programma C deve contenere esattamente una funzione `main()`.
- In particolare, la funzione `main()` è il punto di ingesso di ogni programma C.
- Abbiamo già utilizzato le *funzioni di libreria* del C, come, ad esempio, la `printf()`.
- Vediamo adesso quali sono le regole sintattiche che ci permettono di *costruire* funzioni.
- Nel dettaglio, per il momento vedremo:
  - come **dichiarare** funzioni;
  - come **definire** funzioni;
  - come **invocare** funzioni.

Dichiarazione di funzione

- Una **dichiarazione** di una funzione, detta anche **prototipo**, specifica:
  1. il **tipo di dato restituito** dalla funzione;
  2. il **nome** (identificatore) della funzione;
  3. i tipi ed i nomi (opzionali) degli eventuali **argomenti** della funzione.

  `<Tipo> <NomeFunzione>(<Tipo1> [<Arg1>], .., <TipoN> [<ArgN>]);`
- Una funzione **dovrebbe** essere dichiarata prima di poter essere invocata.
  - A differenza di quanto abbiamo visto per le variabili, non è sempre necessario dichiarare una funzione prima di poterla invocare.
  - Per la precisione, non è necessario se il tipo di ritorno è `int` (vedremo perchè).

```
1  // Calcola il quadrato di un intero
2  int square(int x);
3
4  // Calcola e stampa il quadrato di un intero
5  void print_square(int x);
6
7  // Calcola il numero di secondi a partire da un orario
8  // composto da ora (h), minuti (m) e secondi (s)
9  int seconds(int h, int m, int s);
10
11 // Restituisce il massimo tra i due argomenti
12 double max(double x, double y);
13
14 // Stampa informazioni di copyright
15 void copyright(void);
```

Dichiarazione di funzione: motivazioni

- La dichiarazione di funzione ci permette di capire unicamente come **invocarla**.
  - Specifica quali e quanti argomenti prende in input, oltre al tipo del valore di ritorno.
- **Punto di vista del programmatore**:
  - molto spesso risulta più utile capire **cosa calcola** una funzione (e quindi come deve essere invocata) piuttosto che capire **come calcola** (e quindi come è implementata internamente),
  - ad esempio: vogliamo imparare ad utilizzare correttamente la funzione di libreria `printf()`, siamo meno interessati a capire come sia stata implementata,
  - il prototipo e i relativi commenti dovrebbero essere di aiuto a capire cosa calcola una funzione e come deve essere invocata.
- **Punto di vista del compilatore**:
  - la dichiarazione di funzione permette al compilatore di verificare che la funzione venga invocata correttamente;
  - lo scopo è quello di evitare comportamenti semanticamente non definiti.
- Nota: i prototipi di funzioni definite in *librerie esterne* al file sorgente sono generalmente contenuti nei *file header*. Ad esempio, il prototipo della funzione `printf()` è contenuto nel file header `stdio.h`.

Dichiarazione di funzione: alcune precisazioni aggiuntive

- La funzione `main()` non necessita di un prototipo.
- Il tipo di dato `void` può essere omesso se la funzione non prende argomenti. I due prototipi seguenti sono equivalenti:

```
void copyright(void);
```

```
void copyright();
```

  E' buona norma specificare il `void` per evitare situazioni problematiche (che vedremo).
- E' possibile omettere i nomi delle variabili-argomento nella definizione di funzione.

```
// Calcola il numero di secondi a partire da un orario
// composto da ora, minuti e secondi.
int seconds(int, int, int);
```

  E' buona norma specificare i nomi degli argomenti, soprattutto quando possono aiutare a chiarire come dovrebbe essere invocata la funzione. Nel precedente esempio, come facciamo a capire quale dei tre è l'argomento `ora` (risp. `minuti` e `secondi`)? Il seguente prototipo è maggiormente esplicativo.

```
int seconds(int h, int m, int s);
```

Definizione di funzione

- Una **definizione** di funzione è costituita da due parti:
  1. **Dichiarazione della funzione**. Come per i prototipi, è necessario specificare:
     - tipo di dato restituito;
     - nome della funzione;
     - argomenti della funzione (i nomi degli argomenti non sono opzionali).
  2. **Corpo della funzione**. Porzione di codice racchiusa tra parentesi graffe che comprende le seguenti componenti opzionali:
     - dichiarazioni di variabili locali;
     - istruzioni;
     - istruzione `return` con eventuale valore di ritorno.

```
<Tipo> <NomeFunzione>(<Tipo1> <Arg1>, .., <TipoN> <ArgN>) {
  [<Dichiarazioni>]
  [<Istruzioni>]
  [return [<Espressione>];]
}
```

- Regola sintattica dello standard ISO C89: le dichiarazioni **devono** precedere le istruzioni (questa regola sintattica vale in generale per i blocchi di istruzioni).
- Alcune osservazioni:
  - una definizione di funzione implicitamente include anche una dichiarazione;
  - non bisogna inserire il ; dopo la parte dichiarativa.

Definizione di funzione: esempi di implementazione dei prototipi

- Funzione che calcola il quadrato di un intero.

```
1 int square(int x) {
2   return (x*x);
3 }
```

- Funzione che calcola e stampa il quadrato di un intero.

```
1 void print_square(int x) {
2   printf("%d^2 = %d\n",x,square(x));
3 }
```

- Funzione che calcola il numero di secondi in un orario.

```
1 int seconds(int h, int m, int s) {
2   return (3600*h+60*m+s);
3 }
```

- Funzione che calcola il massimo tra due numeri.

```
1 double max(double x, double y) {
2   return (x>=y ? x : y);
3 }
```

- Funzione che stampa informazioni di copyright.

```
1 void copyright(void) {
2   printf("Copyright (c) 2015\n");
3 }
```

Definizione di funzione: alcune precisazioni aggiuntive

- Tutte le parti del corpo di una funzione sono opzionali, escluse le parentesi graffe. Per quanto poco utili, è possibile definire funzioni che non contengano istruzioni.

```
1 void do_nothing(void) {
2 }
```

- E' possibile dichiarare una funzione all'interno del corpo di una funzione.

```
1 void print_square(int x) {
2   int square(int x); // Valido: dichiarazione.
3
4   printf("%d^2 = %d\n",x,square(x));
5 }
```

- Non è possibile definire una funzione all'interno del corpo di una funzione.

```
1 void print_square(int x) {
2   int square(int x) { // Errore: definizione
3     return (x*x);
4   }
5
6   printf("%d^2 = %d\n",x,square(x));
7 }
```

Definizione di funzione: esempi di funzioni matematiche

- Funzione per calcolare la potenza tra due interi. Nota: si assume $0^0 = 1$.

```
1  // Calcola la potenza di x elevato ad y
2  double power(int x, int y) {
3    double pow = 1;
4    int       i = y>=0?y:-y;
5
6    for(; i>0; i--) pow *= x;
7
8    if(y>=0) return pow;
9    else     return 1/pow;
10 }
```

- Funzione per calcolare il fattoriale. Nota: prende in input solo interi positivi.

```
1  // Calcola il fattoriale di un intero positivo x.
2  unsigned long fact(unsigned int x) {
3    unsigned long res=1;
4    int  i;
5
6    for(i=2; i<=x; i++) res *= i;
7    return res;
8  }
```

Le funzioni variadiche

- Il linguaggio C permette di definire funzioni **variadiche**: funzioni che hanno un numero variabile di argomenti.
- Le funzioni `printf()` e `scanf()` sono esempi di funzioni variadiche.
- I prototipi delle funzioni variadiche sono definiti utilizzando la notazione **ellissi**:

```
<Tipo> <NomeFunzione>(Tipo1 <Arg1>, ...);
```

- L'ellissi `...` (tre puntini) indica che la funzione è variadica.
- E' necessario specificare almeno un argomento fisso (il primo).
- La funzione può essere definita con altri argomenti fissi.
- L'ellissi deve essere essere specificata come ultimo argomento nella lista.
- Per accedere agli argomenti nel corpo della funzione si utilizzano macro definite nella libreria `stdarg.h`.
- Le funzioni variadiche sono un argomento avanzato che vedremo (eventualmente) come approfondimento.

Invocazione di funzione

- L'**invocazione** (o **chiamata**) di funzione è un'istruzione che permette l'esecuzione della funzione.
- Semantica di invocazione di una funzione:
  1. Quando una funzione `F()` viene invocata nel corpo di una funzione `G()`, l'esecuzione di `G()` è sospesa e si passa ad eseguire le istruzioni contenute in `F()`.
  2. Quando la funzione `F()` termina la sua esecuzione, si prosegue con le istruzioni di `G()` successive al punto in cui `F()` è stata chiamata.
- Una funzione che invoca se stessa viene detta **funzione ricorsiva**.
- Approfondiamo di seguito i tre seguenti argomenti.
  - Valore di ritorno di una funzione.
  - Visibilità di una funzione.
  - Passaggio dei parametri a funzione.

Valore di ritorno di una funzione

- Una funzione **ritorna** (o **restituisce**) un valore tramite il costrutto (parola chiave) `return`.

```
return [<Espressione>];
```

- L'esecuzione dell'istruzione `return` causa la terminazione della funzione e il **ritorno** del controllo al chiamante.
- Possiamo utilizzare il valore di ritorno di una funzione all'interno di espressioni.
- In C una funzione non può ritornare molteplici valori e non può ritornare funzioni.
- Le parentesi tonde per proteggere l'espressione di ritorno sono opzionali:

```c
double max(double x, double y) {
  return x>=y ? x : y;
}
```

```c
double max(double x, double y) {
  return (x>=y ? x : y);
}
```

- Se il valore di ritorno di una funzione è `void`, possiamo evitare di utilizzare il costrutto `return`, oppure possiamo utilizzarlo senza parametro.

```c
void copyright(int y) {
  if(y<0)
    return;
  else
    printf("Copyright (c) %d\n",y);
}
```

```c
void copyright(int y) {
  if(y>=0)
    printf("Copyright (c) %d\n",y);
}
```

Valore di ritorno: precisazioni aggiuntive 1/4

```
1  #include <stdio.h>
2
3  // Calcola il fattoriale di un intero positivo x.
4  unsigned long fact(unsigned int x) {
5    unsigned long res=1;
6    int  i;
7
8    for(i=2; i<=x; i++) res *= i;
9    return res;
10 }
11
12 int main() {
13   int x;
14   unsigned long res;
15   printf("Inserici un intero positivo: ");
16   scanf("%d",&x);
17
18   if(x<0) printf("Errore: valore negativo\n");
19   else {
20     res = fact(x); // Espressione:chiamata a funzione e assegnamento
21     printf("%d! = %lu\n",x,res);
22   }
23   return 0;
24 }
```

- Salviamo in `res` il valore di ritorno di `fact()` (riga 20).
- Il tipo di `res` deve coincidere col tipo di ritorno di `fact()` per evitare typecasting.

Valore di ritorno: precisazioni aggiuntive 2/4

```
1  #include <stdio.h>
2
3  void author() {
4    printf("Author: Pippo De Pippis\n");
5  }
6
7  int copyright(int y) {
8    if(y<0)
9      return 1; // Errore: anno negativo
10   else {
11     printf("Copyright (c) %d\n",y);
12     return 0; // Ok
13    }
14 }
15
16 int main() {
17   int x;
18
19   x = author(); // Errore! author() non restituisce un valore.
20   copyright(1932);
21   return 0;
22 }
```

- **Possiamo ignorare** il valore di ritorno di una funzione (riga 20).
- **Dobbiamo ignorare** il valore di ritorno void (riga 19). Errore di sintassi.

Valore di ritorno: precisazioni aggiuntive 3/4

- Le funzioni C permettono di restituire un solo valore.

```
1 int interval(int x) {
2   if(x<0) return (x,-x);
3   else    return (-x,x);
4 }
```

La funzione `interval()` apparentemente restituisce la coppia di valori (`-|x|`, `|x|`), dato in input un intero `x`. In realtà, utilizza l'operatore virgola e calcola quindi il valore assoluto `|x|` di `x`.

- Se non c'è coerenza tra il tipo di ritorno e il tipo dell'espressione passata a `return`, si applicano le regole di conversione implicita .

```
1 // Ritorna la costante Pi-greco
2 int pi() {
3   return 3.14;
4 }
```

La funzione `pi()` ritornerà il troncamento ad `int` (3) della costante `double` (`3.14`).

Valore di ritorno: precisazioni aggiuntive 4/4

- E' sintatticamente permesso (ma poco pulito) far ritornare valori a funzioni con tipo di ritorno `void`. Tale valore viene ignorato.

```
1 void do_nothing() {
2   return 0;
3 }
```

- E' sintatticamente permesso terminare l'esecuzione di una funzione non `void` senza ritornare un valore. In tal caso, il valore restituito non è predicibile, causando un comportamento non definito del programma.

```
1 int max(int x, int y) {
2   if(x>=y) return x;
3 }
```

  Se in primo argomento è minore del secondo, non possiamo predire quale sarà il valore di ritorno della funzione.

- L'istruzione `return` causa la terminazione immediata della funzione.

```
1 void copyright(void) {
2   return;
3   // L'istruzione successiva non    mai eseguita
4   printf("Copyright (c) 2015\n");
5 }
```

Visibilità di una funzione

- La **visibilità** di una funzione dipende dal punto in cui è dichiarata/definita.
- Una funzione è visibile da tutte le funzioni nello stesso file che seguono la sua dichiarazione/definizione.
- Se una funzione è visibile al momento della chiamata il compilatore può eseguire dei check sulla consistenza tra definizione della funzione e modalità di chiamata della stessa.
- E' possibile dichiarare (ma non definire) una funzione `F()` all'interno di blocco di istruzioni di una funzione `G()`. In questo caso, `F()` sarà invocabile da tutte le istruzioni nel blocco di `G()` che seguono la dichiarazione.
  - Nota: costrutto sintattico grammaticalmente corretto ma poco usato e poco elegante.

Visibilità delle funzioni: esempio 1/5

```c
// La dichiarazione della funzione printf()
//     contenuta nel file stdio.h
#include<stdio.h>

int main() {
  // Chiamata di funzione
  printf("Hello, World!");
  return 0;
}
```

- La funzione `printf()` è visibile dalla funzione `main()`.
- La dichiarazione della `printf()` è contenuta nel file `stdio.h`, che viene incluso dal preprocessore nel file sorgente.
- L'inclusione del contenuto di `stdio.h` avviene nel punto in cui compare la direttiva `#include`, quindi prima della definizione del `main()`.

Visibilità delle funzioni: esempio 2/5

```
1  #include<stdio.h>
2
3  // Dichiarazione di funzione
4  double max(double x, double y);
5
6  int main() {
7    double a, b, c;
8    printf("Inserire due interi separati da una virgola [a,b]: ");
9    scanf("%lf,%lf",&a,&b);
10
11   // Chiamata di funzione
12   c = max(a,b);
13
14   printf("Max(%lf,%lf) = %lf\n",c);
15   return 0;
16 }
17
18 // Definizione di funzione
19 double max(double x, double y) {
20   if(x>=y) return x;
21   else     return y;
22 }
```

- La funzione `max()`, dichiarata a riga 4, è visibile dalla funzione `main()`.
- Grazie alla dichiarazione a riga 4, la definizione di `max()` può essere posizionata dopo la definizione del `main()`.

Visibilità delle funzioni: esempio 3/5

```
1  #include<stdio.h>
2
3  // Dichiarazione e definizione di funzione
4  double max(double x, double y) {
5    if(x>=y) return x;
6    else     return y;
7  }
8
9  int main() {
10   double a, b, c;
11   printf("Inserire due interi separati da una virgola [a,b]: ");
12   scanf("%lf,%lf",&a,&b);
13
14   // Chiamata di funzione
15   c = max(a,b);
16
17   printf("Max(%lf,%lf) = %lf\n",c);
18   return 0;
19 }
```

- La funzione `max()`, definita a riga 4, è visibile dal `main()`.
- In questo caso, la definizione vale anche come dichiarazione.

Visibilità delle funzioni: esempio 4/5

```
1  #include<stdio.h>
2
3  // Funzione visibile da print_square() e main()
4  int square(int x) {
5    return x*x;
6  }
7
8  // Funzione visibile da main()
9  void print_square(int x) {
10   printf("%d^2 = %d\n",x,square(x));
11 }
12
13 int main() {
14   int x;
15   printf("Inserire un intero: ");
16   scanf("%d",&x);
17
18   print_square(x);
19   return 0;
20 }
```

- La funzione `square()` è visibilie sia dalla funzione `print_square()` che dal `main()`.
- La funzione `print_square()` è visibile unicamente dal `main()`.

Visibilità delle funzioni: esempio 5/5

```
1  #include<stdio.h>
2
3  int main() {
4    int x;
5
6    // Rende visibile print_square() alle istruzioni successive
7    void print_square(int);
8
9    printf("Inserire un intero: ");
10   scanf("%d",&x);
11
12   print_square(x);
13   return 0;
14 }
15
16 // Funzione visibile da print_square()
17 int square(int x) {
18   return x*x;
19 }
20
21 // Non visibile da square() e da main() prima di riga 7.
22 void print_square(int x) {
23   printf("%d^2 = %d\n",x,square(x));
24 }
```

- Nel blocco del `main()`, `print_square()` è visibile grazie alla dichiarazione a riga 7.
- Grammaticalmente corretto ma poco usato e poco elegante. Da evitare.

Passaggio dei parametri a funzione

- Una funzione può fare uso di parametri (argomenti) per svolgere il proprio compito.
  - **Parametri formali**: parametri dichiarati nella definizione/dichiarazione.
  - **Parametri attuali**: parametri passati al momento della chiamata.
- I parametri attuali possono essere il valore di un qualsiasi tipo di espressione, come:
  - valore costante,
  - valore di una variabile,
  - valore di ritorno di una funzione,
  - valore di una espressione complessa.
- In C il **passaggio dei parametri** avviene secondo la modalità del **passaggio per valore**.
- Dal punto di vista della funzione, gli *argomenti* (**parametri formali**) sono variabili locali i cui *valori iniziali* (**parametri attuali**) sono passati alla funzione al momento in cui questa è invocata.
- Il nome del parametro formale **maschera** le variabili con lo stesso nome esterne alla funzione.
- Il valore del parametro formale può essere modificato all'interno della funzione, senza provocare effetti all'esterno.
  - Attenzione: perchè con la `scanf()` riusciamo invece a modificare il valore di una variabile esterna alla funzione?
- Se il tipo del parametro passato differisce dal tipo dell'argomento della funzione, si applicano le stesse regole di conversione implicita di tipo viste per gli operatori.

Passaggio per valore: esempio

```
1  #include <stdio.h>
2
3  int c() {return -3;}
4
5  void print_incr(int x) {
6    x++;
7    printf("x=%d\n",x);
8  }
9
10 int main() {
11   int x=0;
12
13   print_incr(2);        // Stampa "x=3"
14   print_incr(x+1);      // Stampa "x=2"
15   print_incr(x);        // Stampa "x=1"
16   printf("x=%d\n",x);   // Stampa "x=0"
17   print_incr(-2.7);     // Stampa "x=-1"
18   print_incr(c());      // Stampa "x=-2"
19   return 0;
20 }
```

- In `print_incr()` ogni riferimento ad `x` è un riferimento al parametro formale `x`.
- Il valore dell'espressione passata come argomento a `print_incr()` viene utilizzato per inizializzare il parametro formale `x`.
- La costante `double -2.7` (riga 15) viene troncata in `-2` e convertita in `int` prima di essere assegnata al parametro formale `x`.

Passaggio di parametri: ordine di valutazione degli argomenti

- Che cosa stampa la seguente chiamata a `printf()`?

```
1 int i=0;
2 printf("i=%d i=%d\n",i,++i);
```

  Risposta: non definito!

- Lo standard ISO C89 non impone nessuna regola sull'ordine di valutazione dei parametri attuali di una funzione.
  - L'ordine di valutazione degli argomenti è **non specificato**.
- Nell'esempio sopra, la stampa dipende dal compilatore. Potrebbe essere:

```
i=0 i=1
```

  oppure

```
i=1 i=1
```

- Se i parametri attuali contengono side effects questi sono risolti prima che la chiamata a funzione sia attivata.

Nota storica: le funzioni pre-ANSI

- Abbiamo detto che le dichiarazioni di funzioni (prototipi) che ritornano `int` non sono strettamente necessarie in termini di validità sintattica del codice C.
- Abbiamo anche sottolineato che è sempre importante rendere visibile una funzione prima della sua invocazione.
- Il mancato uso di prototipi o un errato uso dei prototipi può causare situazioni di ambiguità semantica nel codice.
- Per poter comprendere a fondo l'utilità dei prototipi, è necessario vedere come è definita la vecchia sintassi K&R (pre-ANSI) per dichiarazioni e definizioni di funzioni.
- La definizione e dichiarazione di funzione sono i due aspetti del linguaggio C sui cui il comitato di standardizzazione ANSI ha apportato le maggiori modifiche rispetto alla sintassi K&R.
- La sintassi obsoleta K&R è comunque supportata dallo standard ISO C89.

K&R e ISO C89 (in breve)

1. La sintassi K&R supporta parzialmente i prototipi.
   - **Motivazione**: sollevare i compilatori dall'oneroso compito (per i calcolatori dell'epoca) di verificare la coerenza tra tipi di dato al momento della chiamata di funzione.
   - I controlli di consistenza sono relegati al programmatore.
2. La standard ISO C89 supporta la sintassi K&R e la estende nella forma che conosciamo e che dovremmo utilizzare.
   - **Motivazione**: supportare la compilazione di codice scritto con sintassi K&R e, allo stesso tempo, fornire nuovi strumenti di controllo al compilatore.
   - I controlli di consistenza sono relegati al compilatore, che è *costretto* a riconosce come sintatticamente valida anche la vecchia sintassi obsoleta.
3. Nelle revisioni successive dello standard alcuni costrutti sintattici alla K&R sono proibiti.

Dichiarazioni e definizioni nello stile K&R

- Le seguenti dichiarazioni e definizioni di funzioni nello stile K&R sono sintatticamente valide nello standard ISO C89.
- Le dichiarazioni K&R non includono il tipo e il numero di argomenti della funzione. E' richiesto che un prototipo specifichi unicamente il tipo di ritorno della funzione.

```
1  // Calcola il numero di secondi a partire da un orario
2  // composto da ora, minuti e secondi
3  int seconds();
4
5  // Restituisce il massimo tra due argomenti
6  double max();
```

- I tipi degli argomenti della funzione sono elencati subito prima del corpo della funzione.

```
1  int seconds(h,m,s)
2    int h; int m; int s;
3  {
4    return (3600*h+60*m+s);
5  }
6
7  double max(x,y)
8    double x; double y;
9  {
10   return x>=y ? x : y;
11 }
```

Sintassi K&R: il tipo `int` implicito

- Il tipo di dato `int` è considerato come **tipo di default**. Abbiamo già notato questa convenzione nelle dichiarazioni di tipo. Ed esempio, `short` equivale a dire `short int`, `long` equivale a dire `long int`, ecc.
- Se il tipo di dato restituito dalla funzione è `int`, allora può essere omesso, sia nei prototipi che nelle definizioni. Il prototipo può essere interamente omesso.

```
1 // Prototipo
2 seconds();    // Possiamo ometterlo
3
4 // Definizione
5 seconds(h,m,s)
6   int h; int m; int s;
7 {
8   return (3600*h+60*m+s);
9 }
```

- Il tipo di dato `int` può essere omesso anche quando riguarda gli argomenti della funzione.

```
1 // Prototipo
2 seconds();    // Possiamo ometterlo
3
4 // Definizione
5 seconds(h,m,s) {
6   return (3600*h+60*m+s);
7 }
```

Sintassi K&R: chiamata a funzione

- Vediamo alcune caratteristiche della chiamata a funzione utilizzando la sintassi K&R.
- **Tipo di ritorno della funzione**:
  - se la funzione non è visibile al momento della chiamata, il compilatore assume che il tipo di ritorno sia `int`;
  - se la funzione ha tipo diverso da `int`, si ha un errore di sintassi.
- **Passaggio dei parametri**:
  - i parametri della funzione vengono inferiti direttamente dalla chiamata;
  - il compilatore assume che la funzione abbia numero e tipo dei parametri formali corrispondente al numero e tipo dei parametri attuali che vede nella chiamata;
  - per il passaggio dei parametri sono applicate le regole di **argument promotion**:
    1. regole di *integral promotion* applicate agli argomenti di tipo intero,
    2. i tipo di dato `float` sono convertiti in `double`.
- Non essendoci nessun controllo sul tipo e numero di argomenti della funzione, potremmo avere i seguenti comportamenti non definiti:
  - se i tipi dei parametri attuali, dopo le conversioni, non sono compatibili con i tipi dei parametri formali, il comportamento è non definito.
  - se il numero dei parametri attuali non è compatibile col numero dei parametri formali, il comportamento è non definito.

Esempio: quando i prototipi fanno la differenza 1/4

- Esempio sintatticamente errato.

```
1 int sum(int x, int y) { return x+y; }
2
3 int main() {
4   int x = sum(-1); // Errore di sintassi
5   return 0;
6 }
```

La funzione `main()` vede il prototipo della funzione `sum()`. Sono richiesti due parametri: errore di sintassi.

- Esempio sintatticamente corretto ma semanticamente non definito.

```
1 int main() {
2   int x = sum(-1); // Sintassi OK: non vede il prototipo
3   return 0;
4 }
5
6 int sum(int x, int y) { return x+y; }
```

La funzione `main()` non vede il prototipo della funzione `sum()` e lo inferisce dalla chiamata: assume che `sum()` abbia un solo argomento di tipo `int`. Uno dei due parametri di `sum()` avrà un valore non definito al momento dell'esecuzione.

Esempio: quando i prototipi fanno la differenza 2/4

- Esempio sintatticamente errato.

```
1 void copyright(void);
2
3 int main() {
4   copyright(1); // Errore di sintassi
5   return 0;
6 }
7
8 void copyright() { printf("Copyright (c) 2015\n"); }
```

Il `main()` vede il prototipo della funzione `copyright()`: errore di sintassi.

- Esempio sintatticamente corretto ma semanticamente non definito.

```
1 void copyright(); // Prototipo K&R
2
3 int main() {
4   copyright(1); // Sintassi OK
5   return 0;
6 }
7
8 void copyright() { printf("Copyright (c) 2015\n"); }
```

Il `main()` interpreta il prototipo alla K&R. In questo esempio probabilmente non ci sono conseguenze, ma il generale il comportamento di codice simile è non definito.

Esempio: quando i prototipi fanno la differenza 3/4

- Esempio sintatticamente errato: anche in K&R è necessario specificare un prototipo se il tipo di ritorno della funzione è diverso da `int`.

```
1 int main() {
2   double x = sum(-1); // Errore: conflitto tipo di ritorno
3   return 0;
4 }
5
6 double sum(double x, double y) { return x+y; }
```

La chiamata a riga 2 viene interpretata come una *dichiarazione implicita* della funzione `int sum(int)`. Il compilatore individua un conflitto con il tipo di ritorno indicato nella definizione a riga 6.

- Esempio sintatticamente corretto ma semanticamente non definito.

```
1 double sum(); // Prototipo K&R
2
3 int main() {
4   double x = sum(-1); // Sintassi OK.
5   return 0;
6 }
7
8 double sum(double x, double y) { return x+y; }
```

Non ci sono conflitti sul tipo di ritorno, grazie al prototipo a riga 1. Il compilatore non individua il conflitto sul numero di argomenti. Comportamento non definito.

Esempio: quando i prototipi fanno la differenza 4/4

- Esempio sintatticamente corretto ma semanticamente non definito.

```c
#include <stdio.h>

void func(); // Prototipo K&R

int main() {
  unsigned short int x = 1;
  func(x); // Argument promotion tra tipi di dato intero
  return 0;
}
void func(double x) { printf("%f\n",x); }
```

Il valore di `x` viene convertito in `unsigned int` ma `func()` si aspetta un `double`.

- Esempio sintatticamente e semanticamente corretto.

```c
#include <stdio.h>

void func(double); // Prototipo C89

int main() {
  unsigned short int x = 1;
  func(x); // Convertito correttamente in double.
  return 0;
}
void func(double x) { printf("%f\n",x); }
```

Il valore di `x` viene viene correttamente convertito in `double`.