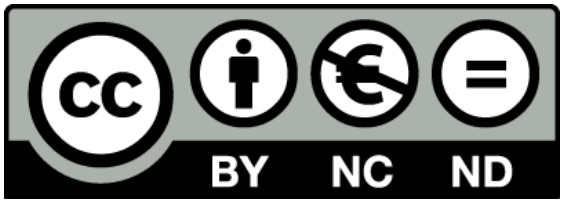
Laboratorio di Programmazione di Sistema

Programmazione Procedurale 2

Luca Forlizzi, Ph.D.

Versione 20.2



Comunicazione tra Procedure

- Nella Programmazione Procedurale, un programma è suddiviso in procedure
- Per poter realizzare programmi non banali, è necessario che le procedure che formano un programma comunichino tra loro, scambiandosi informazioni
- I meccanismi di comunicazione principali adottati dalla maggior parte degli *HLL*, sono il *passaggio di parametri* e la *restituzione del risultato*
- Questa presentazione descrive
  - le regole generali di C Standard in merito a tali meccanismi
  - come realizzare in *ASM* procedure foglia in cui parametri e risultato hanno tipi base

Definizione di Funzione con Parametri e Risultato

- La comunicazione tra funzioni in C, presenta molte analogie sintattiche e semantiche con la comunicazione tra metodi in Java, ma anche alcune significative differenze
- Una notevole differenza tra Java e C è il fatto che non è possibile effettuare l'*overloading* delle funzioni C, diversamente quanto accade con i metodi Java
- Ovvero in C non è possibile definire due funzioni diverse con lo stesso nome, anche se differiscono per il tipo del risultato e per quantità e tipi dei parametri

Definizione di Funzione con Parametri e Risultato

- La sintassi della definizione di una funzione con parametri e risultato è descritta in Schema1, in cui **RetType** è il *return-type*, **Name** è il nome della funzione, **Body** è un insieme di costrutti C disposti in sequenza, **ParList** è la *lista dei parametri*

Schema1

```
RetType Name ( ParList )
{
   Body
}
```

Definizione di Funzione con Parametri e Risultato

- Il *return type* della funzione è il tipo del risultato restituito dalla funzione
  - Nella definizione, il return type precede il nome della funzione
  - Il return type può essere un qualunque tipo di C Standard, ad esclusione dei tipi array
  - Una funzione che non restituisce risultato viene definita specificando `void` come return type

Definizione di Funzione con Parametri e Risultato

- La *lista dei parametri*, inserita tra una coppia di parentesi, è un elenco di *dichiarazioni di parametro*, separate da virgole
- Un parametro è una variabile a disposizione della funzione, con alcune particolarità
  - un parametro non è *visibile* dai costrutti che si trovano al di fuori della definizione di funzione; ovvero se usato in un costrutto non contenuto alla definizione della funzione, il nome del parametro non denota il parametro
  - ogni chiamata di funzione assegna a ciascun parametro un valore, detto *argomento*
- Una dichiarazione di parametro indica nome e tipo di uno dei parametri, mediante la stessa sintassi usata per le ordinarie dichiarazioni di variabile
- Se una funzione non ha parametri, allora lista dei parametri è costituita dalla keyword `void`

Corpo di una Funzione e Blocchi

- Il corpo di una funzione può contenere sia istruzioni che dichiarazioni di variabile
- Inoltre, all'interno del corpo di una funzione, è possibile aggregare istruzioni e dichiarazioni in *blocchi*
- Un blocco è un costrutto che ha la sintassi mostrata in Schema2, dove **Body**, detto *corpo* del blocco, è un insieme di costrutti C disposti in sequenza; i caratteri { e } vengono chiamati, rispettivamente, *delimitatore iniziale* e *delimitatore finale*

Schema2

```
{
   Body
}
```

Corpo di una Funzione e Blocchi

- Si osservi che il corpo di una funzione è un caso specifico di blocco, così come il corpo di un'istruzione di selezione o di iterazione non costituito da una singola istruzione
- Si dice che un blocco B_2 è *annidato* in un altro blocco B_1 se valgono entrambe le seguenti condizioni
  - il delimitatore iniziale di B_2 segue in ordine testuale il delimitatore iniziale di B_1
  - il delimitatore finale di B_2 precede in ordine testuale il delimitatore finale di B_1
- In base alla definizione, un blocco non è annidato in se stesso, mentre, dati tre blocchi B_1, B_2 e B_3, se B_3 è annidato in B_2 e B_2 è annidato in B_1, allora B_3 è annidato in B_1

Corpo di una Funzione e Blocchi

- Sia B un blocco e sia D una dichiarazione contenuta in B ma non in blocchi annidati in B
- Se D non contiene la keyword `extern` e dichiara una variabile di nome v, allora viene allocata una variabile di nome v, distinta da ogni altra variabile del programma; si dice che v è una *variabile locale* di B
- Se un blocco B è il corpo di una funzione, ciascuno dei parametri della funzione è una variabile locale di B

Corpo di una Funzione e Blocchi

- Una variabile v locale di un blocco B, non è *visibile* dai costrutti esterni a B: ovvero l'identificatore v, usato da costrutti esterni a B, non denota la variabile v locale di B
- Siano B_1 e B_2 due blocchi, con B_2 annidato in B_1; se sia B_1 che B_2 dichiarano una variabile locale di nome v, vengono allocate due distinte variabili con tale nome, una locale a B_1 e l'altra a B_2; la dichiarazione della variabile locale di B_2 *nasconde* l'altra dichiarazione per i costrutti contenuti nel corpo di B_2, ovvero nel corpo di B_2 il nome v denota la variabile locale di B_2 e non quella locale di B_1
- La visibilità delle variabili in C verrà trattata in modo più approfondito in una futura presentazione

Corpo di una Funzione e Blocchi

- C89 pone un vincolo sulla posizione delle dichiarazioni e istruzioni presenti in un blocco: le dichiarazioni di variabili locali di un blocco B devono precedere, in ordine testuale, tutte le istruzioni che si trovano in B, comprese le istruzioni che si trovano in blocchi annidati in B
- In C99 e versioni successive non c'è questa restrizione (analogamente a quanto avviene in Java e C++)

Corpo di una Funzione e Blocchi

- Se una funzione ha un risultato, il corpo della funzione deve contenere una o più istruzioni `return` seguite da un'espressione
  - per ciascuna espressione **E** che segue una istruzione `return`, il tipo di **E** deve essere uguale al return type, oppure deve essere possibile convertire un valore del tipo di **E** al return type
  - se viene eseguita una delle istruzioni `return`, l'esecuzione della funzione termina e il risultato dell'esecuzione è pari al valore assunto dall'espressione che segue l'istruzione `return` eseguita, eventualmente convertito al return type
  - se l'esecuzione delle istruzioni che formano il corpo della funzione raggiunge il termine del corpo stesso senza che sia stata eseguita una istruzione `return`, si ha un undefined behavior

Chiamata di Funzione con Parametri e Risultato

- La chiamata di una funzione ha la sintassi mostrata in Schema3, dove **Name** è il nome della funzione e **ArgList** è la *lista degli argomenti*

Schema3

Name (**ArgList**)

- La lista degli argomenti è una lista di espressioni, chiamate *argomenti*, separate da virgole

Chiamata di Funzione con Parametri e Risultato

- Una chiamata di una funzione che non restituisce risultato dovrebbe sempre essere usata per formare un'istruzione costituita dalla sola chiamata
- Una chiamata di funzione che ha un risultato
  - può invece essere parte di un'espressione più estesa
  - produce un risultato, con lo stesso tipo del return type, che viene usato per valutare l'espressione in cui la chiamata è inserita

Chiamata di Funzione con Parametri e Risultato

- Per essere corretta, una chiamata di funzione dovrebbe avere tanti argomenti quanti sono i parametri presenti nella lista dei parametri contenuta nella definizione della funzione
- Infatti, una chiamata corretta stabilisce una corrispondenza biunivoca tra argomenti e parametri, basata sulle posizioni che essi hanno nelle rispettive liste: ogni argomento corrisponde al parametro che occupa, nella lista dei parametri, la stessa posizione che l'argomento occupa nella lista degli argomenti

Chiamata di Funzione con Parametri e Risultato

- La abstract machine esegue una chiamata effettuando le seguenti operazioni
  - per ciascun argomento
    - calcola il valore dell'argomento
    - se necessario, converte il valore dell'argomento al tipo che il corrispondente parametro ha o, come spiegheremo meglio tra poco, *dovrebbe avere*
    - copia il valore ottenuto nel parametro
  - esegue la funzione
  - al termine dell'esecuzione della funzione, utilizza il risultato nella valutazione dell'espressione che contiene la chiamata, se necessario convertendone il tipo

Chiamata di Funzione con Parametri e Risultato

- Naturalmente, l'esecuzione della chiamata è corretta solo se
  - tutti gli argomenti hanno tipo uguale a quello dei parametri corrispondenti, o vengono correttamente convertiti a tali tipi
  - il risultato ha o può essere convertito ad un tipo valido nell'espressione che contiene la chiamata
- Uno degli obiettivi principali del type checking effettuato dal traduttore, è quello di individuare le differenze tra i tipi degli argomenti e quelli dei corrispondenti parametri, in modo da generare un codice eseguibile che effettui le corrette conversioni di tipo oppure da segnalare al programmatore la non correttezza di una chiamata
- Mostreremo tra breve che esiste una significativa differenza tra C e Java nel modo in cui il type checking opera in relazione alle chiamate

Modalità del Passaggio dei Parametri

- In C il passaggio di parametri avviene in generale *per valore*
- Ovvero, come abbiamo detto, quando viene effettuata una chiamata, si calcola il valore dell'argomento, si esegue l'eventuale conversione di tipo e il valore risultante viene copiato nel parametro corrispondente
- Poiché un parametro contiene una copia del valore dell'argomento, ogni eventuale modifica del parametro effettuata durante l'esecuzione della funzione non ha effetto sull'argomento

Modalità del Passaggio dei Parametri

- Nel corpo della funzione, un parametro è una variabile a tutti gli effetti e quindi può essere modificato, senza conseguenze per il contenuto di una variabile passata come argomento
- Ciò è utile perché consente di ridurre il numero di variabili richieste dal programma
- Code1a mostra una funzione per il calcolo del fattoriale di un numero intero non negativo

Code1a

```
unsigned fattoriale( unsigned n ) {
  unsigned i, p = 1;
  for ( i = 2 ; i <= n ; i++ ) p *= i;
  return p;
}
```

Modalità del Passaggio dei Parametri

- In Code1b la stessa funzione viene realizzata evitando di dichiarare la variabile `i`, grazie al fatto che si possono contare le iterazioni effettuate decrementando il parametro `n`

Code1b

```
unsigned fattoriale( unsigned n ) {
  unsigned p = 1;
  for ( ; 2 <= n ; n-- ) p *= n;
  return p;
 }
```

Modalità del Passaggio dei Parametri

- In certi casi potrebbe essere utile la possibilità di modificare gli argomenti
- Ad esempio si supponga di voler realizzare una funzione che data una quantità di tempo espressa in secondi, calcola a quante ore, minuti e secondi residui, tale quantità corrisponde
- Poiché una funzione può restituire un solo valore, si potrebbe pensare di usare tre argomenti per comunicare i valori calcolati al di fuori della funzione
- Ma le variabili gli argomenti non verrebbero modificati dagli assegnamenti fatti ai parametri

Modalità del Passaggio dei Parametri

- L'esempio Code2 mostra che la funzione non avrebbe il comportamento voluto

Code2

```
#include <stdio.h>
void conv_t( int tempo, int ore, int min, int sec ) {
  ore = tempo / 3600;
  min = ( tempo % 3600 ) / 60;
  sec = ( tempo % 3600 ) % 60;
}
int main(void) {
  int h = 0, m = 0, s = 0;
  conv_t( 1354, h, m, s );
  printf( "%d␣%d␣%d", h, m, s ); /* stampa 0 0 0 */
  return 0;
}
```

- In *Programmazione Procedurale 4* verrà mostrato come ottenere il comportamento voluto per mezzo di puntatori

Procedure e Type checking

- Per comprendere come avviene il type checking in C, è utile ricordare alcuni aspetti della semantica di Java
- In Java, come la maggior parte degli *HLL*, l'esecuzione di un metodo prevede il *type checking* ed eventualmente delle conversioni di tipo
- A tempo di traduzione, per ciascuna chiamata di metodo, si controlla che la quantità e i tipi degli argomenti siano corretti in relazione alla quantità e ai tipi dei parametri corrispondenti
  - se il numero degli argomenti è diverso dal numero dei parametri si ha un errore di traduzione
  - altrimenti se i tipi sono uguali, si prosegue con la traduzione
  - altrimenti se i tipi sono diversi ma *è sensato* convertire ciascun argomento al tipo del parametro corrispondente, viene generato il codice che esegue la conversione e si prosegue con la traduzione
  - altrimenti si ha un errore di traduzione

Procedure e Type checking

- Durante la traduzione, per ciascuna chiamata di metodo, si controlla che il tipo del risultato sia valido nell'espressione in cui si trova la chiamata
  - in caso positivo, si prosegue con la traduzione
  - altrimenti si ha un errore di traduzione
- Si osservi che i suddetti controlli
  - devono essere eseguiti nel momento in cui il traduttore esamina ciascuna chiamata di metodo
  - possono essere eseguiti a condizione che il traduttore conosca i tipi del risultato e dei parametri del metodo chiamato

Procedure e Type checking

- I traduttori degli *HLL* operano leggendo il codice sorgente a partire dalla prima riga e procedendo verso l'ultima
- Quindi, in un programma Java, se una chiamata di metodo si trova in una riga precedente la definizione del metodo, il traduttore non ha le informazioni necessarie per effettuare il type checking, in quanto non conosce ancora i tipi del risultato e dei parametri del metodo chiamato

Procedure e Type checking

- La soluzione che i traduttori Java tipicamente adottano, è effettuare la traduzione mediante un processo composto da 2 fasi, chiamate *pass*, ciascuna delle quali corrisponde ad una lettura del codice sorgente
  - In pass 1, si costruisce un *dizionario* in cui, per ogni definizione di metodo, si memorizzano i tipi del risultato e dei parametri
  - In pass 2, avendo a disposizione tutte le informazioni raccolte durante pass 1, si genera il codice che effettua le chiamate dei metodi

Procedure e Type checking

- Come sappiamo, il C è stato progettato per permettere la generazione di codice efficiente, ma anche per facilitare la realizzazione di traduttori semplici ed efficienti (negli anni 70 e 80 l'efficienza del traduttore era molto importante)
- Per questo motivo, le regole del C sono fatte in modo da consentire la traduzione dei programmi C effettuando una sola fase di lettura del codice sorgente
- In particolare, le regole relative alle conversioni di tipo e al type checking nelle chiamate di funzioni C, sono pensate per traduttori che effettuano una sola fase di lettura del codice sorgente

Procedure e Type checking

- La scelta di consentire la traduzione in una sola fase, comporta un problema di fondo: generare, in una sola fase, il codice per le chiamate di funzione che precedono le definizioni delle funzioni chiamate
- Una soluzione estremamente semplice, sarebbe quella di evitare tali situazioni, tramite una regola che proibisca di effettuare chiamate di funzioni le cui definizioni si trovano, nel codice sorgente, dopo la chiamata

Procedure e Type checking

- Questa soluzione *non* venne adottata dal C, per i seguenti motivi:
  - avrebbe costretto i programmatori scegliere con attenzione in che punto del sorgente inserire le definizioni di funzione
  - avrebbe reso impossibile la realizzazione di situazioni di *mutua ricorsione*, in cui una funzione A chiama una funzione B e la funzione B chiama a sua volta A
- Il C risolve il problema attraverso due meccanismi
  - la *dichiarazione pura* di funzione
  - la *dichiarazione implicita* di funzione (solo in C Tradizionale e C89)

Dichiarazioni di Funzione

- In C, una *dichiarazione di funzione* è un costrutto che indica il nome di una funzione, il tipo del suo risultato
- In C Standard esistono 2 grandi categorie di dichiarazioni
  - le *dichiarazioni in forma di prototipo*, o più semplicemente *prototipi*, sono dichiarazioni che indicano, oltre al nome della funzione e al tipo del suo risultato, anche quanti sono e quali tipi hanno i parametri della funzione
  - le *dichiarazioni tradizionali*, invece, indicano solo nome e tipo del risultato di una funzione
- I prototipi sono da preferire, in ogni circostanza, e infatti quasi tutte le dichiarazioni usate in LPS sono prototipi
- Tuttavia, è opportuno che un programmatore conosca, almeno in parte, le dichiarazioni tradizionali anche se non le utilizza di proposito, in quanto la loro presenza in C Standard comporta delle conseguenze che si riflettono anche sull'uso dei prototipi

Dichiarazioni di Funzione

- Naturalmente conosciamo da tempo un costrutto che indica il nome di una funzione e il tipo del risultato: la definizione della funzione
- Una definizione fornisce anche altre informazioni, ovvero la descrizione del corpo della funzione
- Infatti, una definizione di funzione è un caso particolare di dichiarazione di funzione
- Le definizioni di funzione mostrate in precedenti presentazioni, indicano anche quanti sono e quali tipi hanno i parametri della funzione: sono dunque definizioni in forma di prototipo

Dichiarazioni Pure di Funzione

- Esistono però delle dichiarazioni che non sono definizioni di funzione: le chiamiamo *dichiarazioni pure*
- La sintassi di una dichiarazione pura in forma di prototipo è mostrata in Schema3, in cui **RetType** è il *return-type*, **Name** è il nome, **ParList** è la *lista dei parametri*

Schema3

RetType Name (ParList) ;

- Poiché lo scopo di una dichiarazione pura è solo indicare quanti sono e quali tipi hanno i parametri di una funzione, in una dichiarazione pura i nomi dei parametri sono opzionali e vengono usati al solo scopo di documentare meglio il programma

Dichiarazioni Pure di Funzione

- Le dichiarazioni pure (meglio se in forma di prototipo) possono essere usate per risolvere il problema di generare il codice per le chiamate di funzione che precedono le definizioni delle funzioni chiamate
- È sufficiente inserire una dichiarazione pura della funzione prima della chiamata di funzione, per fornire al compilatore tutte le informazioni necessarie ad effettuare il type checking e generare il codice della chiamata

Dichiarazioni Pure di Funzione

- Dunque un programmatore può inserire all'inizio del codice sorgente, prima di qualunque istruzione, le dichiarazioni pure di tutte le funzioni presenti nel programma
- In tal modo
  - conserva la libertà di inserire chiamate e definizioni di funzioni dove preferisce, senza preoccuparsi che le definizioni precedano le chiamate, in quanto ogni chiamata è comunque preceduta da una delle dichiarazioni pure poste all'inizio del codice sorgente
  - ottiene il beneficio del type checking

Dichiarazioni Pure di Funzione

- Quando una chiamata di funzione è preceduta da una dichiarazione in forma di prototipo della funzione, viene effettuato il type checking in modo molto simile a quanto accade in Java
- A tempo di traduzione, si controlla che il return type di ciascuna funzione chiamata sia valido nell'espressione in cui si trova la chiamata
  - in caso positivo, si prosegue con la traduzione
  - altrimenti si ha un errore di traduzione

Dichiarazioni Pure di Funzione

- A tempo di traduzione, per ciascuna chiamata di funzione, si controlla che la quantità e i tipi degli argomenti siano corretti in relazione alla quantità e ai tipi dei parametri corrispondenti
  - se il numero degli argomenti è diverso dal numero dei parametri si ha un errore di traduzione
  - altrimenti se i tipi sono uguali, si prosegue con la traduzione
  - altrimenti se i tipi sono diversi ma *è sensato* convertire ciascun argomento al tipo del parametro corrispondente, viene generato il codice che effettua la conversione e si prosegue con la traduzione
  - altrimenti si ha un errore di traduzione

Dichiarazioni Implicite di Funzione

- La seconda soluzione al problema di generare il codice per le chiamate di funzione che precedono le definizioni delle funzioni chiamate, consiste nell'evitare il type checking
- In C tradizionale e in C89, se una chiamata di funzione non è preceduta da alcuna dichiarazione della funzione, il compilatore genera ugualmente il codice per la chiamata, come se ci fosse una *dichiarazione implicita*, ovvero facendo alcune ipotesi sui tipi del risultato e dei parametri
  - Assume che il tipo del risultato sia `int`
  - Per ciascun argomento che ha un tipo intero standard con rank minore del rank di `int`, assume che il tipo del parametro corrispondente sia il tipo ottenuto applicando all'argomento la conversione di tipo IP (si veda *Conversione dei Dati*)
  - Per ciascun argomento di tipo `float`, assume che il parametro corrispondente abbia tipo `double`
  - Per ciascuno degli altri argomenti, assume che il parametro corrispondente abbia lo stesso tipo dell'argomento

Dichiarazioni Implicite di Funzione

- Il traduttore genera codice che effettua delle conversioni di tipo, dette *default argument promotions*, in accordo con le assunzioni precedenti, ovvero
  - Converte gli argomenti di tipo intero standard con rank minore del rank di `int` in base alla regola IP
  - Converte gli argomenti di tipo `float` a `double`
  - Lascia inalterati il risultato e i restanti argomenti
- Se le assunzioni sono corrette, la chiamata va a buon fine, altrimenti si ha un undefined behavior

Dichiarazioni Implicite di Funzione

- Il meccanismo della dichiarazione implicita offre il vantaggio di non dover aggiungere dichiarazioni pure ad un programma, al prezzo però di rinunciare al type checking e quindi di essere esposti ad errori non controllati dal compilatore
- L'esperienza accumulata in molti anni di uso del C, ha portato a concludere che gli svantaggi superano di gran lunga i vantaggi, e quindi il meccanismo della dichiarazione implicita è stato rimosso da C Standard
- In C99 e nelle versioni successive, una chiamata di funzione che non è preceduta da almeno una dichiarazione della funzione, è una constraint violation

Esercizio di Analisi di Codice

- Si descriva cosa accade traducendo ed eseguendo il programma seguente con una abstract machine di C Standard (segnalando eventuali differenze tra le versioni dello Standard)
- Per ogni violazione di regola sintattica, constraint o regola semantica, si scriva qual'è la regola violata e poi si rimuova l'istruzione o la dichiarazione che la contiene e si continui a tradurre ed eseguire il programma

Exercise1

```
int main (void) {
  short k = 4; int w, y = -3;
  float d = -9.2, e; long double h = .25633e8L
  e = funz4( k, d ); y = funz4( y, h ); w = funz5( 10 );
  return 0; }
int funz4( int x, double y ) {  return 6 * y + ( y - x ); }
float funz5( int x ) { return 2.5 * x * x; }
```

Esercizio di Analisi di Codice

- In C99 e versioni successive, tutte e 3 le chiamate sono constraint violation in quanto non sono precedute da dichiarazioni delle funzioni
- In C89, per tutte e 3 le chiamate si applica il meccanismo della funzione implicita
  - Nella prima chiamata, le default argument promotions convertono gli argomenti ai tipi dei corrispondenti parametri, quindi la chiamata viene effettuata in modo corretto
  - Nella seconda chiamata si ha un undefined behavior in quanto l'argomento di tipo `long double` non corrisponde al tipo `double` del parametro corrispondente
  - Nella terza chiamata si ha un undefined behavior in quanto il traduttore, per via della dichiarazione implicita, genera codice per un risultato di tipo `int`, ma la funzione restituisce un valore di tipo `float`

Dichiarazioni e Definizioni Tradizionali

- Nella programmazione C moderna (e in LPS), si usano quasi esclusivamente dichiarazioni e definizioni in forma di prototipo
- Tuttavia esistono in C anche altre dichiarazioni e definizioni di funzione: le dichiarazioni e definizioni *tradizionali*
- Il loro nome deriva dal fatto che sono le uniche tipologie di dichiarazioni e definizioni di funzione presenti nel C tradizionale
- Le dichiarazioni e le definizioni in forma di prototipo, infatti, sono state introdotte nel C89, la prima versione di C Standard

Dichiarazioni e Definizioni Tradizionali

- Dichiarazioni e definizioni tradizionali indicano al traduttore il tipo del risultato della funzione, ma non forniscono informazioni sui parametri della funzione, rendendo possibile un type checking solo parziale
- Per superare questa limitazione che il C tradizionale aveva (rispetto ad altri *HLL*), in C89 furono introdotte le dichiarazioni e definizioni in forma di prototipo
- Tuttavia, per motivi di backward compatibility, le dichiarazioni e definizioni tradizionali, pur essendo ufficialmente dichiarate come "obsolescent feature", non furono eliminate dal C89, con conseguenze poco desiderabili sul linguaggio, tra cui un aumento della complessità delle regole

Dichiarazioni e Definizioni Tradizionali

- Le successive versioni di C Standard, fino a C18 compresa, non hanno modificato questa situazione
- I *rumors* provenienti da ISO/IEC dicono che nella prossima versione di C Standard (nome provvisorio C2x) le definizioni tradizionali saranno finalmente rimosse dal linguaggio; ma le dichiarazioni pure tradizionali continueranno a far parte di C Standard, almeno per qualche altro anno
- In LPS non trattiamo le definizioni tradizionali, utili solo per mantenere software molto vecchio
- È opportuno, invece fare un accenno alle dichiarazioni pure tradizionali, per mostrare come la loro esistenza sia il motivo per cui, nelle dichiarazioni in forma di prototipo di funzioni prive di parametri, è necessario scrivere `void` tra le parentesi

Dichiarazioni e Definizioni Tradizionali

- La sintassi di una dichiarazione pura tradizionale è mostrata in Schema4, in cui **RetType** è il *return-type* e **Name** è il nome

Schema4

```
RetType Name ( ) ;
```

- Questa sintassi può facilmente indurre in errore un lettore poco esperto di C, suggerendo che la funzione non abbia parametri
- Invece, come detto, una dichiarazione pura tradizionale indica solo il tipo del risultato di una funzione, ma non fornisce alcuna informazione sui parametri

Dichiarazioni e Definizioni Tradizionali

- Se una chiamata di funzione è preceduta da una dichiarazione pura tradizionale e non da dichiarazioni in forma di prototipo, il traduttore genera codice che effettua il type checking solo sul tipo del risultato della funzione
- Per quanto riguarda gli argomenti, il traduttore si comporta in modo simile a quanto accade in C89 nel caso di dichiarazione implicita, ovvero effettua le *default argument promotions*
- In caso di mancata corrispondenza tra quantità e tipi di argomenti e parametri si ha un undefined behavior
- Il motivo per cui si raccomanda di dichiarare funzioni prive di parametri scrivendo `void` tra le parentesi che seguono il nome della funzione, è che non facendolo si scrive una dichiarazione tradizionale, perdendo i vantaggi del type checking

Dichiarazioni e Definizioni Tradizionali

- Code3a dichiara in forma di prototipo una funzione priva di parametri, e, nella chiamata della funzione, contiene un errore che il traduttore scopre immediatamente

Code3a

```
double no_par( void );

int main( void ) {
  double x = no_par( 10 );
  return 0;
}

double no_par( void ) { return 42.0; }
```

Dichiarazioni e Definizioni Tradizionali

- Ma se la funzione priva di parametri fosse dichiarata da una dichiarazione tradizionale, come in Code3b, il type checking non verrebbe effettuato
- Code3b contiene un errore che non viene rilevato dal traduttore; la traduzione del programma produce un codice eseguibile che, se eseguito, ha un undefined behavior

Code3b

```
double no_par( );

int main( void ) {
  double x = no_par( 10 );
  return 0;
}

double no_par( void ) { return 42.0; }
```

La Funzione `main` in Hosted C Standard

- Le versioni *hosted* di C Standard stabiliscono alcune regole relative alla funzione principale di un programma
- Come sappiamo, il nome della funzione principale è `main`
- Vi sono due possibili modi per definire `main` in modo che sia portabile; ogni implementazione può, in aggiunta, permettere definizioni implementation-defined di `main`
- Le due definizioni portabili di `main` sono
  - `int main( void ) { /* corpo */ }`
  - `int main( int argc, char *argv[] ) { /* corpo */ }`
- La seconda di tali forme consente, a differenza della prima, all'ambiente operativo che avvia un'esecuzione di un programma di inviare un'elenco di argomenti al programma

La Funzione `main` in Hosted C Standard

- Nella seconda forma di `main`, i nomi dei parametri potrebbero naturalmente essere diversi, `argc` e `argv` sono solo i nomi usati convenzionalmente nella maggior parte dei programmi
- Il parametro `argv` è un array di puntatori a carattere di lunghezza pari al valore `argc+1`
- L'elemento `argv[ argc ]` è un null pointer che segnala la fine dell'elenco degli argomenti
- Ciascuno degli altri elementi, è il puntatore al primo carattere di un array di caratteri
- Ciascun array di caratteri contiene una stringa terminata da un carattere di valore `0` detto *terminatore di stringa* ed ha lunghezza pari al numero di caratteri della stringa (terminatore di stringa compreso)

La Funzione `main` in Hosted C Standard

- Negli ambienti operativi a riga di comando, di solito vengono inviati al programma i seguenti argomenti
  - la stringa contenente il *nome del programma*, nell'array puntato da `argv[ 0 ]` (nella maggior parte degli ambienti, il nome del programma è il nome del file eseguibile che contiene il programma)
  - le parole, formate dai caratteri diversi da spazio, che sono state digitate sulla riga di comando successivamente al nome del programma (e prima di premere il tasto *invio*), negli array puntati dagli elementi di `argv` di indice maggiore i `0` e minore di `argc`, seguendo l'ordine in cui tali parole sono state digitate
- Altri ambienti possono inviare al programma argomenti diversi
- In ogni caso, il modo in cui un ambiente operativo invia argomenti al programma, non fa parte del programma stesso, e quindi non viene definito da C Standard

La Funzione `main` in Hosted C Standard

- Al termine dell'esecuzione, un programma restituisce all'ambiente operativo che ne ha avviato l'esecuzione, un valore intero detto *codice di stato*
- Tale valore viene di solito usato per indicare all'ambiente operativo se il programma ha funzionato in modo "normale" oppure se è accaduto qualcosa di anomalo, ad esempio un errore dovuto a mancanza di memoria o conseguente a operazioni su file

La Funzione `main` in Hosted C Standard

- Un programma termina in modo *normale* quando si verifica uno dei seguenti casi
  - l'attivazione principale di `main` esegue `return`; il codice di stato è il valore dell'espressione che segue `return`
  - l'esecuzione dell'attivazione principale di `main` raggiunge la fine del corpo di `main`; il codice di stato è `0`
  - viene eseguita la funzione `exit` definita dall'header `<stdlib.h>`; il codice di stato è l'argomento di `exit`
- Il valore del codice di stato viene usato dall'ambiente operativo secondo modalità da esso definite
- C Standard considera un valore del codice di stato pari a `0` oppure al valore della macro `EXIT_SUCCESS` definita in `<stdlib.h>`, come indicazione che il programma ha svolto correttamente il suo compito, mentre un valore pari a quello della macro `EXIT_FAILURE` definita in `<stdlib.h>` come indicazione del fatto che si è verificato un problema

Routine *ASM* con Parametri e Risultati

- Il modo più semplice ed efficiente per comunicare dati (parametri e risultati) tra routine in *ASM*, è quello di memorizzarli in registri, in quanto essi sono dispositivi di memorizzazione condivisi tra tutte le routine che formano un programma
- Quando scrive una routine, il programmatore può scegliere di dedicare alcuni registri al passaggio di parametri in ingresso alla routine e altri alla restituzione del risultato in uscita
- Questo metodo è particolarmente adatto per tradurre funzioni foglia in cui i tipi di parametri e risultato sono tipi base, in quanto di solito i valori di tali tipi possono essere interamente contenuti in singoli registri

Routine *ASM* con Parametri e Risultati

- La comunicazione tra routine effettuata mediante registri è semplice ed efficiente, ma ha severi limiti che emergono al crescere del numero di routine che formano il programma
- Il primo limite è il fatto che nello scrivere una routine A che chiama una seconda routine B, si deve pianificare l'uso dei registri tenendo conto di quali di essi vengono usati da B per ricevere i parametri e restituire i risultati
- Il secondo è che la quantità di dati scambiati tra due routine può essere tale da occupare molti registri o addirittura eccedere la capacità dell'insieme dei registri
- In future presentazioni discuteremo modi di tradurre in *ASM* funzioni non-foglia, con parametri e risultati di qualunque tipo

Materiale Didattico

- Per assimilare e approfondire i contenuti di questa presentazione, si consiglia di studiare anche
  - Esempio *Routine 2* disponibile su *Edu99*
  - Capitolo 9 di **[Ki]**