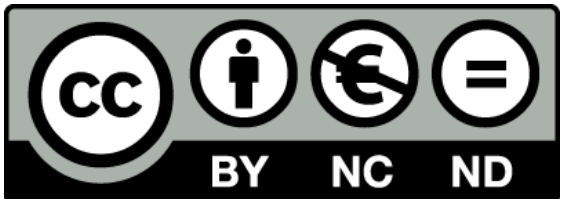
Laboratorio di Programmazione di Sistema

Programmazione Procedurale 2

Luca Forlizzi, Ph.D.

Versione 23.2



Procedure e Struttura del Programma

- Adottando lo stile della Programmazione Procedurale, si suddivide un programma in procedure
- Diviene quindi naturale sfruttare questa suddivisione per dare una struttura logica ai programmi
- La strutturazione di un programma ne rende più semplice la realizzazione, a condizione che le parti abbiano un certo grado di indipendenza
- Infatti, se le parti sono indipendenti
  - Possono essere costruite e modificate senza che sia necessario tenere conto delle altre parti del programma
  - Possono essere realizzate da programmatori diversi che non conoscono i dettagli del lavoro degli altri

Procedure e Struttura del Programma

- Per poter avere procedure indipendenti le une dalle altre, è necessario che ci siano meccanismi per rendere i dati di una procedura *locali*, ovvero inaccessibili dalle altre procedure
- Tuttavia le procedure di un programma non possono essere del tutto indipendenti le une dalle altre, altrimenti sarebbero programmi separati: alcuni dati devono necessariamente essere scambiati tra di loro
- Dunque un linguaggio che supporti appieno la Programmazione Procedurale, deve
  - Permettere di definire e chiamare procedure
  - Rendere possibile la *località dei dati* nelle procedure
  - Fornire meccanismi per scambiare dati tra procedure, in modo controllato

Procedure e Struttura del Programma

- La località dei dati viene ottenuta, nella maggior parte degli *HLL*, mediante la possibilità di definire *variabili locali* ad una procedura e presenta due aspetti, diversi ma tra loro correlati
  - *Località spaziale*: l'accesso ad una variabile locale ad una certa procedura, è consentito solo alle istruzioni di tale procedura
  - *Località temporale*: in certe condizioni, l'esistenza di una variabile locale può essere limitata al tempo di una esecuzione di una procedura
    - se una variabile v è locale ad una procedura P_1, essa è inaccessibile quando P_1 non è in esecuzione
    - se inoltre P_1 non utilizza il contenuto che v ha assunto al termine della precedente esecuzione di P_1 stessa, allora è possibile che v sia creata quando inizia un'esecuzione di P_1 e distrutta al termine di tale esecuzione
- Lo scambio di dati tra procedure avviene, nella maggior parte degli *HLL*, con il *passaggio di parametri* e la *restituzione del risultato*

Località Spaziale delle Variabili in C

- C Standard offre una nozione di località spaziale più sofisticata
- Le variabili possono essere locali a parti del programma più ridotte rispetto ad una procedura, chiamate *blocchi*
- Il corpo di una procedura è un blocco e ogni blocco può contenere altri blocchi, dando così vita ad una gerarchia di località spaziale
- Inoltre è possibile definire variabili locali ad un insieme di procedure, utilizzando la possibilità di dividere il codice sorgente di un programma in file separati

Località Spaziale delle Variabili in C

- Purtroppo, C Standard non consente di specificare la località di una variabile in modo semplice, ad esempio mediante *keyword*
- La località spaziale delle variabili dipende da due proprietà delle dichiarazioni, che nel seguito descriviamo con riferimento alle dichiarazioni di variabile
  - *Scope*: determina la parte del file sorgente nella quale una variabile è accessibile attraverso il nome definito nella dichiarazione
  - *Linkage*: stabilisce le circostanze in cui due o più distinte dichiarazioni di una variabile con lo stesso nome, presenti nello stesso file sorgente o in distinti file sorgente del programma, vengono considerate relative alla stessa variabile
- Il linkage, di solito, entra in gioco solo per programmi costituiti da più di un file sorgente, per questo non lo discutiamo ulteriormente

Località Spaziale delle Variabili in C

- Un blocco è un costrutto che aggrega un insieme di dichiarazioni di variabile e di istruzioni
- Un blocco 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
}
```

Località Spaziale delle Variabili in C

- Il corpo di una funzione e le istruzioni composte che abbiamo usato nelle istruzioni di selezione o di iterazione sono dei blocchi
- 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

Località Spaziale delle Variabili in C

- Come detto, un blocco contiene un insieme di dichiarazioni e di istruzioni (alcune delle quali possono essere contenute in blocchi annidati)
- C89 pone un vincolo sulla posizione delle dichiarazioni e istruzioni presenti in un blocco: le dichiarazioni contenute in un blocco B ma non in blocchi annidati in 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++)

Località Spaziale delle Variabili in C

- Sia B un blocco e sia D una dichiarazione che
  - è contenuta in B ma non in blocchi annidati in B
  - non contiene la keyword `extern`, che serve per controllare il linkage e che quindi non consideriamo
- Se D dichiara una variabile di nome n_v, allora tale variabile è distinta da ogni altra variabile del programma, anche se ha lo stesso nome; si dice che n_v è una *variabile locale* di B
- Lo *scope* della variabile locale di nome n_v inizia dal punto del codice sorgente in cui si trova D e termina nel punto in cui c'è il delimitatore finale di B
- Le variabili dichiarate da una dichiarazione contenuta in un blocco sono dette variabili di tipologia *block-scope*

Località Spaziale delle Variabili in C

- Sia D una dichiarazione che
  - non è contenuta in alcun blocco, ovvero è *esterna*
  - non contiene la keyword `extern`, che serve per controllare il linkage e che quindi non consideriamo
  - non contiene la keyword `static`, che per le dichiarazioni esterne serve per controllare il linkage e che quindi non consideriamo
- Se D dichiara una variabile di nome n_v, allora tale variabile è distinta da variabili locali (dichiarate senza `extern`) che hanno lo stesso nome; si dice che n_v è una *variabile esterna*
- Lo *scope* di una variabile esterna di nome n_v inizia dal punto del codice sorgente in cui si trova D e termina alla fine del file sorgente
- Le variabili dichiarate da una dichiarazione esterna sono dette variabili di tipologia *file-scope*

Località Spaziale delle Variabili in C

- Se un file sorgente contiene due dichiarazioni di variabile con lo stesso nome, gli scope delle due dichiarazioni potrebbero sovrapporsi
- Per come sono definiti gli scope, se due scope si sovrappongono, uno dei due è interamente contenuto nell'altro
- Quando gli scope di due dichiarazioni si sovrappongono, la variabile dichiarata nello scope più esterno è *nascosta* nello scope più interno: ovvero nello scope più interno, il nome della variabile si riferisce alla variabile che ha scope più interno
- Dunque una variabile è *visibile* nello scope della propria dichiarazione, esclusi eventuali scope più interni in cui essa è nascosta

Località Temporale delle Variabili in C

- Ogni oggetto, comprese quindi le variabili, ha un *lifetime*, ovvero una parte del tempo di esecuzione del programma durante la quale l'oggetto esiste nella memoria
- Durante il *lifetime* di un oggetto *o*
  - è possibile accedere ad *o*, tramite il suo nome (se *o* è una variabile) e tramite puntatori ad *o*
  - i puntatori ad *o* sono valori validi e costanti
  - *o* conserva il valore che vi viene memorizzato dagli accessi che effettuano operazioni di scrittura
- È possibile accedere ad un oggetto solo durante il suo *lifetime*
  - ogni tentativo di accesso ad un oggetto al di fuori del *lifetime* è un undefined behavior
  - quando termina il *lifetime* di un oggetto, i puntatori all'oggetto diventano valori indeterminati

Località Temporale delle Variabili in C

- L'inizializzazione di una variabile è l'operazione che memorizza nella variabile il valore che essa contiene quando inizia il *lifetime*
- Il *lifetime* di un oggetto è determinato da una proprietà chiamata *storage duration*
- Per le variabili, sono possibili due diverse *storage duration*
  - *Statica*: il *lifetime* è l'intero tempo di esecuzione del programma
  - *Automatica*: il *lifetime* inizia nel momento in cui la abstract machine esegue la dichiarazione della variabile, e termina quando la abstract machine termina l'esecuzione di tutto il codice contenuto nello scope della variabile

Località Temporale delle Variabili in C

- Una variabile con *storage duration* statica
  - esiste nella memoria sin da quando inizia l'esecuzione del programma, quindi viene allocata e deallocata staticamente
  - viene inizializzata una sola volta, prima che inizi l'esecuzione del programma, ad un valore che è sempre determinato
- Se la dichiarazione di una variabile con *storage duration* statica contiene un inizializzatore, esso deve essere un'espressione formata esclusivamente con valori costanti che viene usata per determinare il valore iniziale della variabile
- Altrimenti, se non è presente un inizializzatore
  - le variabili di tipo aritmetico vengono inizializzate a 0
  - le variabili di tipo puntatore vengono inizializzate al *null pointer*
  - l'inizializzazione delle variabili che hanno un tipo aggregato viene descritta in una futura presentazione

Località Temporale delle Variabili in C

- Una variabile con *storage duration* automatica
  - esiste nella memoria per un tempo che può essere un sotto-intervallo del tempo di esecuzione del programma, quindi viene allocata e deallocata dinamicamente
  - viene inizializzata durante l'esecuzione del programma, ovvero al momento dell'esecuzione della dichiarazione della variabile
- La dichiarazione di una variabile con *storage duration* automatica potrebbe essere eseguita più volte durante l'esecuzione del programma: ogni volta viene creata una nuova variabile, distinta dalle altre, che viene inizializzata
- Se la dichiarazione di una variabile con *storage duration* automatica contiene un inizializzatore, ogni volta che viene creata la variabile, l'inizializzatore viene valutato e il suo valore attuale usato per inizializzare la variabile appena creata
- Altrimenti la variabile ha un valore non determinato

Località Temporale delle Variabili in C

- La *storage duration* di una variabile dipende, in base a regole non immediate, dal tipo di *scope* e dall'eventuale *specificatore di classe di memorizzazione* presente nella dichiarazione della variabile
- Lo *specificatore di classe di memorizzazione* è una delle seguenti keyword che può essere inserita in una dichiarazione
  - `static`
  - `extern`
  - `register`
  - `auto`
- Una dichiarazione può avere al più uno specificatore di classe di memorizzazione
- Gli specificatori di classe di memorizzazione `register` e `auto` sono validi solo per variabili con *block-scope*

Località Temporale delle Variabili in C

- Una variabile dichiarata con *file-scope* ha *storage duration* statica, indipendentemente dalla presenza di uno specificatore di classe di memorizzazione
- Una variabile dichiarata con *block-scope*
  - ha *storage duration* statica se dichiarata con `static` o `extern`
  - ha *storage duration* automatica se dichiarata con `register` o `auto` o senza specificatore di classe di memorizzazione
- Nel seguito useremo il termine *variabile statica* per indicare una variabile con storage duration statica e il termine *variabile automatica* per indicare una variabile con storage duration automatica

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

- Se una funzione restituisce 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

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`

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 imponendo, mediante una regola, che la definizione di una funzione deve essere scritta nel codice sorgente in una posizione precedente a quella di tutte le chiamate a tale funzione

Procedure e Type checking

- Questa soluzione *non* venne adottata dal C, per i seguenti motivi:
  - sarebbe stata scomoda per i programmatori, costringendoli a scegliere con attenzione in che punto del sorgente inserire ciascuna definizione 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

Procedure e Type checking

- Il C risolve il problema attraverso due meccanismi
  - la *dichiarazione pura* di funzione
  - la *dichiarazione implicita* di funzione (solo in C Tradizionale e C89)
- Le dichiarazioni implicite di funzione, non essendo consentite in C99 e nelle successive versioni di C Standard, non vengono studiate in LPS
- Se si vuole approfondire la conoscenza del linguaggio C, è opportuno studiare anche le dichiarazioni implicite di funzione: il capitolo 9 di **[Ki]** è un buon punto di partenza

Dichiarazioni di Funzione

- La soluzione al problema di permettere la scrittura di chiamate di funzione in un punto del codice sorgente che precede la definizione della funzione, trae spunto dall'osservazione che, per poter fare il type checking delle chiamate di funzione
  - non serve conoscere l'intera definizione della funzione
  - è sufficiente sapere quali sono i tipi del risultato e dei parametri
- Da qui l'idea di mettere a disposizione un costrutto linguistico per le funzioni, diverso dalla definizione di funzione, che fornisca esattamente le informazioni necessarie al traduttore per fare il type checking
- In C, una *dichiarazione di funzione* è un costrutto che indica il nome di una funzione e il tipo del suo risultato

Dichiarazioni di Funzione

- In C Standard esistono 2 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 di Funzione

- Esistono delle dichiarazioni di funzione che non sono definizioni di funzione
- In LPS le chiamiamo *dichiarazioni pure* di funzione
- L'utilità delle dichiarazioni pure deriva dal fatto che per ogni funzione `f`, un programma C può contenere
  - un'unica definizione di `f`
  - molteplici dichiarazioni di `f`
- Ovviamente la presenza di due dichiarazioni di una funzione che non sono consistenti (ovvero non indicano le stesse informazioni su parametri e tipo del risultato) è una constraint violation

Dichiarazioni Pure di Funzione

- Le dichiarazioni pure 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

- In C Standard esistono 2 categorie di dichiarazioni pure
  - le *dichiarazioni pure prototipi* indicano, oltre al nome della funzione e al tipo del suo risultato, anche quanti sono e quali tipi hanno i parametri della funzione, e quindi consentono al traduttore di eseguire un type checking sia sul tipo del risultato che sui tipi dei parametri
  - le *dichiarazioni pure tradizionali* indicano solo nome e tipo del risultato di una funzione, e quindi possono essere sfruttate dal traduttore solo per eseguire un type checking sul tipo del risultato

Dichiarazioni Pure di Funzione

- Se una chiamata di funzione non è preceduta da una dichiarazione in forma di prototipo, il traduttore non esegue il type checking sui parametri, ma fa delle ipotesi su quali siano i tipi dei parametri basandosi sui tipi degli argomenti, come spiegheremo meglio nelle pagine successive
- Non vi è alcun vantaggio a usare dichiarazioni tradizionali in nuovi programmi: questo tipo di dichiarazioni esiste solo per motivi di backward compatibility
- Tuttavia, è opportuno che un programmatore conosca, almeno in parte, le dichiarazioni tradizionali, in quanto la loro presenza in C Standard comporta delle conseguenze che si riflettono anche sull'uso dei prototipi

Dichiarazioni Pure Prototipi

- 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, per fare il type checking, il traduttore ha bisogno di sapere solo 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 Prototipi

- Quando una chiamata di funzione è preceduta da una dichiarazione in forma di prototipo della funzione (sia che si tratti di una definizione sia che si tratti di una dichiarazione pura), 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 Prototipi

- 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 e Definizioni Tradizionali

- Nella programmazione C moderna (e in LPS), si usano quasi esclusivamente dichiarazioni e definizioni in forma di prototipo
- Tuttavia è importante avere alcune informazioni relativamente alle dichiarazioni e definizioni *tradizionali*, perché la loro esistenza comporta conseguenze sulle regole del linguaggio
- 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 tradizionale (sia essa una dichiarazione pura o una definizione) e non da dichiarazioni in forma di prototipo, il traduttore effettua il type checking solo sul tipo del risultato della funzione ed effettua la traduzione facendo alcune ipotesi sui tipi dei parametri, basandosi sui tipi degli argomenti presenti nella chiamata
- Il procedimento dettagliato attraverso cui tali ipotesi vengono formulate è abbastanza complesso in quanto dipende da come una specifica implementazione rappresenta i diversi tipi di dato: pertanto non viene studiato in LPS

Dichiarazioni e Definizioni Tradizionali

- Sulla base delle ipotesi fatte, il traduttore genera codice che
  - effettua le opportune conversioni di tipo
  - poi chiama la funzione
- Dunque, se le ipotesi sui tipi dei parametri sono corrette, la chiamata va a buon fine, altrimenti il codice generato avrà un undefined behavior
- Se si vuole approfondire la conoscenza del linguaggio C, è opportuno studiare tutti i dettagli sulle chiamate di funzione non precedute da dichiarazioni in forma di prototipo: il capitolo 9 di **[Ki]** è un buon punto di partenza e **[C99]** fornisce una descrizione completa

Dichiarazioni e Definizioni Tradizionali

- 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
- 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 Non-semplici in *ASM*

- In una precedente presentazione è stata illustrata la realizzazione di routine semplici in *ASM*
- In particolare si è visto come affinché sia possibile il ritorno da una routine, è necessario che al momento di effettuare una chiamata venga memorizzato l'indirizzo di ritorno in una parola, di registro o di memoria
- La realizzazione di una routine priva delle restrizioni cui sono soggette le semplici, richiede inoltre di memorizzare
  - variabili locali
  - parametri, che sono particolari variabili locali
  - risultato che viene restituito attraverso una variabile che in C e in altri *HLL* è nascosta
- Si osservi che in ciascun caso si tratta di dati *locali* che dovrebbero essere usati solo da tale routine e, nel caso di parametri e risultato, dalle routine che la chiamano

Gestione di Dati Locali in *ASM*

- Per memorizzare dati, abbiamo sin qui utilizzato
  - registri
  - memoria allocata staticamente
- Ci proponiamo ora di studiare se e in che modo questi strumenti siano utili anche per rappresentare variabili locali
- Si osservi che sia parole di registro, sia parole di memoria allocate staticamente, sono sempre disponibili per la abstract machine, indipendentemente da quale routine è in esecuzione
  - Da qualunque routine di uno stesso programma si può accedere a tutti i registri e a tutte le parole di memoria disponibili per il programma
  - Parole di registro e di memoria sono accessibili in qualunque momento durante l'esecuzione del programma

Gestione di Dati Locali in *ASM*

- Quindi ci troviamo nella situazione di rappresentare le variabili locali mediante dispositivi che hanno proprietà di località differenti
- In particolare, i linguaggi *ASM* non offrono meccanismi per riservare (ovvero rendere accessibili) registri o aree di memoria ad uso esclusivo di una routine
- Molti sistemi in realtà hanno forme di *protezione della memoria* che riservano l'uso di aree di memoria a specifici programmi, impedendone l'accesso da istruzioni di altri programmi, ma esse operano sempre un controllo degli accessi a livello di programma e mai al più dettagliato livello delle procedure

Gestione di Dati Locali in *ASM*

- La rappresentazione di variabili locali statiche è la meno problematica, in quanto il *lifetime* di tali variabili si estende a tutto il tempo di esecuzione del programma e quindi coincide con quello dei registri e delle parole di memoria allocate staticamente
- Si deve naturalmente rispettare la località spaziale e, non essendoci forme di protezione automatiche, è il programmatore che deve fare attenzione a non utilizzare una stessa parola di registro o memoria per rappresentare variabili di procedure diverse

Gestione di Dati Locali in *ASM*

- Come noto, i registri sono più efficienti della memoria ma l'esiguità del loro numero non permette, di solito, di dedicare uno o più registri alla rappresentazione di variabili usate da una sola routine
- Quindi nella maggior parte dei casi le variabili locali statiche sono rappresentate mediante memoria allocata staticamente

Gestione di Dati Locali in *ASM*

- Le variabili automatiche hanno lifetime limitato al tempo di esecuzione del blocco in cui sono contenute
- Ciò può essere sfruttato per rappresentarle limitando l'occupazione di registri o di memoria
- Ovvero si può cercare di utilizzare le stesse parole di registro o memoria per rappresentare variabili automatiche diverse che appartengano a procedure diverse, a condizione che tali procedure non siano mai attive contemporaneamente
- Il fatto che il lifetime sia limitato, inoltre, rende più semplice usare registri, che sono più efficienti

Gestione di Dati Locali in *ASM*

- Supponiamo che una stessa parola X sia utilizzata per rappresentare due variabili automatiche appartenenti a procedure distinte **A** e **B**
- Ovviamente si deve cercare di fare in modo che **A** e **B** non appartengano ad uno stesso ramo del call tree, ovvero che non esista una successione di chiamate che vada da **A** a **B** o viceversa, altrimenti accadrebbe che una delle due procedure verrebbe eseguita mentre l'altra è sospesa e così facendo potrebbe sovrascrivere il contenuto di X

Gestione di Dati Locali in *ASM*

- Ad esempio si supponga che in una procedura **P** ci si trovi costretti ad utilizzare una parola X che è usata per memorizzare una variabile locale a un'altra procedura che potrebbe essere attiva nel momento in cui inizia l'esecuzione di **P**
- In tal caso è necessario creare una copia temporanea del valore di X all'inizio dell'esecuzione di **P** (*salvataggio di X*) e memorizzare di nuovo in X tale valore al termine dell'esecuzione di **P** (*ripristino di X*)
- La necessità di fare salvataggio e ripristino può capitare più spesso per i registri, sia per via della loro esiguità sia per il fatto che determinate istruzioni di un *ASM-PM* possono richiedere l'uso di specifici registri

Salvataggio e Ripristino dei Registri

- Un caso particolarmente significativo lo troviamo in MIPS32 con il registro `ra`, che la maggior parte delle routine usa per memorizzare il proprio indirizzo di ritorno
- Di ciò si deve tener conto nella realizzazione di routine annidate, in quanto l'esecuzione di una chiamata all'interno di una routine può far sì che la memorizzazione dell'indirizzo di ritorno della routine chiamata, sovrascriva l'indirizzo di ritorno della routine chiamante
- Ad esempio, se la routine **main** chiama la routine **A**, la quale a sua volta chiama la routine **B**, l'istruzione che chiama **B** sovrascrive l'indirizzo di ritorno da **A** memorizzato in `ra`
- È dunque necessario che l'indirizzo di ritorno da **A**, venga copiato da `ra` in una diversa parola, prima di eseguire l'istruzione di chiamata di **B**

Salvataggio e Ripristino dei Registri

- In generale, se una routine **A** chiama una seconda routine **B**, allora **A** deve tenere conto delle modifiche al contenuto dei registri operate da **B**
- Se **B** modifica il contenuto di un registro **R** e **A** necessita che **R** abbia il medesimo contenuto prima e dopo la chiamata di **B**, è necessario:
  1. *salvare* il contenuto di **R**, prima che venga modificato durante l'esecuzione di **B**
  2. al termine dell'esecuzione di **B** (o della parte di **B** che modifica **R**), *ripristinare* il contenuto di **R** al valore che esso aveva prima della chiamata di **B**
- Nella maggioranza dei casi, salvataggio e ripristino possono avvenire o all'interno di **A** o all'interno di **B**

Salvataggio e Ripristino dei Registri

- Salvataggio e ripristino all'interno della *routine chiamante* **A**
  - Permette di fare tali operazioni solo per i registri di cui **A** necessita venga preservato il contenuto, e quindi porta ad avere codice più efficiente in termini di velocità d'esecuzione
  - Ha lo svantaggio che le istruzioni che effettuano tali operazioni, devono essere scritte per ciascuna chiamata di **B**
- Salvataggio e ripristino all'interno della *routine chiamata* **B**
  - Permette di scrivere una sola volta il codice che effettua tali operazioni, ottenendo programmi con meno istruzioni
    - Maggior comodità di scrittura
    - Minor rischio di errori
    - Minore occupazione di memoria

Salvataggio e Ripristino dei Registri

- La necessità di tenere conto delle possibili modifiche apportate al contenuto dei registri dalle routine chiamate, appesantisce notevolmente il lavoro dei programmatori, sopratutto nel caso di programmi sviluppati da più persone
- Per tentare di limitare il problema, si adottano convenzioni nell'utilizzo dei registri
- In MIPS32, esiste una convezione di riferimento, proposta dalla documentazione ufficiale
  - Attribuisce dei ruoli ai vari registri, anche in relazione alla comunicazione tra routine
  - Ripartisce il compito di salvare e ripristinare i registri in parte sulle routine chiamanti e in parte su quelle chiamate
- Per una descrizione completa della convenzione, si veda **[MIPS32]**; nel seguito descriviamo le regole principali

Salvataggio e Ripristino dei Registri

- I registri `v0` e `v1` servono per restituire i risultati di una routine (la lettera "v" sta per "value")
- I registri `a0`, `a1`, `a2`, `a3` sono usati per passare gli argomenti a una routine (la lettera "a" sta per "argument")
- Otto registri, `s0-s7` sono usati per contenere dati di cui una routine chiamante ha bisogno, sia prima che dopo aver effettuato chiamate ad altre routine (la lettera "s" sta per "static")
  - Se tali registri sono necessari sia alla routine chiamante che a quella chiamata, è compito della routine chiamata preservarne il contenuto

Salvataggio e Ripristino dei Registri

- Dieci registri, `t0-t9` sono usati per contenere dati di cui si ha bisogno solo "nel breve termine" ovvero per contenere risultati temporanei di alcune elaborazioni ("t" sta per "temporary")
  - Se tali registri sono necessari sia a una routine chiamante che a una chiamata, è compito della routine chiamante preservarne il contenuto
- Si osservi come, con questa convenzione, il compito di salvare e ripristinare il contenuto dei registri che vengono utilizzati sia da una routine chiamante che da una chiamata, viene ripartito su entrambe

Salvataggio e Ripristino dei Registri

- Adottare una convenzione nell'uso dei registri non è comunque sufficiente a superare le limitazioni dei registri, in modo particolare in *ASM-PM* che hanno un numero limitato di registri
- Ad esempio, in MC68000 ci sono 16 registri a disposizione (tra dati e indirizzi): difficilmente bastano per soddisfare le esigenze di tutte le routine di un programma di media complessità
- La CPU 8086 ha a disposizione 8 registri ...

Activation frame

- In questa presentazione si è dunque visto che la realizzazione di routine non-semplici rende necessario memorizzare, per ciascuna routine, diverse informazioni
- L'insieme delle informazioni che è necessario memorizzare per realizzare una routine **R**, viene detto *activation frame* o *activation record* di **R**, e comprende:
  - Indirizzo di ritorno al chiamante
  - Contenuto dei registri da preservare
  - Contenuto delle variabili locali
  - Argomenti per le procedure chiamate
  - Risultati delle procedure chiamate

Activation frame

- L'activation frame può essere memorizzato in memoria allocata staticamente, o in registri, oppure con un mix di registri e parole di memoria allocate staticamente
- In una prossima presentazione verrà però mostrato che nessuna di queste tecniche è adeguata per un'importante categoria di procedure