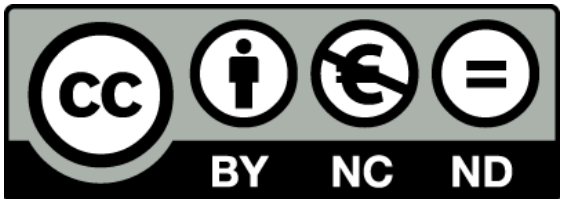
Laboratorio di Programmazione di Sistema

Puntatori in C

Luca Forlizzi, Ph.D.

Versione 23.1



Introduzione

- Una tecnica fondamentale di programmazione è accedere ai dati in modo *indiretto*, ovvero *tramite riferimento*
- Consiste nell'utilizzo di *variabili riferimento* che sono in grado di memorizzare *riferimenti* ad altre variabili
- Se una variabile riferimento R si riferisce ad una variabile V_1, è possibile accedere al dato al contenuto in V_1 tramite R
- Inoltre, poiché R è una variabile, è possibile modificarne il contenuto durante l'esecuzione del programma: ad esempio le si può assegnare un riferimento ad un'altra variabile V_2

Introduzione

- Gli accessi tramite riferimenti aumentano la flessibilità a disposizione del programmatore: attraverso la modifica di variabili riferimento, si può stabilire durante l'esecuzione del programma quali sono le variabili a cui determinate istruzioni accedono
- Inoltre permettono una gestione più efficiente di dati che richiedono una quantità di memoria superiore a quella usata da dati di tipo base, come spesso accade con dati aggregati
  - In tali casi, infatti, un riferimento a un dato richiede una quantità di memoria sensibilmente inferiore di quella richiesta dal dato stesso
  - Quindi in determinate operazioni, come assegnamenti o passaggi di parametro, usare riferimenti ai dati risulta più efficiente che dell'uso diretto dei dati

Introduzione

- In Java, come in molti *HLL*, i riferimenti sono *opachi*, ovvero il linguaggio non consente di effettuare operazioni di modifica dei riferimenti
- Le operazioni principali sui riferimenti in Java
  - Creazione di un nuovo oggetto e di un riferimento ad esso, attraverso l'operatore `new`
  - Copia di un riferimento in una variabile riferimento, attraverso l'operatore di assegnamento
- La maggior parte delle operazioni su una variabile riferimento, agiscono, tramite il riferimento, sul dato riferito

Introduzione

- Nei linguaggi assembly i riferimenti sono costituiti da indirizzi di memoria
- Pertanto, al contrario di quanto accade nella maggioranza degli *HLL*, è possibile modificare in molti modi un riferimento
- Infatti, per ogni *ASM-PM*
  - Il formato degli indirizzi di memoria è noto (alcuni *ASM-PM* hanno più di un formato per indirizzi di memoria)
  - Gli indirizzi di memoria possono essere memorizzati in parole
  - Il contenuto delle parole può ovviamente essere modificato
- Dunque, un indirizzo memorizzato in una parola P
  - Può essere usato per accedere al dato riferito
  - Può essere modificato tramite qualunque istruzione che può accedere a P

Introduzione

- Nel linguaggio C, i riferimenti sono chiamati *puntatori* e si presentano come un costrutto che ha un livello di astrazione intermedio tra quello degli indirizzi di memoria e quello dei riferimenti opachi tipici degli *HLL*
- Un puntatore è un valore che *si riferisce* ad una variabile
- Nella terminologia di C Standard, se P è un puntatore che si riferisce ad una variabile X, si dice che P *punta* X, ovvero che X *è puntata* da P

Introduzione

- Oltre che a una variabile, un puntatore può riferirsi anche a ciò che C Standard chiama *oggetto*
- È opportuno sottolineare subito che C Standard attribuisce al termine *oggetto* in un significato diverso da quello che tale termine assume nel contesto di Java e di altri linguaggi di programmazione ad oggetti
- Un *oggetto* in C Standard è un entità molto più semplice rispetto agli oggetti di Java, ovvero è un'area di memoria che può contenere dati ma che, diversamente dalle variabili, non ha un tipo stabilito in modo statico

Introduzione

- Ciò vuol dire che un programma C, durante l'esecuzione, può utilizzare lo stesso oggetto per memorizzarvi, in momenti diversi, dati che hanno tipo diverso
- Gli oggetti verranno studiati in maggior dettaglio in future presentazioni, per il momento ci limitiamo a dire che gli oggetti di C Standard
  - vengono utilizzati proprio mediante puntatori
  - sono aree di memoria allocate dinamicamente

Introduzione

- Se si vuole realizzare un programma C portabile, in generale ci si deve limitare ad operazioni sui puntatori simili a quelle dei riferimenti in Java
  - Creazione di un puntatore che si riferisce ad una certa variabile o ad un certo oggetto
  - Copia di un puntatore in una variabile puntatore, attraverso l'operatore di assegnamento
  - Accesso alla variabile o all'oggetto puntato
- Tuttavia su alcuni puntatori, ovvero quelli che puntano elementi di array, possono essere eseguite alcune operazioni di modifica senza compromettere la portabilità del programma
- Tali operazioni sono definite dall'*aritmetica dei puntatori* e verranno discusse in una futura presentazione

Introduzione

- Se si è disposti a rinunciare alla portabilità, è possibile fare su un puntatore le stesse operazioni che i linguaggi assembly consentono di fare sugli indirizzi di memoria
- Infatti C Standard consente le conversioni tra puntatori e valori interi
  - Il risultato di tali conversioni è implementation defined
  - Un puntatore, una volta convertito in un intero che abbia una dimensione opportuna per l'implementazione di interesse, può essere modificato attraverso tutte le operazioni permesse su un intero, e poi convertito di nuovo in puntatore

Tipi Puntatore

- Ogni puntatore, può riferirsi solo a variabili che hanno un determinato tipo
- L'insieme dei puntatori che possono riferirsi a variabili di tipo **T** costituisce a sua volta un tipo, chiamato *tipo puntatore a* **T**
- Ogni tipo puntatore contiene un valore speciale, detto *null pointer*
- Per un certo tipo **T**, i valori di tipo puntatore a **T** sono:
  - il *null pointer*
  - i puntatori a variabili di tipo **T**, ottenuti applicando a tali variabili l'*operatore di indirizzo* &
  - i puntatori a oggetto di tipo **T** che possono essere restituiti come risultato da funzioni

Tipi Puntatore

- Il *null pointer* viene usato per indicare che una variabile di tipo puntatore non contiene un riferimento ad un oggetto
- Il valore *null pointer* del tipo puntatore a **T** risulta diverso da qualunque puntatore ad oggetti o variabili di tipo **T**
- Il *null pointer* può essere rappresentato nel codice sorgente come
  - la costante `0`
  - l'espressione `(void *)0`
  - la macro `NULL` definita nell'header `<stddef.h>`

Tipi Puntatore

- Per ottenere un puntatore ad una variabile `v` di tipo **T**, si applica a `v` l'operatore di indirizzo
- L'operatore di indirizzo è un operatore prefisso, ed è indicato dal simbolo `&`
- Quindi se `v` è una variabile di tipo **T**, l'espressione `&v` ha tipo puntatore a **T** ed è un puntatore a `v`
- Esempi

```
double d;
int h;
&d;       // puntatore a d, tipo (double *)
&h;       // puntatore a h, tipo (int *)
```

Dichiarazione di Puntatori

- Una variabile di tipo puntatore a **T** viene dichiarata in una dichiarazione di variabili di tipo **T**, scrivendo il nome della variabile puntatore preceduto da *
- La dichiarazione di una variabile di tipo puntatore a **T**, crea una variabile puntatore a **T**
- Esempi

```
// x e y sono int, p1 e p2 puntatori a int
int x, *p1, y, *p2;

// f è float, p3 e p4 puntatori a float
float *p3, f, *p4;
```

Dichiarazione di Puntatori

- La variabile creata può essere inizializzata ad un valore puntatore a **T**, con la consueta sintassi per le inizializzazioni
- Una variabile esterna non inizializzata in modo esplicito, viene inizializzata al *null pointer*
- Una variabile locale non inizializzata, ha un contenuto indefinito che potrebbe non essere un valore valido di tipo puntatore a **T** (in particolare, potrebbe non essere il *null pointer* del tipo puntatore a **T**)
- Esempi di dichiarazioni esterne

```
// p1 e p3 inizializzati con null pointer
// p2 inizializzato con puntatore a x
// p4 inizializzato con puntatore a f
int x = 6, *p1, y;
int *p2 = &x;
float *p3 = 0, f = -4.3f, *p4 = &f;
```

Puntatori Generici

- Per ogni tipo di dato **T** di C Standard, è possibile costruire un puntatore ad oggetti di tipo **T**, che può ovviamente riferirsi a variabili di tipo **T**
- A volte, però, è utile poter memorizzare in una variabile un puntatore *generico*, che punta un oggetto ma non ne specifica il tipo
- È possibile dichiarare un puntatore generico `p` con la dichiarazione `void *p;`
- Per via della forma della dichiarazione, i puntatori generici vengono, informalmente detti "puntatori a `void`"
- Il *null pointer* non va confuso con un puntatore a void: il primo è un valore, il secondo un tipo
- In future presentazioni mostreremo alcune applicazioni dei puntatori generici

Dereferenziazione di Puntatori

- Un puntatore che si riferisce ad un oggetto può essere usato per accedere all'oggetto con l'*operatore asterisco*
- L'operatore asterisco è un operatore prefisso, ed è indicato dal simbolo $*$
- Può essere applicato ad un valore di tipo puntatore, di solito al contenuto di una variabile di tipo puntatore
- Esempi

```
char c1 = 'X', c2, *p = &c1;
// la variabile puntatore p è inizializzata
//  con un puntatore a c1
printf( "%c␣%c\n", c1, *p); // stampa "X X"
c2 = 'E';
// applicazione di * ad un puntatore
// non contenuto in una variabile
printf( "%c\n", * &c2 ); // stampa "E"
```

Dereferenziazione di Puntatori

- Applicare l'operatore asterisco a un *null pointer*, causa un undefined behavior
- Applicare l'operatore asterisco ad una variabile puntatore il cui contenuto non è un valore valido (quindi diverso dal *null pointer*), causa un undefined behavior
- Applicare l'operatore asterisco ad una variabile di tipo `void *` causa un undefined behavior, in quanto tali variabili hanno lo scopo di contenere dei puntatori ma non di usarli

```
void f( void ) {
  short int *p1 = 0, *p2, x;
  void *p3 = &x;
  x = *p1;        // undefined behavior
  *p2 = 2;        // undefined behavior
  *p3 = 3;        // undefined behavior
}
```

Assegnamento di Puntatori

- L'operatore di assegnamento consente di copiare un puntatore in una variabile puntatore
- L'assegnamento tra puntatori è ben definito nei seguenti casi
  - per un certo tipo **T**, l'operando sinistro è una variabile di tipo puntatore a **T** e l'operando destro ha tipo puntatore a **T**
  - l'operando sinistro è una variabile di tipo `void *`
  - l'operando destro ha tipo `void *`
- Altrimenti si ha constraint violation

```
long x = 12, *p1;
int w = 14, *p2 = &w;
void *p3;
p1 = &w;          // constraint violation
p1 = p2;          // constraint violation
p3 = &x;          // corretto
p2 = p3;          // corretto
```

Assegnamento di Puntatori

- L'assegnamento, permette di memorizzare in una variabile puntatore un *null pointer* o un puntatore ad una variabile, durante l'esecuzione del programma
- Attenzione a non confondere dal punto di vista sintattico un assegnamento con una dichiarazione che ha un'inizializzatore
  - nel primo caso non ci deve essere * (altrimenti si tenterebbe di applicare l'operatore asterisco)
  - nel secondo caso ci deve essere * prima del nome della variabile, per indicare che si sta dichiarando un puntatore

```
unsigned x = 5, y = 6, *p1, *p3, *p4;
// la riga seguente è una dichiarazione
unsigned *p2 = &x;
// le righe seguenti sono invece assegnamenti
p3 = &y;          // assegna a p3
p1 = 0;           // assegna null pointer
p4 = p3;          // assegna contenuto p3 in p4
```

Assegnamento di Puntatori

- Se il contenuto di una variabile puntatore p viene assegnato ad un'altra variabile puntatore q, dopo l'assegnamento p e q puntano lo stesso oggetto

```
int x, *p, *q;
p = &x;
q = p;
*p = 2;
// stampa "2 2 2"
printf( "%d␣%d␣%d\n", x, *p, *q );
*q += 3;
// stampa "5 5 5"
printf( "%d␣%d␣%d\n", x, *p, *q );
```

Assegnamento di Puntatori

- Si presti attenzione alla differenza tra un assegnamento tra puntatori e un assegnamento tra gli oggetti puntati da due puntatori
- Esempio

```
int x, y = 4, *p = &x, *q = &y;
*p = *q;
// stampa "4 4"
printf( "%d␣%d\n", *p, *q );
*p = 6;
// stampa "6 4"
printf( "%d␣%d\n", *p, *q );
p = q;
// stampa "4 4"
printf( "%d␣%d\n", *p, *q );
*p = 6;
// stampa "6 6"
printf( "%d␣%d\n", *p, *q );
```

Valori Booleani e Puntatori

- È possibile applicare gli operatori == e != ad una coppia di puntatori dello stesso tipo
- Se due puntatori puntano lo stesso oggetto, essi risultano uguali
- Di solito è vero anche il viceversa, ma c'è un'eccezione, che verrà discussa in una futura presentazione
- Il valore *null pointer* del tipo puntatore a **T** risulta diverso da qualunque puntatore ad oggetti o variabili di tipo **T**

```
int x = 0, y = 1, z = 0, *p, *q;
p = 0; q = &x;
if ( p == q ) y -= 1; else y += 2;
printf( "%d", y ); // stampa 3
p = &x;
if ( p != q ) z = 666;
printf( "%d", z ); // stampa 0
```

Valori Booleani e Puntatori

- Le espressioni di controllo delle istruzioni `if`, `while`, `do-while`, `for`, possono avere un tipo intero (come già sappiamo) oppure un tipo puntatore
- Un'espressione di controllo che ha un tipo puntatore viene considerata falsa se e solo se il valore dell'espressione è il *null pointer*

```
int x = 9, y = 0, *p = &x;
if ( p ) y -= 2; else y += 2;
printf( "%d", y ); // stampa -2
if ( !p ) y -= 3; else y += 3;
printf( "%d", y ); // stampa 1
y = 10;
p = 0;
if ( p ) y -= 2; else y += 2;
printf( "%d", y ); // stampa 12
if ( !p ) y -= 3; else y += 3;
printf( "%d", y ); // stampa 9
```

Conversioni tra Puntatori

- Come sappiamo, C Standard consente assegnamenti tra un puntatore ad un tipo **T** e un puntatore a `void`
- In tali assegnamenti è implicita una conversione di tipo, da puntatore a **T** in puntatore a `void` o viceversa
- Non sono invece consentiti assegnamenti tra puntatori a oggetti che hanno tipi diversi, ovvero non vi sono conversioni implicite tra puntatori di tipi diversi, a meno che uno dei due tipi sia puntatore a `void`
- Tuttavia, dati due qualunque tipi **T** e **R**, tramite l'operatore di cast è possibile convertire in modo esplicito un puntatore a **T** in un puntatore a **R**
- In generale, tali conversioni potrebbero produrre puntatori non validi (per violazione di vincoli di allineamento) e di conseguenza causare undefined behavior

Conversioni tra Puntatori

- È però sempre possibile convertire un puntatore P a oggetti di tipo **T** che punta ad un oggetto O, in un puntatore P' a oggetti di tipo carattere
- P' può essere dereferenziato per accedere al byte di indirizzo di memoria minimo tra quelli che formano O
- A partire da P', mediante operazioni permesse dell'aritmetica dei puntatori che verranno introdotte in una futura presentazione, è possibile accedere anche agli altri byte di O

```
long l1 = 0x12345678, *p1 = &l1;
unsigned char *p2;
p2 = (unsigned char *) p1;
/* l'istruzione seguente stampa il byte di l1
   che ha indirizzo minimo */
printf( "%hhx", *p2 );
```

Conversioni tra Puntatori

- Una conversione esplicita è possibile anche tra puntatori a oggetti di tipo **T** e interi
- Il risultato di tali conversioni è *implementation defined*, quindi per definizione non portabile
- Essendo detto risultato un intero, può essere modificato in molti modi e poi convertito di nuovo in puntatore
- Ovviamente il comportamento di un programma che faccia operazioni di questo tipo dipende dalla particolare implementazione su cui viene eseguito

```
float f, *p1, *p2 = &f;
unsigned long long w;
p1 = 0x1000;     // constraint violation
p1 = (float *) 0x1000;
w = (unsigned long long) p2;
w = w & 0xfffffffc;
p2 = (float *) w;
```

Puntatori e Tipi Aggregati

- È possibile referenziare mediante un puntatore anche un elemento di un array o un membro di una struttura, purché abbiano il tipo corretto

```
float *p1, *p2, *p0, b[3] = { 4.1f, 1.2f };
struct { int m1; float m2 } s = { 1, 2.5f };

p1 = &b[1];    /* p1 punta b[1] */
p2 = &b[2];    /* p2 punta b[2] */
p0 = &s.m2;    /* p0 punta s.m2 */
*p2 = b[0] + *p1 + *p0;
printf("%f", b[2]); /* stampa 7.8 */
```

Puntatori e Strutture

- È possibile definire puntatori a tipi struttura

```
struct type1 { int m1; long m2 };
struct type1 s1 = { -1, 100000 }, s2, *ps;

ps = &s1;  /* ps punta s1 */
s2 = *ps;  /* copia *ps (ovvero s1) in s2 */
```

Puntatori e Strutture

- Attraverso un puntatore a una struttura `ps`, si può accedere ai membri della struttura utilizzando due operatori in sequenza
  1. Applicando `*` a `ps` si accede alla struttura puntata
  2. Applicando `.` a `*ps` si accede ai membri della struttura
- Ovviamente, è necessario indicare, attraverso la sintassi, che si deve applicare prima `*` e poi `.`

Puntatori e Strutture

- Si osservi a tal proposito che `*` è un operatore prefisso mentre `.` è un operatore postfisso
- In base alle regole sintattiche di C Standard, gli operatori postfissi hanno sempre precedenza maggiore rispetto ai prefissi
- Quindi, se `ps` è un puntatore a una struttura che ha un membro `m`, non è corretto accedere al membro `m` della variabile puntata da `ps` mediante l'espressione `*ps.m`
- Tale espressione è infatti constraint violation, in quanto applica l'operatore `.` alla variabile `ps`, ma ciò è sintatticamente scorretto in quanto `ps` non ha un tipo struttura

Puntatori e Strutture

- Per accedere ad un membro di una struttura attraverso un puntatore è necessario scrivere delle parentesi tonde attorno all'operatore * applicato al puntatore: ciò non è molto comodo e il codice ha un aspetto poco elegante

```
struct type1 { int m1; long m2 };
struct type1 s = { 0, -800 }, *ps = &s;
long v;
/* nella istruzione seguente si accede al
   membro m2 della struttura puntata da ps */
v = (*ps).m2;
```

Puntatori e Strutture

- Per rendere più comodo da leggere e da scrivere il codice che usa un puntatore per accedere ai membri di una struttura, C Standard definisce un operatore apposito, indicato da una coppia di simboli `->` e talvolta chiamato informalmente "operatore freccia"
- Se `p` è un puntatore a una struttura che ha un membro `m`, l'operazione `p -> m` è semanticamente equivalente a `(*p).m`

```
struct type1 { int m1; long m2 };
struct type1 s = { 0, -800 }, *ps = &s;
long v;
/* nella istruzione seguente si accede al
   membro m2 della struttura puntata da ps */
v = ps -> m2;
```

Puntatori e Strutture

- Lavorando su strutture attraverso puntatori, si ottiene una semantica simile a quella di Java

```
struct type1 { int m1; long m2 };
struct type1 s = { 1, 100000 }, *ps1, *ps2;

ps1 = &s;    /* ps1 punta s */
(*ps1).m1 = 2;
printf("%d", s.m1 ); /* stampa 2 */
ps2 = ps1;
ps2 -> m1 += 2;
printf("%d", ps1 -> m1 ); /* stampa 4 */
printf("%d", s.m1 ); /* stampa 4 */
```