Il linguaggio C

Puntatori e Array

Puntatori

- I puntatori sono variabili i cui valori sono indirizzi di locazioni in cui sono memorizzate altre variabili
    - architettura a 32 bit: 2^{32}-1 indirizzi, ma non si usa unsigned int come tipo (tipo=valori + operazioni)
    - hanno senso somme e sottrazioni, non altre operazioni
    - soprattutto si definisce l'operazione per referenziare la variabile identificata dal valore del puntatore

Dichiarazione puntatori

- si usa il modificatore * prima del nome della variabile, es.:

```
int *pA;  /* pA è puntatore a
int */

int **ppA; /* ppA è puntatore
a puntatore a int */
```

Sintassi puntatori

- Sintassi di `<decl>` che tiene conto dei puntatori:

```
<decl>::= <identifier> |
*<decl> | ...
```

Esercizio

- Costruire l'albero sintattico di:

```
int **ppX;
```

Operazioni aritmetiche

- Hanno senso incrementi e decrementi
- Il valore cambia di multipli del numero di byte necessari a rappresentare il tipo del puntatore:

```
int *pInt;
...
pInt += 2; // aggiunge
2*sizeof(int) al valore di pInt
```

Operatore dereferenziazione

- * applicato in forma prefissa ad un'espressione che restituisce un valore di tipo puntatore, es.:

```
int *pA;
....
....
*pA = 12; /* assegna 12 alla
variabile puntata da pA */
```

Sintassi dereferenziazione

- `<var>::= <identifier> | *<expr> | ...`
  - in cui `<expr>` restituisce un valore di tipo indirizzo di una variabile di un qualche tipo
  - Quindi `*pA = ...` è legale

Operatore di indirizzo

- L'operatore `&` è all'incirca l'inverso di `*`
- `<expr> ::= ... | &<var> | ...`
  - Γ(&<var>) restituisce il valore dell'indirizzo della variabile referenziata da `<var>`. Non ha side effects.
    - `&A = ...` non è legale (compara la sintassi con quella di `*A`)!

Puntatori a void

- Si possono dichiarare puntatori di tipo void, es.:

```
void *ptr;
int n;
ptr = &n;
```

- il compilatore sa quanti byte servono per rappresentare un indirizzo: posso usare &...

- Per dereferenziare un puntatore void devo fare un cast
- come farebbe il compilatore a sapere quanti byte leggere...

```
void *ptr;
int m,n;
ptr = &n;
m=*((int *)ptr);
```

Sintassi cast di puntatore

- `<expr>::= ... | (<cast-type>) <expr1> | ...`
- `<cast-type> ::= <type> | <cast-type> *`

Array

- Insieme di variabili, dello stesso tipo, che possono essere referenziate con un nome collettivo ed un indice che le distingue tra loro
  - si estende `<decl>`
- `<declaration>::=<type><decl>;`
  `<decl>::=<identifier>|*<decl>`
  `| <decl>[<const>]`
- `<type> <decl>[<const>];`

Ordine di * e []

- Il modificatore suffisso `[<const>]` ha priorità sul modificatore prefisso `*`

`float *X[128];` è un array di 128 puntatori a float

Interpretazione di dichiarazioni complesse

- Si parte dal nome della variabile
  nome = nome è un...
- Si legge verso destra finché ci sono
  modificatori suffissi
  [<const>] = ...array di <const> variabili di tipo...
- Si legge verso sinistra fino ad arrivare al tipo elementare
  * = ...puntatore ad una variabile di tipo...

Esempio

- float **XX[12][18];
- XX è un - array di 12 variabili di tipo - array di 18 variabili di tipo - puntatore a variabile di tipo - puntatore a variabile di tipo - float
- XX è un array di 12 array di 18 puntatori a puntatori float

Scrivere dichiarazioni complesse

- Si segue la regola di lettura all'inverso.
  Es.: dichiarare X come array di 18 puntatori a int:

1. ... X ... // X....
2. ... X[18] ... // X è un array di 18...
3. ...*X[18]; // X è un array di 18 puntatori a...
4. int *X[18]; // X è un array di 18 puntatori a int

Estensione della sintassi

- La categoria `<decl>` usa le parentesi per forzare la priorità di `*` e `[]`

```
<decl>::= ...|(<decl>)|...
```

- La parentesi da la priorità ai modificatori interni: ora si può dichiarare un puntatore ad array!

Esempio

- int (* ptr)[128];
- ptr è un - puntatore a - array di 128 variabili di tipo - int

Riferimento ad array

- Il nome associato ad una variabile array denota il valore (costante) dell'indirizzo a partire dal quale l'array è memorizzato
- il tipo di tale valore è quello di un puntatore a variabili del tipo di quelle raccolte nell'array

Sintassi

- `<var>::= <identifier> | *<expr>`
  `| <expr1>[<expr2>] | ...`
  - <expr1> restituisce un valore di tipo indirizzo
  - <expr2> restituisce un valore di tipo intero

Semantica

- Se *ptr* e *n* sono i valori restituiti da <expr1> e <expr2>, e se *t* è il tipo di variabile puntata da *ptr* allora:
  <expr1>[<expr2>] denota una variabile di tipo *t* che si trova all'indirizzo *ptr+n*, con + intesa con la semantica della somma dei puntatori
- <expr1>[<expr2>] == *(<expr1>+<expr2>)

Array multidimensionali

- Le variabili sono serializzate nell'ordine riga colonna
- *tipo* `C[X][Y];`
  [i][j] indica l'elemento che si trova alla posizione i*Y+j
  `C[i][j]` si trova nella locazione C+i*Y+j
- discende da sintassi e semantica (array di array)

- Un array multidimensionale si riporta comunemente ad un array monodimensionale
- Esercizio:
  `float A[X][Y][Z];`
  `A[i][j][k]` corrisponde all'elemento...

- Un array multidimensionale si riporta comunemente ad un array monodimensionale
- Esercizio:
  `float A[X][Y][Z];`
  `A[i][j][k]` corrisponde all'elemento...

```
i*Y + j*Z + k
```

Allocazione dinamica

- la funzione di libreria *malloc()* richiede al S.O. l'allocazione di un array di variabili char di dimensione determinata. Restituisce il puntatore (void) alla locazione di base dell'array:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Esempio

- `float *A;`
- `A = (float *)`
  `malloc(20*sizeof(float));`
  - `sizeof(float)` restituisce la dimensione di un float
  - il cast a `(float *)` del puntatore restituito da malloc consente di usarlo con `A`

- se il S.O. non ha a disposizione un segmento consecutivo di memoria sufficientemente grande *malloc()* fallisce
- riporta `NULL`
  definito come `((void *)0)`

Puntatori e array: differenze

- `float xA[128];`
- `float *xP;`
  `xP = (float *)`
  `malloc(128*sizeof(float));`
- `xA` e `xP` restituiscono un valore di tipo indirizzo di float; `xA` è una costante, `xP` una variabile (può apparire a sinistra di =)

Deallocazione

- Quando la memoria allocata non serve più la si libera con la funzione di libreria *free()*

Es.:

```
int *C;
...
C=(int *)malloc( X*sizeof(int));
...
free(C);
```

Memory debugging

- Alcuni problemi: uso di indici oltre il limite dell'array, mancate deallocazioni (memory leak), riuso di puntatori deallocati, uso di puntatori non inizializzati, buffer overflow...
- esistono librerie che aiutano ad identificare questi errori, es.: DMalloc, Valgrind, ElectricFence, etc.