Lezione 13

Laura Bozzelli

a.a. 2020/2021

Sommario - Lezione 13: Puntatori

- Puntatori.
- Operatori di indirizzamento e dereferenziazione.
- Puntatori e funzioni.
- Qualificatore const e suo utilizzo per puntatori.
- Aritmetica dei puntatori.
- Confronto tra puntatori.
- Relazione tra puntatori e array.
- Operatore sizeof
- Operatori bit a bit.

Puntatori (1/3)

- I puntatori rappresentano uno dei costrutti più potenti del linguaggio C.
- Fanno parte delle funzionalità del C più difficili da padroneggiare.
- Consentono di realizzare il passaggio per riferimento favorendo la scrittura di codice compatto ed efficiente.
- Sono fondamentali nel linguaggio C per creare e manipolare **strutture dati dinamiche** , e cioè strutture dati la cui ampiezza può aumentare e diminuire durante l’esecuzione: ad esempio, liste concatenate, code, pile, ed alberi.

Puntatori (2/3)

Abbiamo visto che una variabile è caratterizzata da quattro elementi:

- Nome (identificatore)
- Tipo
- Locazione di memoria
- Valore (contenuto corrente della locazione di memoria)

Un puntatore è una variabile i cui possibili valori sono indirizzi (iniziali) di (locazioni di) memoria.

Puntatori (3/3)

- In generale, un valore ‘valido’ di un puntatore è l’indirizzo di memoria iniziale associato alla locazione di memoria di un’altra variabile. Quest’ultima può essere a sua volta una variabile puntatore (**puntatori di puntatori**), e così via.

- I **puntatori sono tipizzati**: ogni puntatore punta ad un dato di un certo tipo. Fanno eccezione i puntatori a **void** che vengono utilizzati per supportare qualsiasi tipo di puntatore.

- Per ogni tipo di dato predefinito e per ogni tipo aggregato (struttura o unione) o tipo enumerativo, è possibile definire una variabile puntatore di quel tipo.

Dichiarazione di puntatore a dati(1/2)

Sintassi dichiarazione di singola variabile puntatore a dati

<nome tipo> * <nome puntatore>;

- **<nome tipo>**: corrisponde o ad un tipo predefinito, o ad un tipo aggregato (struttura o unione) o ad un tipo enumerativo. **<nome tipo>** può essere anche la parole chiave **void**. I puntatori di tipo void vengono utilizzati per supportare qualsiasi tipo di puntatore.
- **<nome puntatore>**: identificatore valido.

Esempio: dichiarazione di un puntatore ad un intero **int**.

```
int * countPtr;
```

Dichiarazione di puntatore a dati (2/2)

Sintassi dichiarazione di multipli puntatori a dati

<nome tipo> * <nome puntatore 1>, …, * <nome puntatore N>;

La notazione con l'asterisco * usata per dichiarare puntatori a dati non viene distribuita a tutti i nomi delle variabili in una dichiarazione. Ogni puntatore deve essere dichiarato con il simbolo * che precede il nome.

Esempio:

```
int * xPtr, * yPtr;
```

Dichiarazione di puntatore a puntatore

Sintassi dichiarazione di singolo puntatore a puntatore

<nome tipo> ** <nome puntatore>;

- **<nome tipo>**: corrisponde o ad un tipo predefinito, o ad un tipo aggregato (struttura o unione) o ad un tipo enumerativo. **<nome tipo>** può essere anche la parole chiave **void**.
- **<nome puntatore>**: identificatore valido.

Sintassi dichiarazione di multipli puntatori a puntatore

<nome tipo> ** <nome puntatore 1>, … , ** <nome puntatore N>;

Esempio:

```
int ** xPPtr, ** yPPtr;
```

Inizializzazione e assegnazione di puntatori (1/3)

Un valore ‘valido’ di un puntatore **fa indirettamente riferimento (e, cioè esso punta)** ad un valore avente come tipo il tipo del puntatore e corrispondente al contenuto della locazione di memoria (sequenza di celle di memoria) univocamente determinata da:

- L’indirizzo della prima cella di memoria (informazione fornita dal valore del puntatore).
- Il numero di celle di memoria (informazione fornita dal tipo del puntatore).

Inizializzazione e assegnazione di puntatori (2/3)

I puntatori devono essere inizializzati quando sono dichiarati (alias definiti) oppure assegnando loro un valore tramite un'istruzione di assegnazione. Un valore valido di un puntatore può essere specificato in due modi:

- Tramite l'operatore di indirizzamento & (lo vedremo nelle slide seguenti).
- Tramite chiamate di funzioni di libreria per l'allocazione dinamica di memoria (lo vedremo in seguito).

Inizializzazione e assegnazione di puntatori (3/3)

Ad una variabile puntatore può essere assegnato il valore 0. Il linguaggio C garantisce che **0 non sia mai un indirizzo valido**. Il valore 0 non punta a niente e rappresenta l'unico valore che si può assegnare direttamente a una variabile puntatore.

- La costante simbolica NULL viene spesso usata al posto dello zero come nome mnemonico per indicare che questo, per un puntatore, è un valore speciale.
- La costante simbolica NULL è definita in <stdio.h>.
- Inizializzare un puntatore a 0 equivale ad inizializzare un puntatore a NULL, ma NULL è preferibile poiché evidenzia il fatto che la variabile è un puntatore.

Operatori per i puntatori: & e * (1/5)

Operatore di indirizzamento &

Il C fornisce l'operatore unario & per ottenere l'indirizzo di memoria della locazione di memoria associata ad un dato.

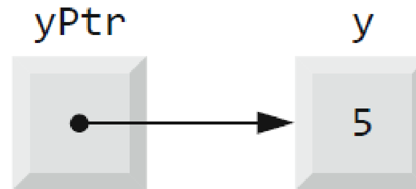
- L'operando deve essere un **left value** e, cioè, un'espressione a cui è associata una locazione di memoria e che, dunque, può occorrere come operando sinistro di un'operazione di assegnazione:
  - un nome di variabile,
  - un elemento di un array specificato tramite l'operatore di indicizzazione,
  - il campo di una struttura o di un unione.

- Il valore restituito dall'operatore & può essere assegnato ad una qualsiasi variabile puntatore il cui tipo corrisponde a quello dell'operando a cui è applicato &.

Operatori per i puntatori: & e * (2/5)

Esempio per operatore di indirizzamento &

```
int y=5;
int * yPtr;

//Assegna l'indirizzo della variabile y
// alla variabile puntatore yPtr
yPtr = &y;
```



Rappresentazione di y e yPtr in memoria assumendo che l’indirizzo di memoria di y sia 600000 e l’indirizzo di memoria di yPtr sia 500000

Operatori per i puntatori: & e * (3/5)

Operatore di dereferenziazione o di indirezione *

Il C fornisce l'operatore unario * per ottenere il valore della locazione di memoria alla quale punta il suo operando (un puntatore).

- L'operando deve essere una variabile puntatore avente un valore valido (in particolare, diverso da NULL). Applicare l'operatore * ad un puntatore equivale a **deferenziare un puntatore**.

```
int y=5;
int * yPtr;

//Assegna l'indirizzo della variabile y
// alla variabile puntatore yPtr
yPtr = &y;

//Stampa il valore della variabile y, cioè 5
printf("%d",*yPtr);
```

Operatori per i puntatori: & e * (4/5)

Operatore di dereferenziazione o di indirezione *

- Deferenziare un puntatore che non è stato correttamente inizializzato o a cui non è stato assegnato un indirizzo valido (ad esempio, un puntatore il cui valore è NULL) è un errore. Ciò potrebbe determinare un **errore irreversibile** in fase di esecuzione o potrebbe causare la modifica accidentale di dati importanti.
- La deferenziazione di un puntatore può essere gestita alla stessa stregua di una variabile: essa può ricorrere in qualsiasi contesto in cui una variabile del tipo referenziato dal puntatore può occorrere. In particolare, essa può ricorrere come operando sinistro in un'operazione di assegnazione (**left value**). Ad esempio, la deferenziazione di un puntatore di tipo intero può occorrere all'interno di un'espressione numerica.

Operatori per i puntatori: & e * (5/5)

Esempio

```
int x=1, y=2, z[10];
int * ip;  //ip è un puntatore ad un intero
ip = &x; //ora ip punta a x
y= *ip   //ora y vale 1
*ip = 0; // ora x vale 0
ip =&z[0] // ora ip punta a z[0]
*ip = *ip +10 //incrementa *ip e cioè z[0] di 10
y = *ip +1; // assegna a y il valore di z[0] più 1
*ip += 1; //incrementa di uno l'oggetto puntato da ip e cioè z[0]
(*ip)++; // incrementa di uno l'oggetto puntato da ip e cioè z[0]
```

Poiché i puntatori sono delle variabili essi possono essere usati senza deferimento.

```
//Inizializza il puntatore iq con il
// valore del puntatore ip.
//Dopo l'inizializzazione ip e iq puntano allo stesso oggetto
int  * iq = ip;
```

Esempio di puntatore a puntatore (1/2)

Lo specificatore di conversione %p di **printf** invia in uscita l'indirizzo di memoria come un intero esadecimale.

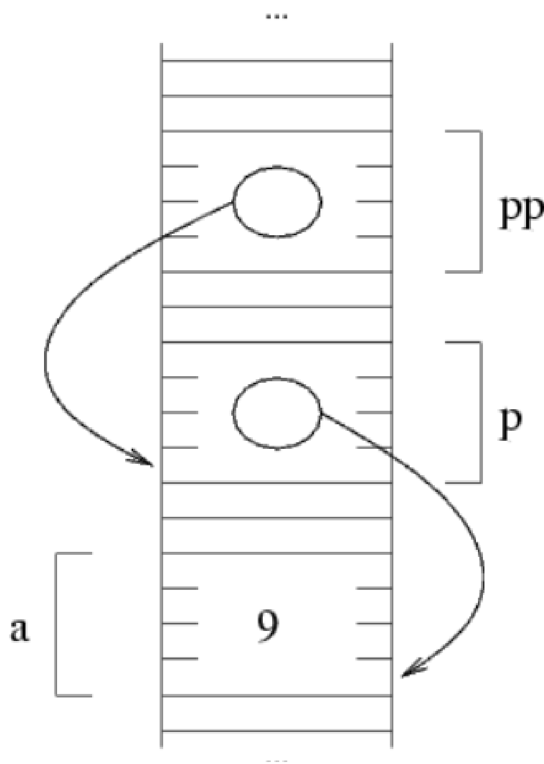
```
int main()
{
  int a;
  int *p;
  int **pp;

  a=9;
  p=&a; //p punta ad a
  pp=&p; //pp punta a p

  //Stampa l'indirizzo ed il valore di pp
  printf("Indirizzo di pp=%p, valore=%p\n", &pp, pp);

  //Stampa l'indirizzo ed il valore di p
  printf("Indirizzo di  p=%p, valore=%p\n", &p, p);

  //Stampa l'indirizzo ed il valore di a
  printf("Indirizzo di  a=%p, valore=%i\n", &a, a);

  return 0;
}
```

OUTPUT

```
Indirizzo di pp=000000000062FE08, valore=000000000062FE10
Indirizzo di  p=000000000062FE10, valore=000000000062FE1C
Indirizzo di  a=000000000062FE1C, valore=9
```

Esempio di puntatore a puntatore (2/2)

- In riferimento all’esempio precedente, il valore della variabile **pp** coincide con l'indirizzo della variabile **p**, e il valore di **p** coincide con l'indirizzo di **a**.



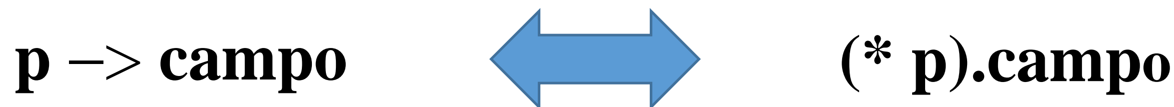
- Dato il valore di **pp** è chiaro che è possibile accedere al valore di **a**: basta seguire i puntatori, ossia prima trovare il valore di ***pp**, che è l'indirizzo di **a**, e questo permette di trovare il valore di **a** usando ancora l'operatore *. Quindi, dato **pp**, il valore di a si può trovare con ****pp**.
- In questo modo si può anche assegnare un valore alla variabile **a** usando **pp**: basta usare una istruzione del tipo ****p = ...** .

Deferenziazione di puntatori a dati aggregati (1/2)

I puntatori alle strutture sono usati spesso e, per brevità, è possibile utilizzare una notazione alternativa all'operatore * per accedere ai campi di una struttura (o di un unione) tramite un puntatore alla struttura. Se **p** è un puntatore ad una struttura (o unione) e **campo** è il nome di un membro di una struttura referenziata da **p**, allora l'espressione

p –> campo

consente di accedere al membro **campo** della struttura referenziata da **p** (l'operatore –> è un segno meno seguito da un maggiore).



Deferenzíazione di puntatori a dati aggregati (2/2)

Esempio:

```
struct  Punto
{
    float x; //coordinata lungo l'asse X
    float y; //coordinata lungo l'asse Y
};

struct  Rettangolo
{
    struct Punto pt1; //angolo inferiore sinistro
    struct Punto pt2; //angolo superiore destro
};

struct Rettangolo r, *rp = r;
```

Le seguenti espressioni sono equivalenti

```
r.pt1.x
rp->pt1.x
(*rp).pt1.x
```

Puntatori e funzioni (1/8)

- Nel linguaggio C, richiamiamo che gli argomenti nelle chiamate di funzioni **sono passati per valore**: il valore di un argomento formale viene copiato nel corrispondente parametro formale (variabile locale automatica) della funzione.
- Nel linguaggio C, un parametro formale in una definizione di funzione può essere una variabile puntatore. In altri termini, una dichiarazione di un parametro formale può essere una dichiarazione di una variabile puntatore. In questo caso, un corrispondente argomento (parametro attuale) in una chiamata della funzione deve essere un'espressione che restituisce un valore puntatore dello stesso tipo del parametro formale, ad esempio:
  - una variabile dello stesso tipo,
  - o l'applicazione dell'operatore di indirizzamento & ad un'espressione che restituisce un tipo di dato coincidente con il tipo di dato puntato dal parametro formale.

Puntatori e funzioni (2/8)

- In generale, una chiamata consente di modificare il contesto chiamante restituendo un risultato che può essere utilizzato dalla funzione chiamante.
- Quando un parametro formale di una funzione è una variabile puntatore, l'indirizzo di memoria (valore dell'argomento) viene copiato nel parametro formale. In questo modo, tramite il parametro formale, è possibile accedere alla locazione di memoria puntata dall'argomento passato nella chiamata di funzione. In altri termini, è possibile **modificare le variabili nella funzione chiamante** realizzando in modo implicito un **passaggio per riferimento**.
- L'utilizzo di parametri formali di tipo puntatore (implicito passaggio per riferimento) consente ad una funzione di "restituire" più valori alla sua funzione chiamante modificando variabili nella funzione chiamante.

Puntatori e funzioni (3/8)

L'utilizzo di variabili puntatore come parametri formali di funzioni è importante anche per questioni di efficienza:

- Quando un oggetto (ad esempio una struttura) contiene grandi quantità di dati, passare l'oggetto per valore comporta un consumo di tempo e memoria per fare una copia del valore dell'oggetto e trasferirlo al corrispondente parametro formale.
- Passando un puntatore all'oggetto è un'operazione molto più efficiente, dal momento che bisogna copiare solo un indirizzo di memoria.

Puntatori e funzioni (4/8)

Esempio:

```
//Calcola e restituisce il quadrato di un argomento intero
int QuadratoPerValore(int n)
{
    return n*n;
}

//Aggiorna l'intero puntato da pn al
//quadrato del suo valore all'inizio della chiamata
void QuadratoPerRiferimento(int * pn)
{
    *pn = (*pn) * (*pn);
}

int main()
{
   int number =5;
   printf("Quadrato di number = %d\n",QuadratoPerValore(number));
   //Il valore di number dopo la chiamata a QuadratoPerValore è ancora 5
   printf("number = %d\n",number);
   QuadratoPerRiferimento(&number);
    //Il valore di number dopo la chiamata a QuadratoPerRiferimento è 25
   printf("Dopo chiamata a QuadratoPerRiferimento number = %d",number);
}
```

OUTPUT

```
Quadrato di number = 25
number = 5
Dopo chiamata a QuadratoPerRiferimento number = 25
```

Puntatori e funzioni (5/8)

Ricordiamo cha una variabile array monodimensionale è una variabile puntatore il cui valore è l'indirizzo di memoria del primo elemento di un array.

- Una dichiarazione di un parametro formale corrispondente ad un array monodimensionale può essere equivalentemente sostituita con una dichiarazione di una variabile puntatore del tipo coincidente con il tipo degli elementi dell'array (per il compilatore le due dichiarazioni sono equivalenti).

Sintassi parametro formale array monodimensionale:

<tipo elemento> <nome parametro> [ ]

La dichiarazione precedente è equivalente a

<tipo elemento> * <nome parametro>

Puntatori e funzioni (6/8)

Esempio:

```
void ModificaArray1(int b[], int size)
{
    int i=0;
    for(;i<size;i++)
        b[i] *= 2;
}

void ModificaArray2(int * b, int size)
{
    int i=0;
    for(;i<size;i++)
        b[i] *= 2; //Utilizzo dell'operatore
                   // di indicizzazione per una variabile
                   //puntatore
}
void main()
{
   int A[5] = {1,2,3,4,5};

   ModificaArray1(A,5);
   for(int i=0;i<5;i++)
        printf("Elemento indice %d di A = %d\n",i,A[i]);

   ModificaArray2(A,5);
   for(int i=0;i<5;i++)
        printf("Elemento indice %d di A = %d\n",i,A[i]);
}
```

OUTPUT

```
Elemento indice 0 di A = 2
Elemento indice 1 di A = 4
Elemento indice 2 di A = 6
Elemento indice 3 di A = 8
Elemento indice 4 di A = 10
Elemento indice 0 di A = 4
Elemento indice 1 di A = 8
Elemento indice 2 di A = 12
Elemento indice 3 di A = 16
Elemento indice 4 di A = 20
```

Puntatori e funzioni (7/8)

Una funzione può restituire come risultato un valore di tipo puntatore. In questo caso, per puntatori a dati, l'intestazione della funzione ha il seguente formato:

<tipo valore ritorno> * <nome funzione>(<elenco parametri>)

Restituire un puntatore ad una variabile locale automatica od il nome di un array dichiarato all'interno del corpo della funzione (senza allocarlo in modo dinamico) è un errore! Al termine dell'esecuzione della funzione la variabile locale automatica viene distrutta e l'indirizzo del puntatore restituito non è più valido in generale!

Puntatori e funzioni (8/8)

Esempio:

```
int * BadFunction1(int b)
{
    return &b;//errore
}

int * BadFunction2()
{
    int A[5] = {1,2,3,4,5};
    return &A;//errore
}
```

Qualificatore const

Il qualificatore **const** del linguaggio C viene utilizzato nelle dichiarazioni di variabili e nelle dichiarazioni di parametri formali:

- nel caso di dichiarazione di variabili, per specificare che una variabile non possa essere acceduta in scrittura dopo la sua inizializzazione.
- nel caso di dichiarazione di parametri formali, per specificare che il parametro non possa essere acceduto in scrittura nel corpo della funzione.
- Il compilatore segnala un errore in caso contrario!

Qualificatore const per dati costanti (1/2)

Quando si usa il qualificatore **const** in una dichiarazione di variabile, l'inizializzazione nella dichiarazione di variabile è necessaria.

Sintassi dichiarazione variabile non puntatore costante:

```
const <tipo variabile><nome variabile> = <inizializzazione>;
```

Sintassi dichiarazione parametro formale costante non puntatore:

```
const <tipo variabile><nome variabile>
```

Qualificatore const per dati costanti (2/2)

Esempio:

```
//Il parametro b può essere acceduto solo in lettura
void Prova(const int b)
{
    printf("Valore di b =%d",b);
    b=3;//il compilatore segnala un errore
}


void main()
{
    const int c= 5;
    printf("Valore di c =%d",c);
    c= 4;//il compilatore segnala un errore

}
```

Qualificatore const per puntatori

Vi sono tre modi per utilizzare il qualificatore **const** per puntatori a dati.

- Puntatori costanti a dati non costanti
- Puntatori non costanti a dati costanti.
- Puntatori costanti a dati costanti.

Puntatore costante a dati non costanti (1/2)

Un **puntatore costante a dati non costanti** punta sempre alla stessa locazione di memoria e i dati in quella locazione *possono essere modificati* per mezzo del puntatore.

Sintassi dichiarazione variabile puntatore costante a dati non costanti:

```
<tipo variabile> * const <nome puntatore> = <inizializzazione>;
```

Sintassi dichiarazione parametro puntatore costante a dati non costanti:

```
<tipo variabile> * const <nome puntatore>
```

Puntatore costante a dati non costanti (2/2)

Esempio:

```
void main()
{
    int x=0;
    int y=0;

    //ptr è un puntatore costante ad un intero che può
    // essere modificato tramite ptr
    //ptr è inizializzato con l'indirizzo della
    //variabile intera x
    int * cost ptr = &x;

    *ptr = 7; //consentito: aggiorna il valore di x a 7

    //Assegnazione non consentita al valore di ptr:
    // il compilatore genera un error
    ptr = &y;  //errore: ptr è costante

}
```

Puntatore non costante a dati costanti (1/3)

Un **puntatore non costante a dati costanti** *può essere modificato* per puntare ad oggetto qualunque del tipo appropriato, ma i *dati* a cui punta *non possono essere modificati* per mezzo del puntatore.

Sintassi dichiarazione variabile puntatore non costante a dati costanti:

```
const <tipo variabile> * <nome puntatore>;
```

Sintassi dichiarazione parametro puntatore non costante a dati costanti:

```
const  <tipo variabile> * <nome puntatore>
```

Puntatore non costante a dati costanti (2/3)

Esempio:

```
//xPtr non può essere usato per modificare
//la locazione di memoria a cui punta:
//il corpo della funzione non può accedere
// a tale locazione di memoria
void f(const int * xPtr)
{
    *xPtr =100; // errore *xPtr è costante
}

void main()
{
    int y;
    f(&y);//f tenta una modifica non permessa
}
```

Puntatore non costante a dati costanti (3/3)

Parametri formali rappresentati da **puntatori non costanti a dati costanti** consentono di combinare il vantaggio del passaggio per valore (i valori delle variabili del chiamante non possono essere modificati dalla funzione chiamata) con i vantaggi del passaggio per riferimento (efficienza dal momento che viene copiato solo un indirizzo di memoria).

- Ciò può essere utile quando è necessario accedere solo in lettura a dati aggregati (strutture) molto grandi passati dal chiamante alla funzione chiamata. In questo caso, si specifica come parametro formale della funzione chiamata un puntatore a struttura, dichiarato come puntatore non costante a dati costanti. In questo modo, una struttura viene passata per riferimento e ci si assicura allo stesso tempo che la struttura definita nel chiamante non possa essere modificata dal chiamante.

Puntatore costante a dati costanti (1/2)

Un **puntatore costante a dati costanti** punta sempre alla stessa locazione di memoria e i dati in quella locazione di memoria *non possono essere modificati* per mezzo del puntatore. Questo è ad esempio il modo in cui un array deve essere passato ad una funzione che accede solo in lettura agli elementi dell'array.

Sintassi dichiarazione variabile puntatore costante a dati costanti:

const <tipo variabile> * const <nome puntatore> = <inizializzazione>;

Sintassi dichiarazione parametro puntatore costante a dati costanti:

const <tipo variabile> * const <nome puntatore>

Puntatore costante a dati costanti (2/2)

Esempio:

```
void main()
{
    int x=0;
    int y=0;

    //ptr è un puntatore costante ad un intero costante. ptr
    // punta sempre alla stessa locazione; l'intero in quella
    //locazione non può essere modificato tramite ptr
    const int * const ptr = &x;

    *ptr = 7; //errore: *ptr è costante
     ptr = &y;  //errore: ptr è costante

}
```

Aritmetica dei puntatori (1/5)

È possibile eseguire operazioni *'pseudo' aritmetiche* sui puntatori che consentono di aggiornare l'indirizzo di un puntatore che punta ad un elemento di un array all'indirizzo di un altro elemento dell'array che precede o segue il primo lungo l'array. Il valore ottenuto è generalmente valido solo se esso si riferisce ad un elemento dell'array a cui appartiene l'elemento puntato inizialmente dal puntatore.

Aritmetica dei puntatori (2/5)

Sia **<Ptr>** una variabile puntatore o un'espressione che possa essere utilizzata alla stessa stregua di una variabile puntatore (ad esempio l'applicazione dell'operatore di indirizzamento ad una variabile ordinaria).

- **Operazione di incremento:** l'espressione **<Ptr>++** o **++<Ptr>** *incrementa* **<Ptr>** in modo tale da farlo puntare all'oggetto che segue l'oggetto dello stesso tipo correntemente puntato da **<Ptr>**. Il valore di **++<Ptr>** dipende dal tipo dell'oggetto puntato da **<Ptr>** : il valore viene incrementato del numero di celle di memoria utilizzate per memorizzare gli oggetti del tipo puntabili da **<Ptr>** . L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che non è l'ultimo.
- **Operazione di incremento:** l'espressione **<Ptr>--** o **--<Ptr>** *decrementa* **<Ptr>** in modo tale da farlo puntare all'oggetto che precede l'oggetto dello stesso tipo correntemente puntato da **<Ptr>**. L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che non è il primo.

Aritmetica dei puntatori (3/5)

Sia **<Ptr>** una variabile puntatore o un'espressione che possa essere utilizzata alla stessa stregua di un puntatore e **<N>** un'espressione numerica che restituisce un intero non negativo.

- **Aggiungere un intero ad un puntatore:** l'espressione **<Ptr>+ <N>** indica l' **<N>** -esimo oggetto del tipo puntato da **<Ptr>** che segue quello attualmente puntato da **<Ptr>**. Il valore viene incrementato del numero di celle di memoria utilizzare per memorizzare gli oggetti del tipo puntabili da **<Ptr>** moltiplicato per **<N>**. L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che è seguito da almeno altri **<N>** elementi.
- **Sottrarre un intero ad un puntatore:** l'espressione **<Ptr> – <N>** indica l' **<N>** -esimo oggetto del tipo puntato da **<Ptr>** che precede quello attualmente puntato da **<Ptr>**. L'operazione è generalmente valida solo se **<Ptr>** punta ad un elemento di un array che è preceduto da almeno altri **<N>** elementi.

Aritmetica dei puntatori (4/5)

Siano **<Ptr>** e **<Qtr>** espressioni dello stesso tipo utilizzabili alla stessa stregua di variabili puntatore:

- **Sottrarre due puntatori:** l'espressione **<Ptr> – <Qtr>** indica il numero di oggetti compresi tra **<Ptr> – <Qtr>** se **<Ptr>** punta ad un oggetto che segue l'oggetto puntato da **<Qtr>** e l'opposto altrimenti.

Le operazioni pseudo aritmetiche sui puntatori viste precedentemente, ad eccezione della sottrazione tra puntatori, possono essere combinate in modo arbitrario per dar luogo ad espressioni complesse che possono essere utilizzate alla stessa stregua di variabili puntatore.

Aritmetica dei puntatori (5/5)

Esempio.

```
#include <stdio.h>
#define N 10

void Inizializza(int *,int);
void Stampa(int *,int);

int main()
{
    int vett[N];
    Inizializza(vett,N);
    Stampa(vett,N);
    return 0;
}
```

```
void Inizializza(int * v,int n)
{
    int i=0;
    for (;i<n;++i)
    {
        printf("Inserisci l'elemento %d:",i+1);
        scanf("%d",&v[i]);
        /*Istruzione alternativa usando
          l'aritmetica dei puntatori:*/
        /* scanf("%d", v+i); */
    }
}

void Stampa(int * v,int n)
{
    int i=0;
    for (i=0;i<n;++i)
      printf("Elemento in posizione %d :"
          " %d\n",i+1,v[i]);
      /*Istruzione alternativa usando
       l'algebra dei puntatori:*/
      /*printf("Elemento in posizione %d :
          %d\n",i+1,*(v+i)); */
}
```

Confronto fra puntatori

- È possibile confrontare puntatori dello stesso tipo tramite gli operatori relazionali ==, !=,<,<=, ecc. a patto che i due operatori puntino ad elementi dello stesso array. Il confronto può essere utile per determinare ad esempio se uno dei due punta ad un elemento con indice più alto di quello puntato dall'altro.
- Qualsiasi puntatore può essere confrontato con lo 0 (alias la costante NULL) tramite gli operatori == e !=.
- Gli operatori di confronto possono essere applicati anche ad espressioni puntatore utilizzabili alla stessa stregua di variabili puntatore.

Assegnare puntatori ad altri puntatori

- Un puntatore può essere assegnato ad un altro puntatore se entrambi hanno lo stesso tipo. L'eccezione a questa regola è costituita dal **puntatore a void** (cioè **void ***), il quale è un puntatore generico che può rappresentare qualsiasi tipo di puntatore.
- Ad ogni tipo di puntatore è possibile assegnare un puntatore a **void**, e a un puntatore a **void** è possibile assegnare un puntatore di qualsiasi tipo. In entrambi i casi non è necessaria alcuna operazione di cast.
- Un puntatore a **void** <u>non può essere deferenziato</u>: questo perché un puntatore a **void** punta ad una locazione di memoria per un tipo di dati *sconosciuto*. Il compilatore deve conoscere il tipo di dati per determinare il numero di byte che rappresentano l'oggetto puntato.

Relazione tra puntatori e array (1/3)

Array e puntatori sono intimamente correlati in C e spesso possono essere usati in maniera intercambiabili.

- Per definizione, il valore di una variabile array o di un espressione array è l'indirizzo del primo elemento dell'array (elemento di indice 0). In effetti, una variabile array si comporta allo stesso modo di un **puntatore costante**. Essa punta sempre all'inizio dell'array e non può essere utilizzata come operando sinistro di un'operazione di assegnazione.
- I puntatori possono essere usati per fare qualsiasi operazione che implichi l'indicizzazione di un array.

Per illustrare ciò, consideriamo le seguenti definizioni:

```
int b[5]; //Array di 5 interi
int * bPtr; //Puntatore ad un intero
```

Relazione tra puntatori e array (2/3)

```
int b[5]; //Array di 5 interi
int * bPtr; //Puntatore ad un intero
```

Poiché la variabile **b** (nome dell’array) punta al primo elemento dell’array, l’assegnamento:

```
bPtr = &b[0]; //bPtr punta al primo elemento dell'array b
```

può anche essere scritto nella forma.

```
bPtr = b;
```

Inoltre, ad un elemento dell’array, ad esempio **b[3]**,si può fare alternativamente riferimento con l’espressione con puntatori:

```
*(bPtr +3) //notazione puntatore/offset
```

In generale, l’operazione di indicizzazione di array può essere equivalentemente sostituita dalla notazione puntatore/offset

Relazione tra puntatori e array (3/3)

```
int b[5]; //Array di 5 interi
int * bPtr; //Puntatore ad un intero
```

I puntatori possono essere indicizzati come le variabili array. Se **bPtr** ha il valore di **b**, allora l'espressione

```
bPtr[1]
```

si riferisce all'elemento **b[1]** dell'array.

Riassumendo, possiamo dire che un'espressione sotto forma di array e indici è equivalente ad una che utilizza puntatori e offset.

Riepilogo utilizzo puntatori

Le operazioni consentite sui puntatori sono:

- Dereferenziazione e (ulteriore) indirizzamento.
- Assegnazione fra puntatori dello stesso tipo.
- Addizione/sottrazione fra puntatori ed interi.
- Sottrazione e confronto fra due puntatori ad elementi di uno stesso array.
- Assegnazione e confronto con lo zero (costante NULL).
- Indicizzazione di puntatori.

Operatori sizeof (1/2)

- L'operatore unario predefinito **sizeof** viene utilizzato per determinare l'ampiezza in byte della locazione di memoria associata ad un tipo di dato.
- Deve essere utilizzato con la notazione per chiamate di funzioni. Esempio: **sizeof(int)**
- Può essere applicato al nome di una qualsiasi variabile, all'identificatore di un qualsiasi tipo di dati, e anche ad una generica espressione.
- Quando applicato al nome di un array, restituisce l'ampiezza complessiva dell'array (il numero di elementi per l'ampiezza in byte del tipo di dato associato a ciascun elemento).

Operatori sizeof (2/2)

```
void main()
{
    unsigned int x;
    scanf("%u",&x);

    int A[x];
    printf("Ampiezza dell'array = %u",sizeof(A));
}
```

OUTPUT

```
9
Ampiezza dell'array = 36
```

Operatori bit a bit (1/6)

- Gli operatori bit a bit si applicano a operandi di tipo intero con segno o senza segno.
- Sono utilizzati per manipolare i bit nella rappresentazione binaria di operandi interi. Il primo bit (quello più a sinistra) è il bit più significativo.
- Il linguaggio C fornisce 6 operatori bit a bit:
  - AND bit a bit: simbolo **&**
  - OR inclusivo bit a bit: simbolo **|**
  - OR esclusivo bit a bit: simbolo **^**
  - Shift (spostamento) a sinistra: simbolo <<
  - Shift (spostamento) a destra: simbolo >>
  - Complemento a uno: simbolo **~**

Operatori bit a bit (2/6)

AND bit a bit &

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se entrambi i corrispondenti bit dei due operandi sono 1, e a 0 altrimenti.

Esempio:

Il risultato di combinare

65535 = 00000000 00000000 11111111 11111111

1 = 00000000 00000000 00000000 00000001

utilizzando l'operatore & è

1 = 00000000 00000000 00000000 00000001

Operatori bit a bit (3/6)

OR inclusivo bit a bit |

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se almeno uno dei corrispondenti bit nei due operandi è 1, e a 0 altrimenti.

Esempio:

Il risultato di combinare

15 = 00000000 00000000 00000000 00001111

241 = 00000000 00000000 00000000 11110001

utilizzando l'operatore | è

255 = 00000000 00000000 00000000 11111111

Operatori bit a bit (3/6)

OR esclusivo bit a bit ^

- Operatore binario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se i bit corrispondenti nei due operandi sono differenti, e a 0 altrimenti.

Esempio:

Il risultato di combinare

139 = 00000000 00000000 00000000 10001011

199 = 00000000 00000000 00000000 11000111

utilizzando l’operatore ^ è

76 = 00000000 00000000 00000000 01001100

Operatori bit a bit (4/6)

Complemento a 1 ~

- Operatore unario. Ogni bit nella rappresentazione binaria del risultato è posto a 1 se il bit corrispondente dell’operando è 0, e a 0 altrimenti.

Esempio:

Il complemento a 1 di

21845 = 00000000 00000000 01010101 01010101

è

4294945450 = 11111111 11111111 10101010 10101010

Operatori bit a bit (5/6)

Shift a sinistra <<

- Operatore binario. Sposta a sinistra i bit del suo operando sinistro del numero di bit **N** specificato nel suo operando destro.
- Gli ultimi **N** bit sono sostituiti con 0.
- I primi **N** bit dell'operando vanno perduti.
- Il risultato dello shift a sinistra è indefinito se l'operando destro è negativo o se l'operando destro è più grande del numero di bit con cui è memorizzato l'operando sinistro.

Esempio:

Il risultato dello shift a sinistra << di

960 = 00000000 00000000 00000011 11000000

di 8 posizioni bit (e, cioè, 960 << 8) è

245760 = 00000000 00000011 11000000 00000000

Operatori bit a bit (6/6)

Shift a sinistra >>

- Operatore binario. Sposta a destra i bit del suo operando sinistro del numero di bit **N** specificato nel suo operando destro.
- I primi **N** bit sono sostituiti con 0.
- Gli ultimi **N** bit dell'operando vanno perduti.
- Il risultato dello shift a destra è indefinito se l'operando destro è negativo o se l'operando destro è più grande del numero di bit con cui è memorizzato l'operando sinistro.

Esempio:

Il risultato dello shift a destra >> di

960 = 00000000 00000000 00000011 11000000

di 8 posizioni bit (e, cioè, 960 >> 8) è

3 = 00000000 00000000 00000000 00000011

Esempio utilizzo operatori bit a bit

Stampa di un intero nella sua rappresentazione in bit

```
void VisualizzaBits(unsigned int value)
{
    unsigned int numBits = 8 *sizeof(unsigned int);

    //Alla variabile maschera è assegnato il valore il cui unico bit
    // pari a 1 è il primo
    unsigned int maschera = 1<< (numBits-1);

    //effettua un'iterazione sul numero di bit
    for(unsigned int c=1; c<= numBits;c++)
    {
        //L'operatore & è usato con la maschera per nascondere
        // tutti i bit di value ad eccezione del primo
        //Primo bit del valore corrente di value
        unsigned int bit = (value & maschera) ? 1: 0;
        printf("%d", bit);

        //shift di value a sinistra di un bit
        value <<= 1;

        if(c % 8 ==0)
           printf("   ");
    }
}

void main()
{
    printf("65000 = ");
    VisualizzaBits(65000);
}
```

```
65000 = 00000000   00000000   11111101   11101000
```

Operatori di assegnazione bit a bit

Ogni operatore bit a bit binario ha un operatore di assegnazione corrispondente. Questi operatori di assegnazione bit a bit sono usati in modo simile agli operatori di assegnazione aritmetici.

Operatori di assegnazione bit a bit	
&=	Operatore di assegnazione AND bit a bit.
|=	Operatore di assegnazione OR inclusivo bit a bit.
^=	Operatore di assegnazione OR esclusivo bit a bit.
<<=	Operatore di assegnazione di spostamento a sinistra.
>>=	Operatore di assegnazione di spostamento a destra.

Precedenza e associatività degli operatori

Precedenza e associatività dei vari operatori nel linguaggio C: mostrati dall’alto in basso in ordine decrescente di precedenza.

<table>
<tr><th>Operatori</th><th>Associatività</th><th>Tipo</th></tr>
<tr><td>() [] . -> ++ (postfisso) -- (postfisso)</td><td>da sinistra a destra</td><td>con precedenza più alta</td></tr>
<tr><td>+ - ++ -- ! & * ~ sizeof (tipo)</td><td>da destra a sinistra</td><td>unario</td></tr>
<tr><td>* / %</td><td>da sinistra a destra</td><td>moltiplicativo</td></tr>
<tr><td>+ -</td><td>da sinistra a destra</td><td>additivo</td></tr>
<tr><td><< >></td><td>da sinistra a destra</td><td>di spostamento</td></tr>
<tr><td>< <= > >=</td><td>da sinistra a destra</td><td>relazionale</td></tr>
<tr><td>== !=</td><td>da sinistra a destra</td><td>di uguaglianza</td></tr>
<tr><td>&</td><td>da sinistra a destra</td><td>AND bit a bit</td></tr>
<tr><td>^</td><td>da sinistra a destra</td><td>XOR bit a bit</td></tr>
<tr><td>|</td><td>da sinistra a destra</td><td>OR bit a bit</td></tr>
<tr><td>&&</td><td>da sinistra a destra</td><td>AND logico</td></tr>
<tr><td>||</td><td>da sinistra a destra</td><td>OR logico</td></tr>
<tr><td>?:</td><td>da destra a sinistra</td><td>condizionale</td></tr>
<tr><td>= += -= *= /= &= |= ^= <<= >>= %=</td><td>da destra a sinistra</td><td>di assegnazione</td></tr>
<tr><td>,</td><td>da sinistra a destra</td><td>virgola</td></tr>
</table>