Università degli Studi di Cassino

Corso di Fondamenti di Informatica

Allocazione dinamica di memoria

Anno Accademico 2011/2012

Francesco Tortorella

Allocazione dinamica di memoria

- Finora abbiamo considerato variabili le cui dimensioni erano note a *compile time*. Ci riferiamo in particolare ad array le cui cardinalità dovevano essere note all'atto della definizione.
- Spesso però l'effettiva dimensione non è nota se non a *run time*. La soluzione adottata consisteva nel definire un array di ampia dimensione, sicuramente adeguata per le esigenze a tempo di esecuzione.

Allocazione dinamica di memoria

- Il problema è che una gran parte di spazio rimaneva inutilizzata.
- Una soluzione molto più efficiente sarebbe quella di allocare la variabile solo quando ne conosciamo la dimensione effettiva, a tempo di esecuzione.
- Questa possibilità è fornita dall'allocazione dinamica di memoria.

L’operatore new

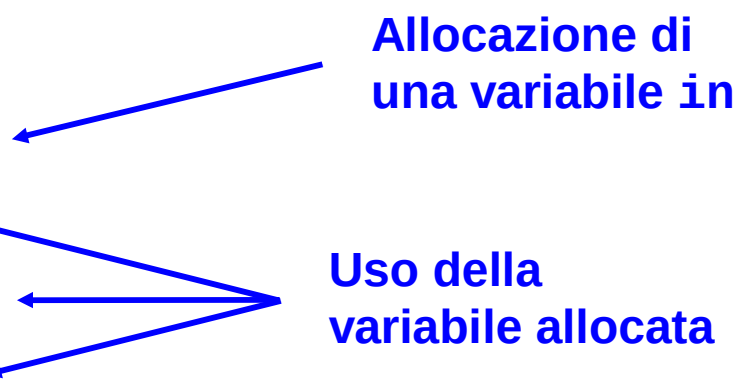
- Per allocare dinamicamente una variabile si usa l’operatore **new**.
- È necessario precisare il tipo della variabile da allocare.
- L’operatore restituisce l’indirizzo della variabile allocata che deve essere assegnato ad un puntatore.
- Diversamente da quanto succedeva con le variabili allocate staticamente, la variabile allocata dinamicamente non ha un identificatore con cui essere riferita.

L’operatore new

- Sintassi dell’operatore new:
  puntatore = **new** *tipo;*
- L’effetto dell’esecuzione è l’allocazione in memoria di un blocco della dimensione adeguata per ospitare una variabile di tipo *tipo*.
- L’indirizzo del blocco viene restituito dall’operatore e assegnato a *puntatore*.
- D’ora in poi si potrà accedere alla variabile allocata tramite *puntatore*.

L'operatore new

```
int main(int argc, char** argv) {
    int *p;

    p = new int;
    cin >> *p;
    *p = *p + 4;
    cout << *p;
}
```



Allocazione dinamica di array

- Tramite **new** è possibile allocare una variabile array.
- Questa volta, oltre al tipo, va specificata la dimensione dell'array:
  puntatore = **new** *tipo*[*size*];
- `size` è la dimensione dell'array e può essere specificata da una variabile (o un'espressione).

Allocazione dinamica di array

```
int main(int argc, char** argv) {
    int *p;
    int n;

    cout << "Dimensione array: ";
    cin >> n;

    p = new int[n];

    leggi_array(p,n);
    stampa_array(p,n);

    return (EXIT_SUCCESS);
}
```

Allocazione di un array di `int` di dimensione `n` letta da input

Uso dell'array allocato

Controllo dell'allocazione

- Nel caso la memoria richiesta non possa essere allocata, **new** restituisce **NULL**.
- È quindi necessario fare una verifica sul puntatore prima di usarlo per evitare di indirizzare un puntatore **NULL**.

```
int main(int argc, char** argv) {
    int *p;

    p = new int;
    if (p==NULL){
   cout << "Memoria non disponibile" << endl;
   return -1;
    }
    cin >> *p;
    *p = *p + 4;
    cout << *p;
    return(EXIT_SUCCESS);
}
```

Deallocazione

- L’allocazione dinamica di memoria opera su uno spazio di memoria limitato e quindi non è possibile procedere con l’allocazione in maniera indefinita.
- Le variabili utilizzate e di cui non si richiede ulteriore impiego vanno quindi deallocate per lasciare spazio alle allocazioni successive.

Deallocazione

- Per questo scopo esiste l'operatore **`delete`** che, applicato ad un puntatore, dealloca (cioè rende disponibile per richieste successive di allocazione) il blocco di memoria il cui indirizzo è presente nel puntatore.
- Sintassi per variabili semplici:
  `delete` *`puntatore`*`;`
- Sintassi per variabili array:
  `delete` `[]` *`puntatore`*`;`

Deallocazione dell'array

```
int main(int argc, char** argv) {
    int *p;
    int n;

    cout << "Dimensione array: ";
    cin >> n;

    p = new int[n];

    leggi_array(p,n);
    stampa_array(p,n);

    delete [] p;

    return (EXIT_SUCCESS);
}
```

Allocazione di un array di `int` di dimensione `n` letta da input

Uso dell'array allocato

Deallocazione dell'array

Possibili problemi

- Esistono diverse situazioni che causano un uso inconsistente della memoria allocata dinamicamente.
- In particolare consideriamo:
  - *Dangling pointer*
  - *Memory leak*

Dangling pointer

- L’operatore delete non cancella il puntatore cui viene applicato, nè modifica il valore del puntatore.
- Questo continua a puntare ad un blocco di memoria che non è più disponibile e che può essere concesso ad un altro processo.
- Dereferenziare il puntatore porta quindi a riferirsi a variabili inesistenti.

Dangling pointer

```
int main(int argc, char** argv) {
    int *p;

    p = new int;
    if (p==NULL){
   cout << “Memoria non disponibile” << endl;
   return -1;
    }
    cin >> *p;

    delete p;
```

Dangling pointer

```
    *p = *p + 4;
    cout << *p;
    return(EXIT_SUCCESS);
}
```

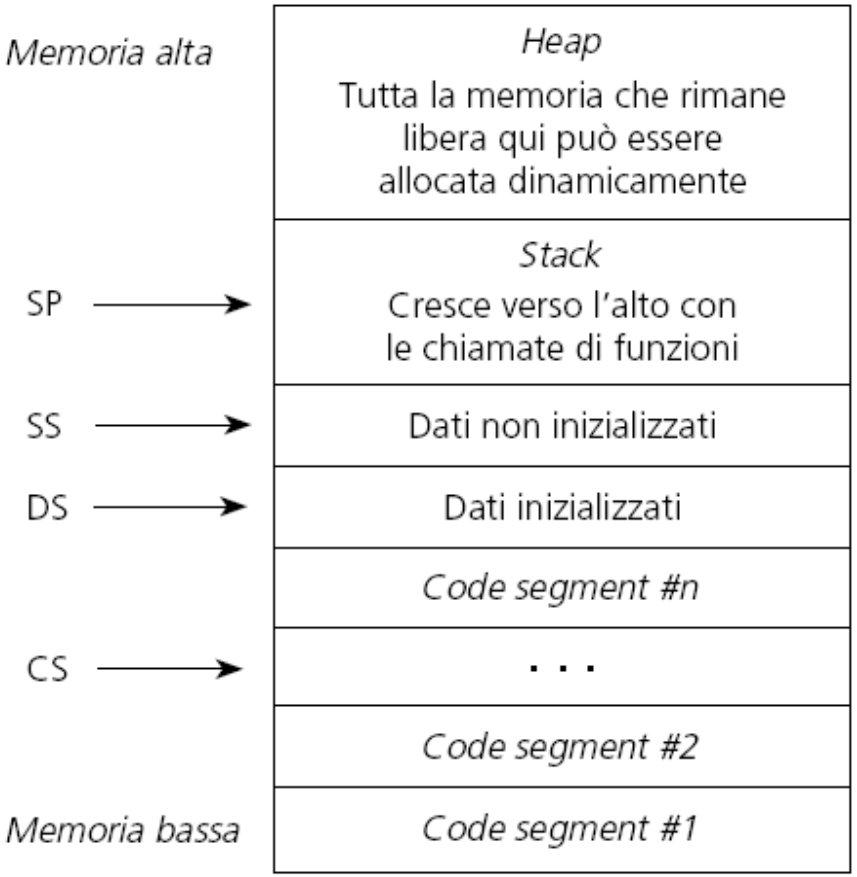
Memory leak

- A valle di un'allocazione il puntatore potrebbe essere erroneamente modificato, perdendo così ogni possibilità di accedere alla memoria allocata.
- In questo caso, oltre a perdere il contenuto eventualmente presente in memoria, questa non potrà essere deallocata.

Memory leak

```
int main(int argc, char** argv) {
    int *p;
    int n, t=0;

    cout << "Dimensione array: ";
    cin >> n;

    p = new int[n];

    leggi_array(p,n);
    p = &t;
          ←———————————————  Array ora irrangiugibile

    return (EXIT_SUCCESS);
}
```

Mappa della memoria per un processo



Garbage collection

- Alcuni linguaggi (p.es. Java e Python) mettono a disposizione la **garbage collection** (letteralmente *raccolta dei rifiuti*, a volte abbreviato con **GC**).
- La garbage collection è una modalità automatica di gestione della memoria, mediante la quale il sistema operativo, o il compilatore e un modulo di run-time, liberano le porzioni di memoria che non sono più *referenziate*, cioè allocate da un processo attivo, e le libererà automaticamente .