

# Gestione della memoria

(Classi di memorizzazione,  
Record di attivazione,  
Spazio di indirizzamento)

# Classi di memorizzazione

- Stiamo per vedere la classificazione di un oggetto in funzione del suo tempo di vita
- In funzione del proprio tempo di vita un oggetto appartiene ad una delle seguenti classi di memorizzazione:
  - 1) Automatica
  - 2) Statica
  - 3) Dinamica

# Oggetti automatici

- Oggetti definiti nei blocchi e parametri formali
- Creati al raggiungimento della loro definizione durante l'esecuzione del programma, distrutti al termine dell'esecuzione del blocco in cui sono definiti
- Se non inizializzati hanno valori **casuali**
  - Dimenticare di inizializzarli è una classica fonte di errori

# Oggetti statici

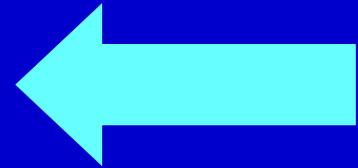
- Oggetti globali (definiti al di fuori di ogni funzione) o definiti nelle funzioni utilizzando la parola chiave **static** (di quest'ultima non vedremo i dettagli)
- Creati all'inizio dell'esecuzione del programma, distrutti al termine dell'esecuzione del programma
- Se non inizializzati hanno valore **zero**

# Oggetti dinamici

- Allocati nella memoria libera
- Esistono dal momento dell'allocazione fino alla deallocazione (o al più fino alla fine dell'esecuzione programma)
- Se non inizializzati hanno valori **casuali**
  - Come per gli oggetti automatici, dimenticare di inizializzarli è una classica fonte di errori

# I tre elementi fondamentali per comprendere le funzioni

- **Definizione/Dichiarazione della funzione**
- **Uso della funzione (chiamata)**
- **Esecuzione della funzione (record di attivazione)**



# Istruzioni e memoria

- Così come gli oggetti, anche le istruzioni (in linguaggio macchina) stanno in memoria
- Sono eseguite nell'ordine in cui compaiono in memoria
  - con l'eccezione delle istruzioni di salto che possono far continuare l'esecuzione da una istruzione diversa da quella che le segue
- Definiamo indirizzo di una funzione l'indirizzo in cui è memorizzata la prima istruzione della funzione

# Esecuzione funzioni

- Prima dell'inizio dell'esecuzione di una funzione, viene creato in memoria il corrispondente record di attivazione
  - Struttura dati contenente fondamentalmente gli oggetti locali alla funzione (variabili/costanti con nome locali nonché parametri formali)
  - Tipicamente memorizzata in una zona contigua di memoria
- I record di attivazione sono a loro volta memorizzati in una zona della memoria del processo chiamata **stack (pila)**

# Record di attivazione

Un record di attivazione rappresenta l'*ambiente della funzione*, e contiene:

Parametro formale 1
Parametro formale 2
...
Parametro formale M
Variabile/costante locale 1
Variabile/costante locale 2
...
Variabile/costante locale N
Collegamento dinamico
Indirizzo di ritorno

Parametri formali, inizializzati con i parametri attuali

Variabili (costanti) locali

Indirizzo del record di attivazione del chiamante, per ripristinarne l'ambiente alla fine dell'esecuzione della funzione

Indirizzo dell'istruzione del chiamante a cui tornare alla fine dell'esecuzione

# Record di attivazione (*cont.*)

- Quando la funzione termina, il controllo torna al chiamante, che deve:
  - riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione
  - trovare tutti suoi dati inalterati
- E' questo il motivo per cui, quando il chiamante chiama la funzione, prima di tutto si inseriscono nel record di attivazione della funzione:
  - **indirizzo di ritorno**, ossia l'indirizzo della prossima istruzione del chiamante che andrà eseguita quando la funzione terminerà
  - **collegamento dinamico**, ossia un collegamento al record di attivazione del chiamante, in modo da poter ripristinare l'ambiente del chiamante quando la funzione terminerà
- La sequenza dei collegamenti dinamici costituisce la cosiddetta ***catena dinamica***, che rappresenta *la storia delle attivazioni* (“chi ha chiamato chi”)

# Record di attivazione (*cont.*)

- La dimensione del record di attivazione:
  - varia da una funzione all'altra
  - ma, per una data funzione, è fissa e calcolabile a priori
- Il record di attivazione:
  - viene *creato dinamicamente* nel momento in cui la funzione viene chiamata
  - rimane sullo **stack** per tutto il tempo in cui la funzione è in esecuzione
  - viene *deallocato alla fine* quando la funzione termina
- **Funzioni che chiamano altre funzioni danno luogo a una sequenza di record di attivazione**
  - allocati secondo l'ordine delle chiamate
  - de-allocati in ordine inverso
  - **i record di attivazione sono *innestati***

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

...

...

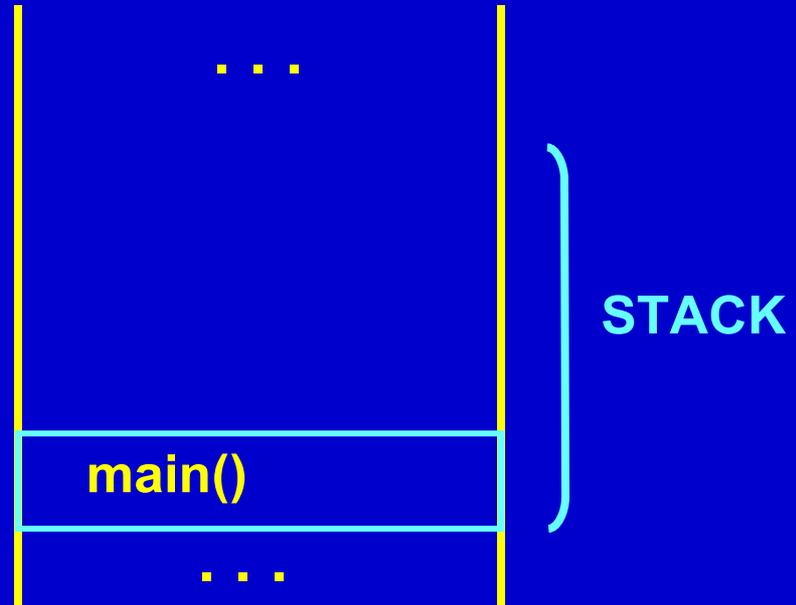
STACK

Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

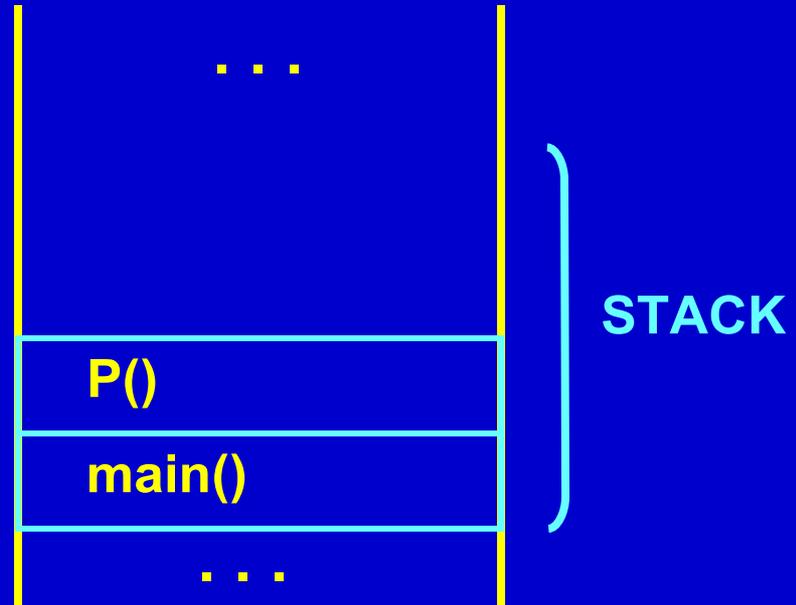


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

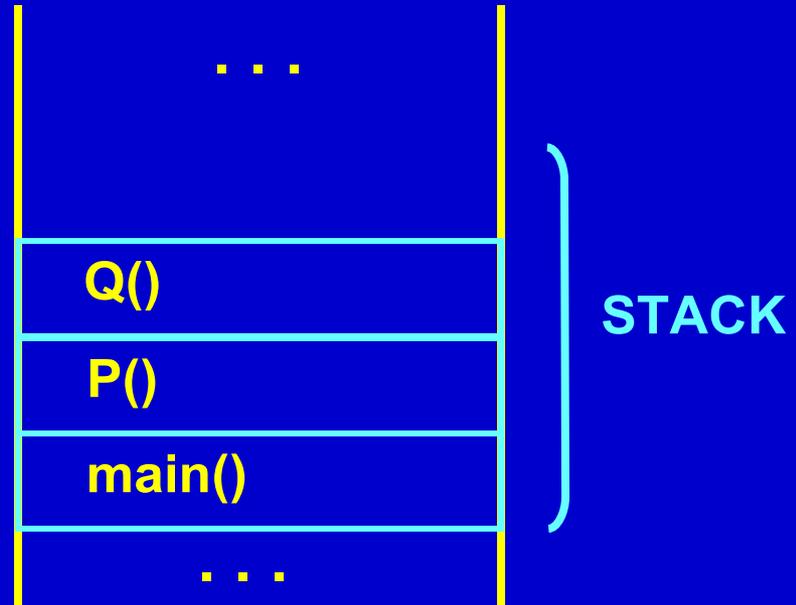


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

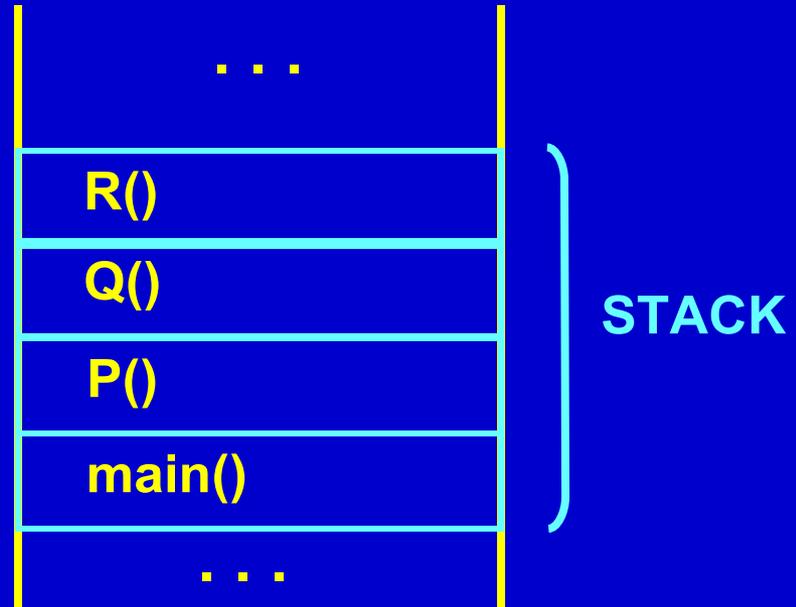


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

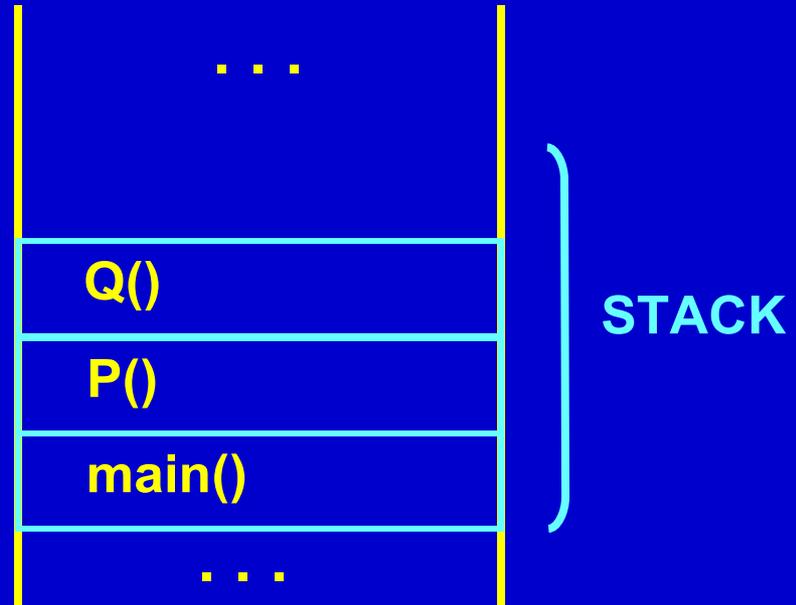


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

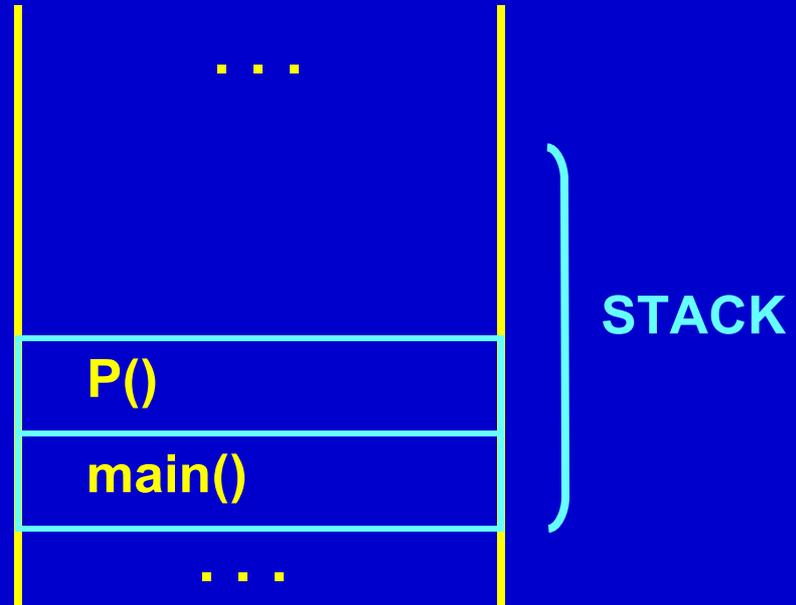


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

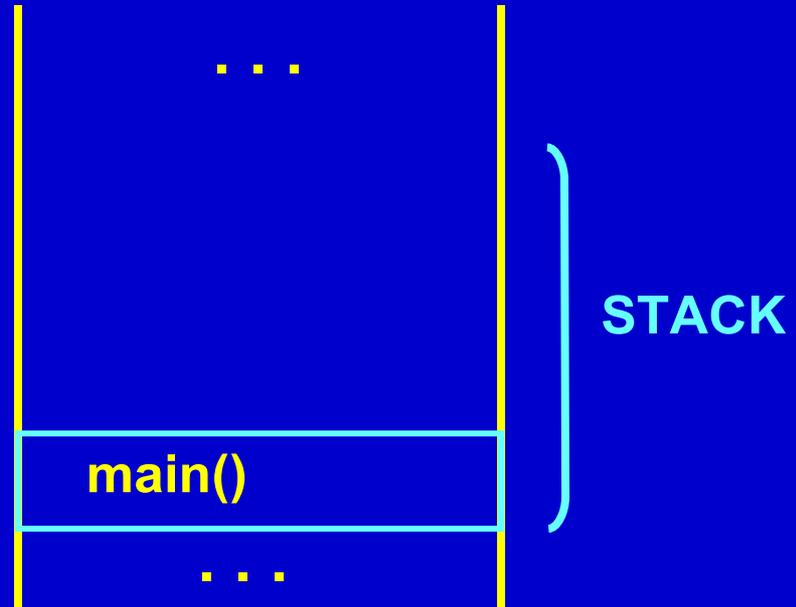


## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```



## Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Esempio

```
int x = 4;
void R(int A) { x = A; }
void Q(int x) { R(x); }
void P() { int a=10; Q(a); }
main() { P(); }
```

...

...

STACK

Sequenza di attivazioni

( Sist. Oper. → ) **main** → **P** → **Q** → **R**

# Uso dello Stack (*Pila*)

- Si ricorda che nel linguaggio C/C++ tutto si basa su funzioni (anche il **main()** è una funzione)
- Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione, che ne chiama un'altra ...), è necessario gestire l'area di memoria che contiene i record di attivazione relative alle varie chiamate di funzioni come una **pila (stack)**:

***Last In, First Out → LIFO***

(L'ultimo record ad entrare è il primo a uscire)

# Esecuzione fattoriale

- Programma che calcola il fattoriale di un valore naturale  $N$

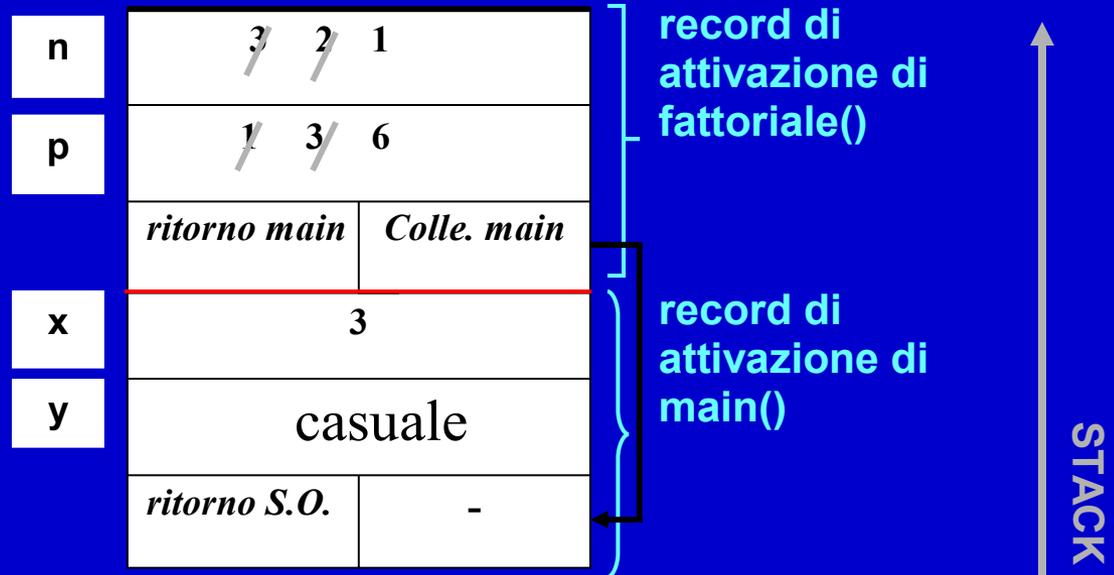
Nota:  $n$  viene inizializzato con il VALORE di  $x$ .  $n$  viene modificato, mentre  $x$  rimane inalterato dopo l'esecuzione della funzione *fattoriale()*

```
int fattoriale(int n)
```

```
{ int p=1;  
  while (n>1)  
    { p=p*n; n-- }  
  return p;  
}
```

```
main()
```

```
{ int x=3, y;  
  y=fattoriale(x);  
  cout<<y<<endl; }
```



La figura illustra la situazione nello stack nel momento di massima espansione, quando la funzione sta per terminare. Quando la funzione *fattoriale()* termina, il suo record viene deallocato e il risultato viene trasferito (tramite un registro della CPU) nella cella di nome  $y$

# Memorizzazione valore di ritorno

- Spesso prima di chiamare una funzione si aggiunge un ulteriore elemento nel record di attivazione del chiamante, destinato a *contenere il valore di ritorno* della funzione che si sta per chiamare
- Tale risultato viene copiato dal chiamante (ovviamente) all'interno di tale elemento prima di distruggere il record della funzione appena terminata
- Altre volte, il risultato viene restituito dalla funzione al chiamante semplicemente *lasciandolo in un registro* della CPU

# Prologo ed epilogo

- Il codice che crea il record di attivazione di una funzione prima di saltare all'indirizzo della (prima istruzione) della funzione si chiama tipicamente **prologo**
- Il codice che immette il valore di ritorno di una funzione nel record di attivazione del chiamante o in un registro di memoria, e poi dealloca il record di attivazione della funzione stessa si chiama tipicamente **epilogo**
- Dato un programma in C/C++, sia il prologo che l'epilogo sono inseriti automaticamente dal compilatore nel file oggetto

# Uno sguardo verso il basso ...

- La macchina virtuale del linguaggio C/C++ fa vedere al programmatore la memoria come un contenitore di celle in cui si creano automaticamente le variabili con classe di memorizzazione statica e automatica (queste ultime si distruggono anche automaticamente), ed in cui si possono allocare dinamicamente oggetti
- Per ottenere queste funzionalità il compilatore inserisce nel programma eseguibile del codice aggiuntivo
  - Prologo ed epilogo per la gestione dei record di attivazione
  - Chiamate a funzioni del sistema operativo per gestire la memoria dinamica
- Non entriamo nei dettagli di questo codice, ma vediamo solo la struttura tipica dello **spazio di indirizzamento** di un processo

# Spazio di indirizzamento

- Lo spazio di indirizzamento di un processo è l'insieme di locazioni di memoria accessibili dal processo



Automaticamente messi a 0, per questioni di sicurezza e ripetibilità

- Direzioni di crescita:
  - lo stack cresce man mano che si annidano chiamate di funzione
  - la memoria dinamica cresce man mano che vengono allocati oggetti

# Valore oggetti statici

- Gli oggetti statici non inizializzati sono inizializzati a zero
  - per questioni di sicurezza, perché altrimenti lanciando un nuovo programma potremmo usarli per leggere il precedente contenuto di locazioni di memoria della macchina
  - per rendere più deterministico e ripetibile il comportamento dei programmi
- Ora abbiamo una spiegazione operativa del fatto che gli oggetti statici sono automaticamente inizializzati a 0

# Valore oggetti automatici e dinamici

- Memoria dinamica e stack possono crescere finché lo spazio libero non si esaurisce
  - sia quando un record di attivazione è rimosso che quando un oggetto dinamico è deallocato, le locazioni di memoria precedentemente occupate **non sono** settate a 0
    - sono invece lasciate inalterate per efficienza
  - ora abbiamo una spiegazione del fatto che gli oggetti dinamici ed automatici hanno valori casuali
    - il valore di un oggetto automatico/dinamico allocato in una certa zona di memoria dipende dai valori precedentemente memorizzati in quella zona di memoria
    - cambia quindi a seconda di quello che è successo prima che l'oggetto in questione venisse allocato