Lezione 20

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 particolare, in funzione del proprio tempo di vita un oggetto appartiene ad una delle seguenti **classi di memorizzazione**:
  - Automatica
  - Statica
  - 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

Uso oggetti dinamici

- Gli oggetti dinamici comportano una evidente maggiore difficoltà di gestione rispetto agli oggetti statici ed automatici
- Comportano anche un maggiore utilizzo della memoria, a causa delle strutture dati nascoste necessarie per gestirne correttamente l'allocazione e la deallocazione
- Quindi
  - Gi oggetti dinamici vanno utilizzati **solo se** sono chiaramente la soluzione migliore o addirittura l'unica soluzione per il problema da risolvere

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

- Quando una funzione **fun** parte
  - l'esecuzione deve saltare all'indirizzo della funzione **fun**
  - devono essere creati tutti gli oggetti locali della funzione **fun** (variabili, costanti con nome e parametri formali)
- Viceversa, quando la funzione termina, il controllo torna al **chiamante**, ossia alla funzione che conteneva la chiamata alla funzione **fun**. Il chiamante deve:
  - riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione
  - trovare tutti i suoi oggetti inalterati
- Il meccanismo che fa sì che accadano correttamente tutte queste cose si basa sull'uso dei cosiddetti <u>record di attivazione</u>

Record di attivazione 1/2

- Quando traduce una chiamata di funzione, il compilatore inserisce nel file oggetto del codice aggiuntivo che
  - Prima dell'inizio dell'esecuzione di una funzione, crea 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

Record di attivazione 2/2

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

Parametri formali, inizializzati con i parametri attuali

Variabili (costanti) locali

Indirizzo del record di attivazione del chiamante, per tornare al suo record di attivazione alla fine dell'esecuzione della funzione

Indirizzo dell'istruzione del chiamante a cui saltare alla fine dell'esecuzione della funzione

Domanda

- Data la modalità in cui gli oggetti locali, variabili o costanti con nome, sono memorizzati in un record di attivazione
- Quali informazioni occorre conoscere per poter accedere ad un oggetto locale della funzione in esecuzione in un dato momento?

Base ed offset 1/2

- L'indirizzo a cui inizia il record di attivazione della funzione
  - Indirizzo base
- Distanza dell'oggetto da tale indirizzo
  - Offset
- Un registro del processore, chiamato (Stack) Base Pointer (BP) è utilizzato tipicamente per memorizzare l'indirizzo base del record di attivazione della funzione correntemente in esecuzione

Base ed offset 2/2

- Il compilatore traduce l'accesso ad un oggetto locale con un accesso alla variabile presente all'indirizzo contenuto nel base pointer più l'offset
  - Ogni oggetto locale è quindi di fatto individuato dal suo offset nel record di attivazione

Prologo

- Il codice aggiuntivo inserito dal compilatore per realizzare la chiamata di una funzione è chiamato **prologo**
- Le azioni principali compiute dal prologo sono
  - Creare il record di attivazione
  - Inizializzare il base pointer con l'indirizzo del record di attivazione
    - Una volta effettuata tale inizializzazione, il codice macchina con cui il compilatore ha tradotto gli accessi agli oggetti locali potrà correttamente accedere a tali oggetti

Dimensione

- La dimensione del record di attivazione:
  - varia da una funzione all'altra
  - ma, per una data funzione, è tipicamente fissa e calcolabile a priori

Creazione/distruzione

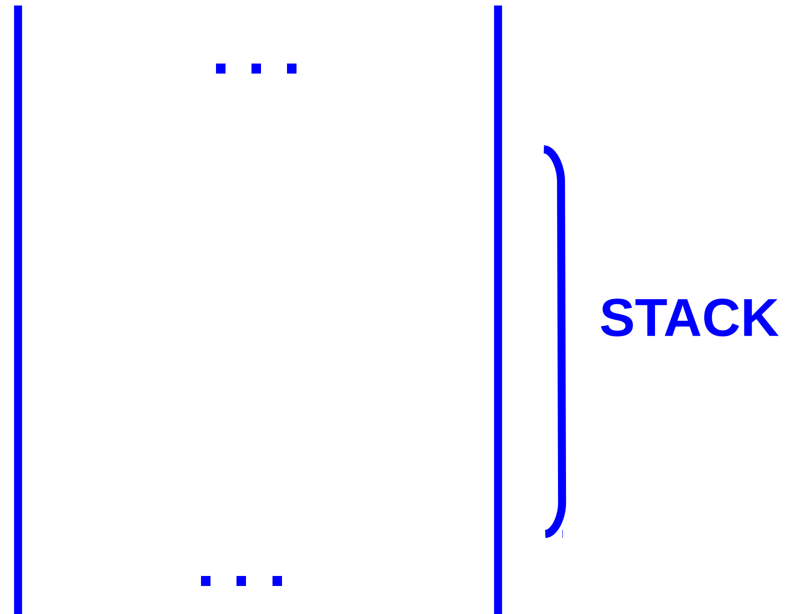
- I record di attivazione sono a loro volta memorizzati in una zona della memoria del processo chiamata **stack (pila)**
- Il record di attivazione di una funzione:
  - viene **creato dinamicamente** nel momento in cui la funzione viene chiamata
  - rimane sullo *stack* per tutto il tempo in cui la funzione è in esecuzione
  - viene deallocato (solo) quando la funzione termina

Sequenza record

- 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();  }
```

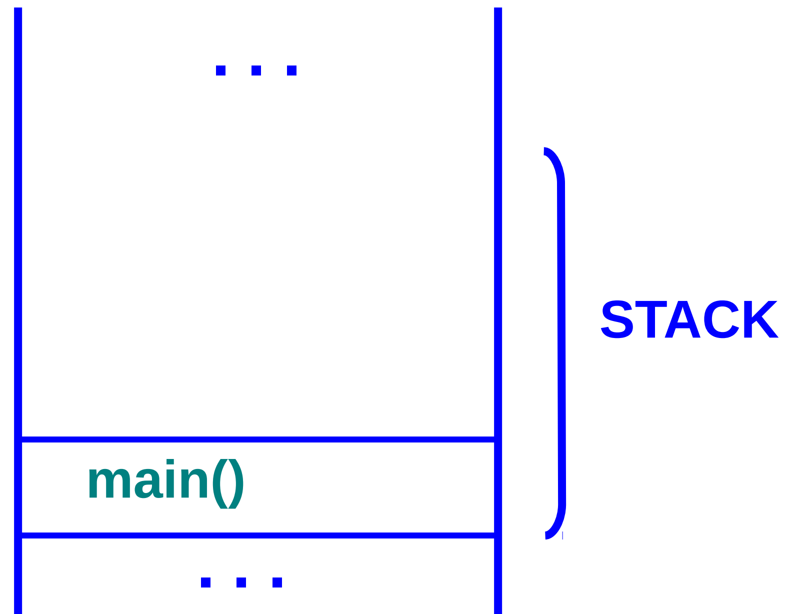


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

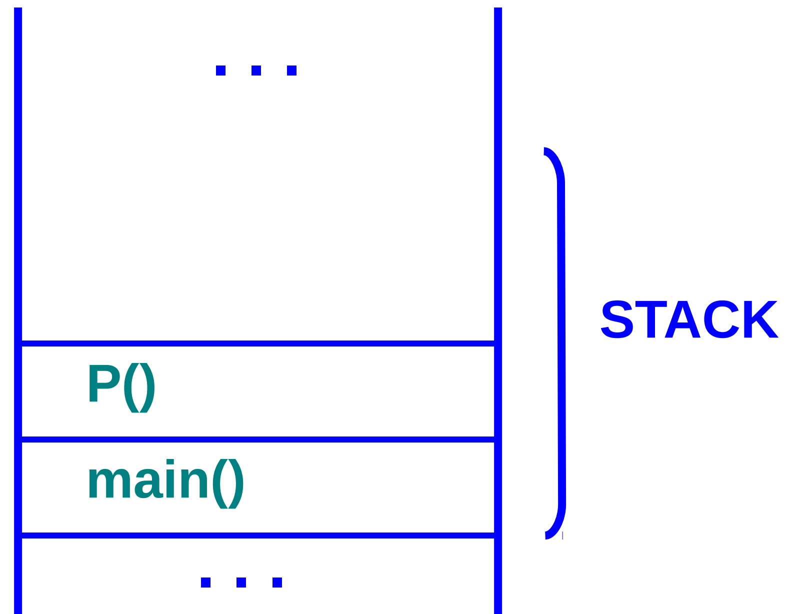


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

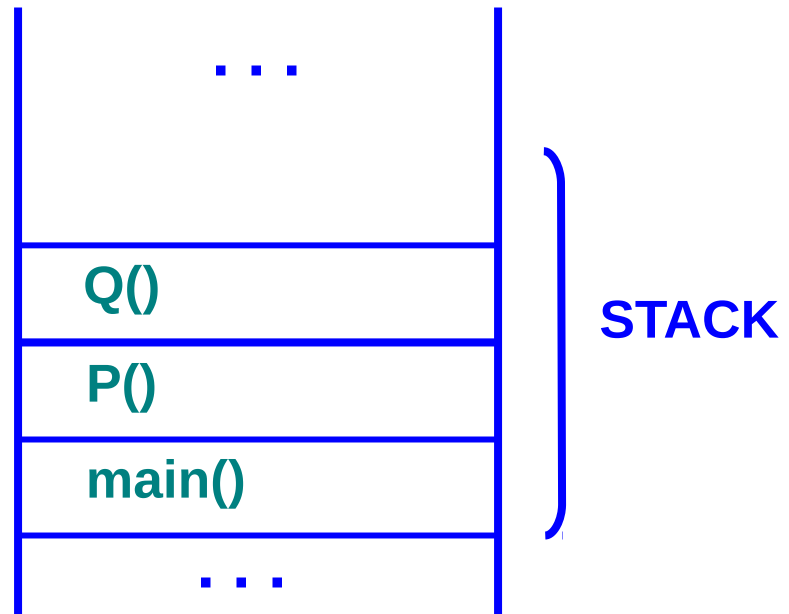


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

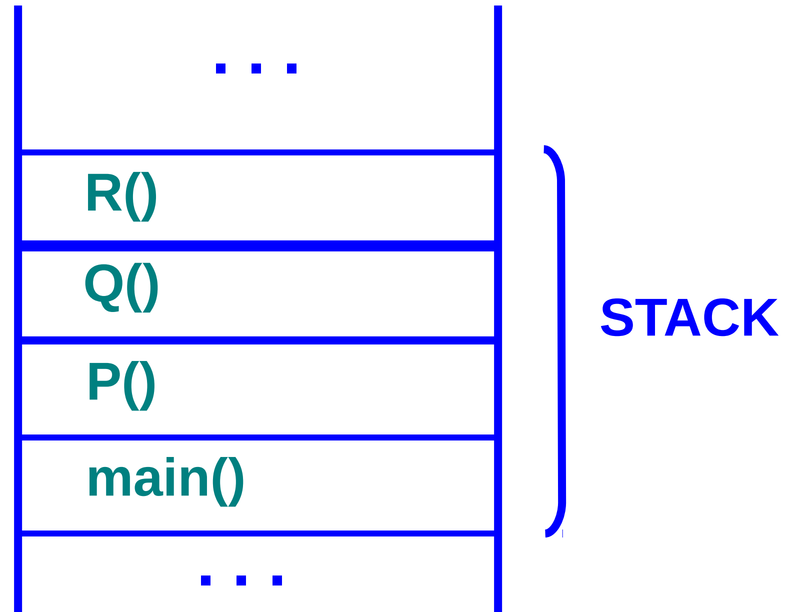


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

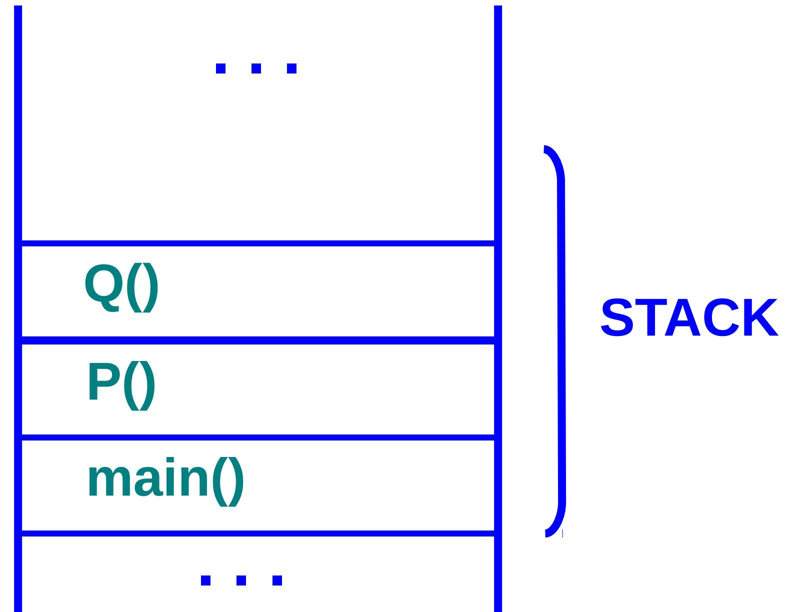


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

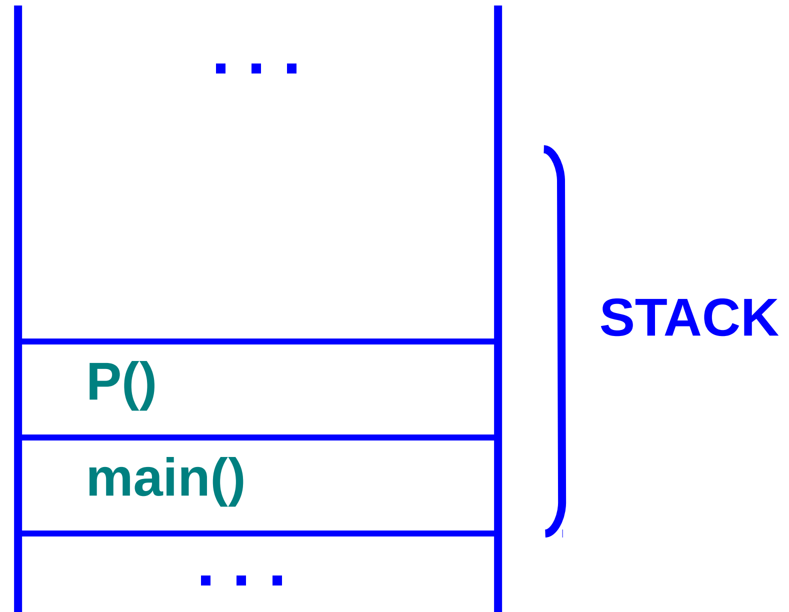


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

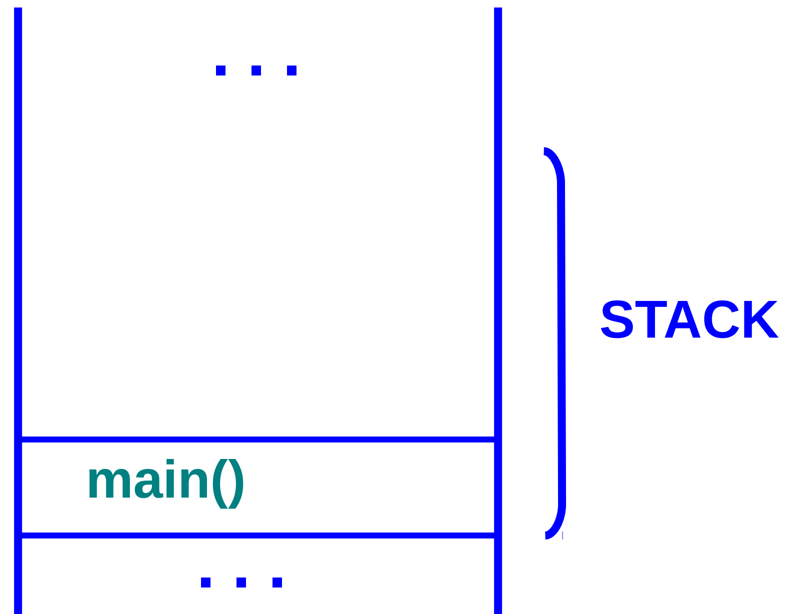


- Sequenza di attivazioni:

  Sistema Operativo → **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();  }
```

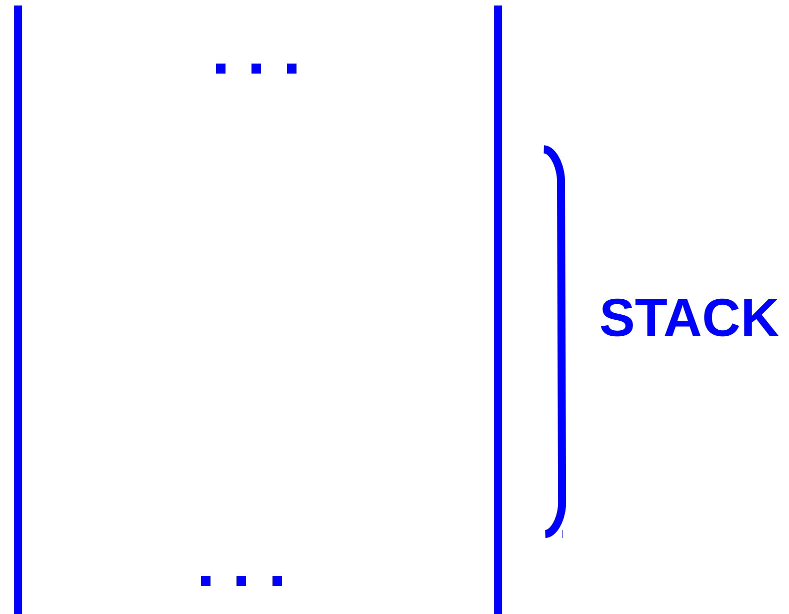


- Sequenza di attivazioni:

  Sistema Operativo → **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:

  Sistema Operativo → **main** → **P** → **Q** → **R**

Stack

- 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 funzione proprio come una pila (stack):

  Last In, First Out → LIFO

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

Domanda

- Abbiamo visto che nel record di attivazione è sempre inserito un campo denominato
  - collegamento dinamico, che contiene l'indirizzo del record di attivazione del chiamante
- Perché è importante tale campo?

Catena dinamica

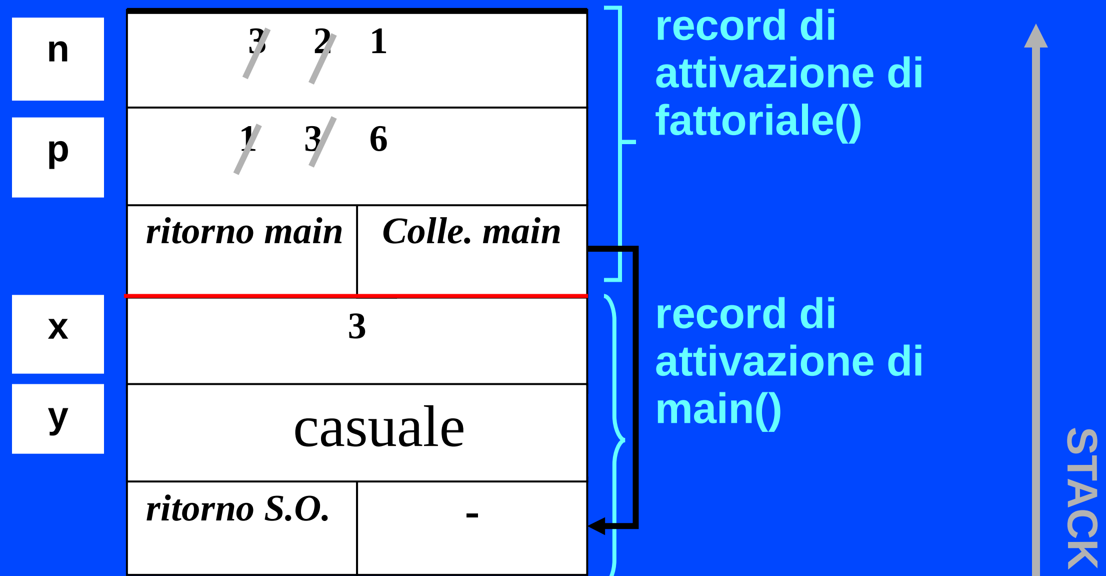
- Perché al ritorno da una funzione permette di sapere l'indirizzo del record di attivazione della funzione che torna in esecuzione
- Il base pointer può così essere aggiornato con tale indirizzo
  - Si può quindi di nuovo accedere, mediante l'offset, agli oggetti locali della funzione che torna in esecuzione
- In sostanza, si ripristina l'ambiente del chiamante quando la funzione chiamata termina
- La sequenza dei collegamenti dinamici costituisce la cosiddetta **catena dinamica**, che rappresenta la storia delle attivazioni ("chi ha chiamato chi")

Esempio: 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 nella cella di nome y col seguente meccanismo

Valore di ritorno

- Prima di chiamare una funzione si aggiunge tipicamente un ulteriore elemento nel record di attivazione del chiamante, destinato a contenere *il valore di ritorno della funzione* che si sta per chiamare
- Il valore di ritorno viene copiato dal chiamante all'interno di tale elemento prima di terminare
- Altre volte, il risultato viene restituito dalla funzione al chiamante semplicemente *lasciandolo in un registro della CPU*

Prologo ed epilogo

- Come abbiamo detto, 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

Macchina virtuale

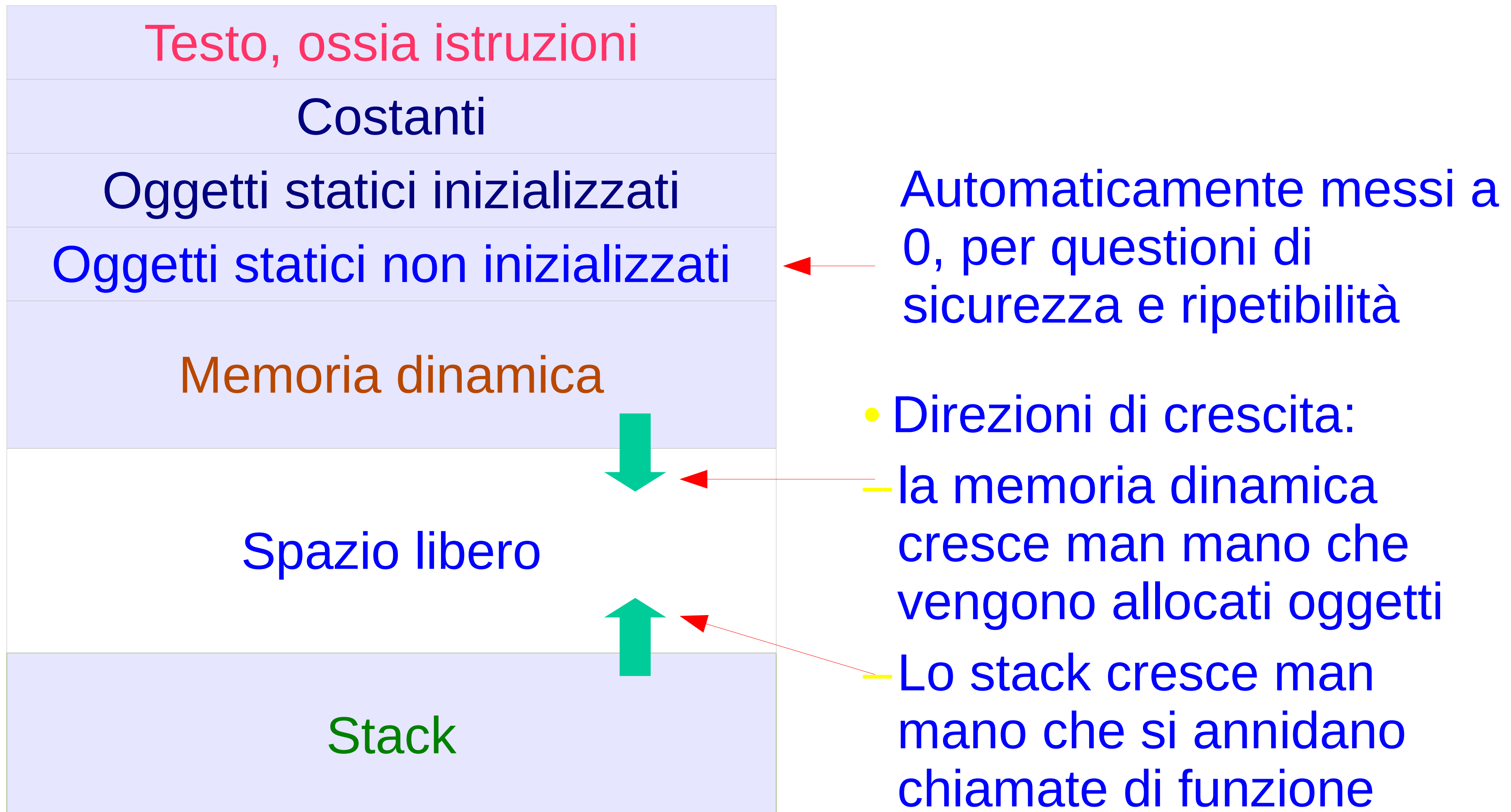
- Come sappiamo, un programmatore C/C++ scrive il proprio programma assumendo concettualmente che venga eseguito da una macchina virtuale che
  - fa vedere la memoria come un contenitore di celle in cui
    - si creano automaticamente le variabili con classe di memorizzazione statica e automatica (queste ultime si distruggono anche automaticamente)
    - si possono allocare dinamicamente oggetti

Uno sguardo verso il basso ...

- Per ottenere queste funzionalità il compilatore inserisce nel programma eseguibile del codice aggiuntivo
  - Come abbiamo visto, prologo ed epilogo per le gestione dei record di attivazione e la creazione/distruzione degli oggetti automatici
  - 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
  - Ricordiamo che si chiama processo un programma in esecuzione

Spazio di indirizzamento

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



Inizializzazione oggetti statici

- Gli oggetti statici non inizializzati sono automaticamente 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 perché gli oggetti statici sono automaticamente inizializzati a 0

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 reinizializzate** a 0
    - sono invece lasciate inalterate per efficienza
  - ora abbiamo una spiegazione del perché gli oggetti dinamici ed automatici hanno valori casuali se non inizializzati
    - 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