Introduzione al linguaggio C++

Programmazione di Sistema

A.A. 2017-18

Argomenti

- Introduzione al linguaggio
- Classi ed oggetti
- Allocazione dinamica
- Passaggio dei parametri

Il linguaggio C++

- Linguaggio vasto ed articolato
  - Compatibile sintatticamente e a livello di moduli oggetto con il C
  - Dotato di estensioni uniche, a volte un po' "esoteriche"
- Progettato per garantire un'elevata espressività
  - Senza penalizzare le prestazioni
  - Le funzionalità che non sono usate da un programmatore non devono pesare sull'esecuzione

Il linguaggio C++

- Offre supporto alla programmazione ad oggetti
  - Incapsulamento
  - Composizione
  - Ereditarietà
  - Polimorfismo
- Ma anche a molti altri stili di programmazione
  - Programmazione generica
  - Programmazione funzionale
  - Programmazione strutturata
- Con una libreria standard relativamente limitata
  - Stringhe, flussi, librerie C, STL
- Dotato di compilatori con caratteristiche molto diverse
  - … in continua evoluzione

C++ “moderno”

- Lo standard è stato aggiornato nel 2011 e 2014
  - Inseriti nuovi meccanismi, derivati dai linguaggi gestiti, per semplificare la programmazione, mantenendo le prestazioni elevate
  - Prossima versione: 2017

Innovazioni nel C++

- Uso di smart pointer
  - Come alternativa ai puntatori nativi
- Set di contenitori generici
  - Dotati di algoritmi standard, al posto di array e strutture dati custom
- Miglioramenti al sistema di gestione delle eccezioni
  - Per aumentare la robustezza del codice
- Uso di funzioni lambda
  - A beneficio della compattezza del codice
- Gestione dell'elaborazione concorrente e della sincronizzazione
  - Attraverso costrutti portabili tra S.O.
  - Capaci di supportare diversi gradi di astrazione

Tipi base

- Interi
  - short / unsigned short
  - int / unsigned int
  - long / unsigned long
- Numeri reali
  - float
  - double
- Caratteri
  - char / unsigned char
  - wchar_t
- Valori logici
  - bool

Tipi derivati

- Enumerazioni
  - enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
  - Definiscono un nuovo tipo a partire da un insieme di valori
- Array
  - int x[1O];
- Puntatori
  - int* ptr;
  - Un puntatore può essere invalido (NULL, nullptr)
  - Gli operatori * e -> dereferenziano un puntatore

Tipi derivati

- Riferimenti
  - int i=O;
    int& r = i; //r è alias di i
  - Si può dichiarare un riferimento solo a partire da una variabile esistente
  - Accedere all'originale o al riferimento produce gli stessi effetti
  - Lo standard non definisce come un riferimento sia implementato
  - Per lo più, il compilatore codifica un riferimento come un puntatore inizializzato che viene automaticamente dereferenziato

Tipi derivati

- Struct
  - struct product {
    int weight;
    double price;
    };
  - Blocchi di informazioni organizzati sequenzialmente
  - Permettono la memorizzazione di dati tra loro collegati
- Union
  - union mytypes_t {
    char c; int i; float f;
    };
  - Modella un'area di memoria che può contenere valori di tipo diverso in momenti diversi

Classi

- Una classe C++ è una struttura dati
  - Può contenere variabili e funzioni legate all'istanza
  - Una funzione membro è simile ad un metodo Java: ha un parametro implicito ("this") che rappresenta l'indirizzo dell'istanza

```
class ResultCode {
  private:
    int code;
  public:
    int get_code() { return code; }
    char* get_description();
};
```


Funzioni membro

- L’implementazione delle funzioni membro può essere
  - Inline, all’interno della definizione della classe stessa
  - Separata, nella stessa unità di compilazione o in un altro modulo collegato

Incapsulamento

- Variabili e funzioni membro possono avere un diverso grado di accessibilità
  - Per default, sono private, accessibili solo alle funzioni membro della classe
- È possibile inserire sezioni public e protected
  - Tutti i membri public sono accessibili a chiunque
  - I membri protected sono accessibili solo alla classe stessa ed alle sue sotto-classi

Definire una classe

- Di solito, si definisce la struttura di una classe in un file «.h»
  - E la si implementa in un file «.cpp»
  - Questo permette di utilizzare la classe senza dover rivelare i dettagli della sua implementazione

```
#ifndef MYCLASS
#define MYCLASS
class MyClass {
  char* ptr;
public:
  int doWork();
};
#endif
```

File.h

```
#include "File.h"

int
MyClass::doWork() {
  //codice
  return 0;
}
```

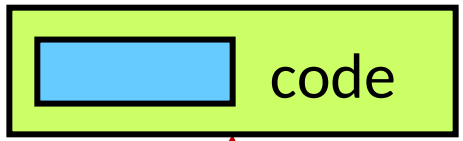
File.cpp

Disposizione in memoria

- Ad ogni oggetto corrisponde un blocco di memoria
  - Contiene gli attributi non statici (pubblici, privati e protetti) ed eventuali altre informazioni utili al compilatore
- Un oggetto può essere allocato in varie aree
  - Memoria globale
  - Stack (local store)
  - Heap (free store)
  - All'interno di un altro oggetto (che a sua volta sta in una delle tre aree precedenti)
- Gli attributi statici vengono sempre allocati nella memoria globale

Disposizione in memoria

```
class ResultCode {
public:
  int getCode();
  char* getDescription();

private:
  int code;

  static char** descs;
};
```



Costruttori

- Un costruttore ha il compito di inizializzare lo stato di un oggetto
  - Non ha tipo di ritorno
  - Ha il nome della classe
  - Sono possibili costruttori differenti se hanno parametri diversi

```
class ResultCode {
public:
  ResultCode(): code(0) {}
  ResultCode(int c) { code=c; }
  int getCode();
  char* getDescription();

private:
 int code;
};
```


Distruttore

- Ha il compito di rilasciare le risorse contenute in un oggetto
  - Ha il nome della classe preceduto da ~ (tilde)
  - Non ha mai parametri
  - È chiamato SOLO dal compilatore
- Se un oggetto non possiede risorse esterne, non occorre dichiararlo
  - Utile quando un oggetto incapsula puntatori a memoria dinamica o altre risorse del S.O. (handle, file_descriptor, …)

Distruttore

```
class ResultCode {
public:
  ResultCode(): code(0) {}
  ResultCode(int c): { code=c; }
  ~ResultCode() { /* azioni */ }
  int getCode();
  char* getDescription();
private:
 int code;
};
```


Azioni del compilatore

- Il compilatore invoca costruttore e distruttore al procedere del ciclo di vita di un oggetto
  - Le variabili globali sono costruite prima dell'avvio del programma e distrutte dopo la sua terminazione
  - Le variabili locali sono costruite all'ingresso del blocco di codice in cui sono definite e distrutte alla sua uscita
  - Le variabili dinamiche sono costruite e distrutte esplicitamente dal programmatore

Allocazione dinamica

- Un oggetto può essere allocato sullo heap
  - Utilizzando un puntatore e l'operatore new
  - Prima o poi, dovrà essere rilasciato tramite l'operatore delete
- L'operatore new
  - Acquisisce dall'allocatore della libreria di esecuzione un blocco di memoria di dimensioni opportune
  - Inizializza tale blocco invocando l'opportuno costruttore

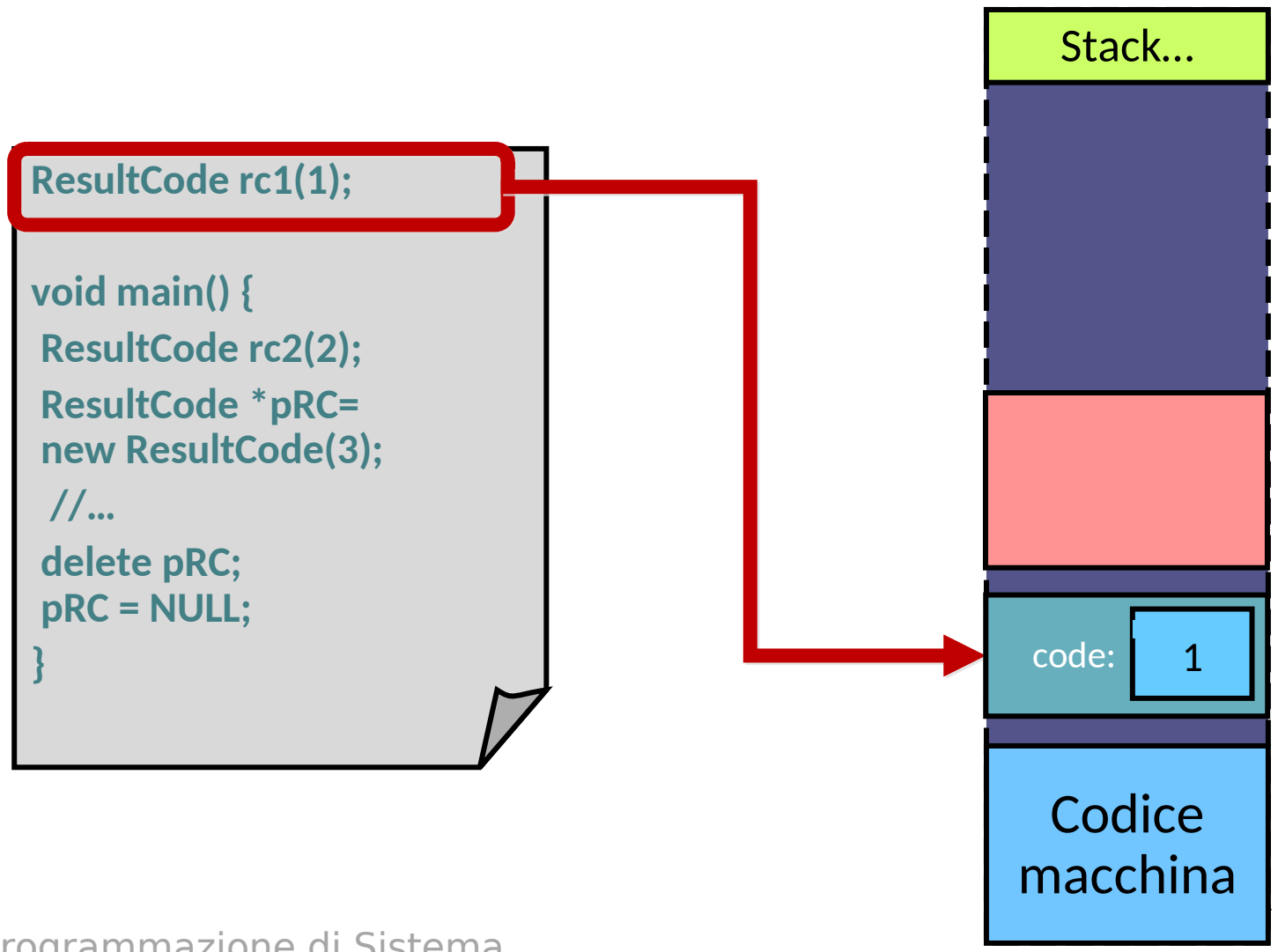
Allocazione dinamica

- L'operatore delete esegue i compiti duali, in ordine inverso
  - Invoca il distruttore dell'oggetto per rilasciarne eventuali risorse
  - Rilascia la memoria occupata dall'oggetto attraverso
    la libreria di esecuzione

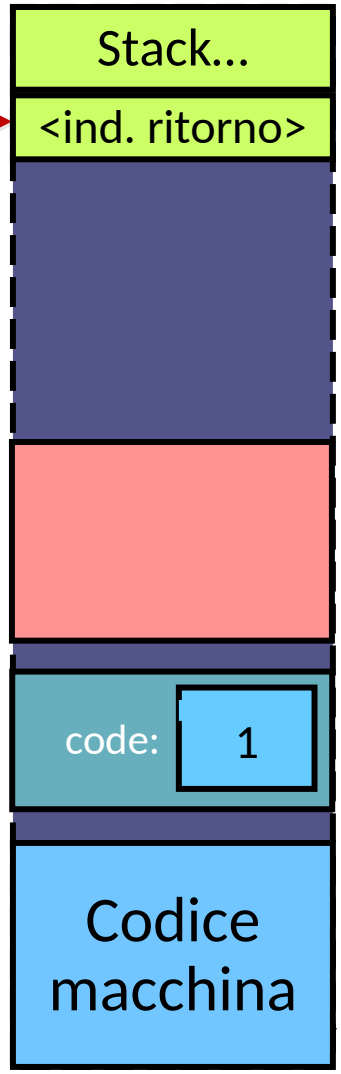
```
ResultCode *pRC;
pRC = new ResultCode( GetLastError() );
//...
printf("%s\n" pRC->getDescription() );
//...
delete pRC;
pRC = NULL; // per evitare dangling ptr
```


Array dinamici

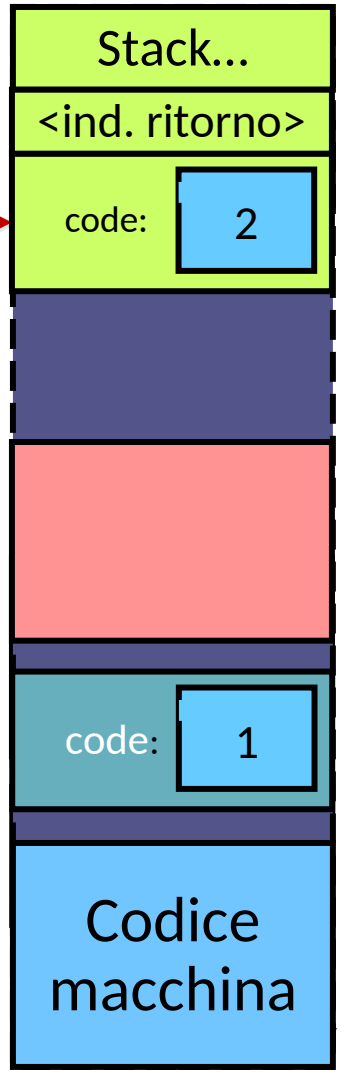
- Si possono allocare dinamicamente array di oggetti
  - Tramite il costrutto ptr = new[count] ClassName
- Occorre rilasciare il blocco tramite l'operatore duale
  - delete[] ptr;



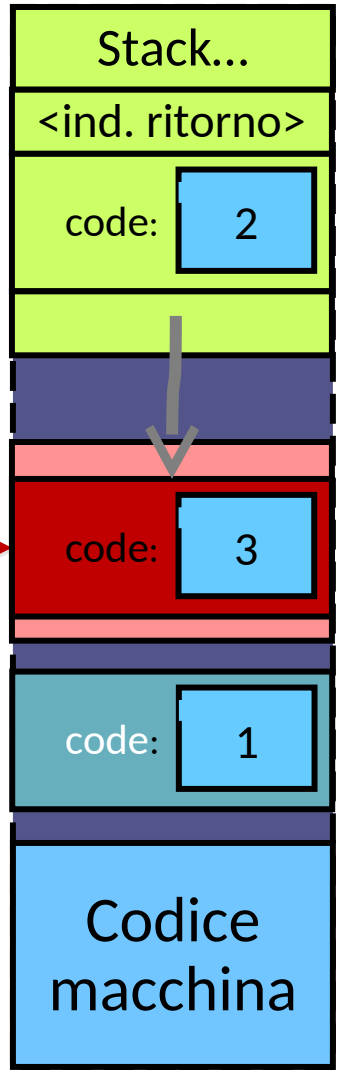
```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
  //...
 delete pRC;
 pRC = NULL;
}
```



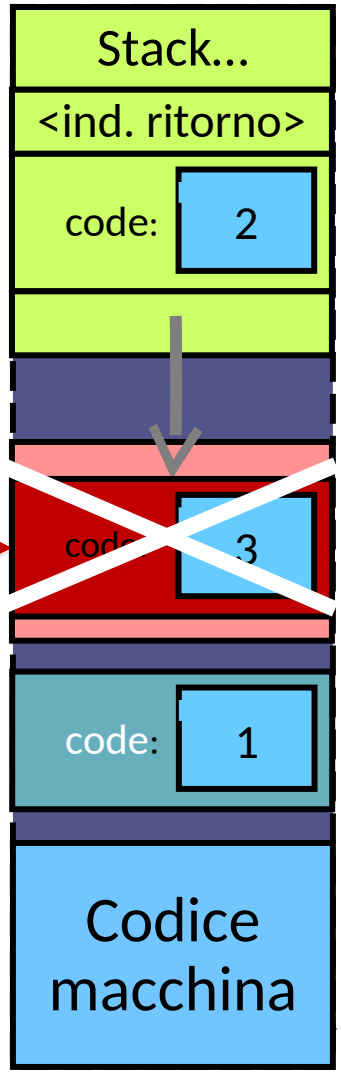
```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
  //...
 delete pRC;
 pRC = NULL;
}
```



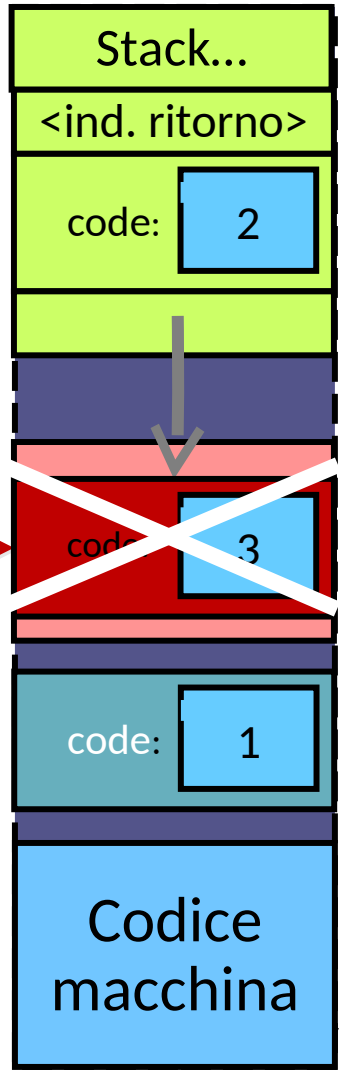
```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```



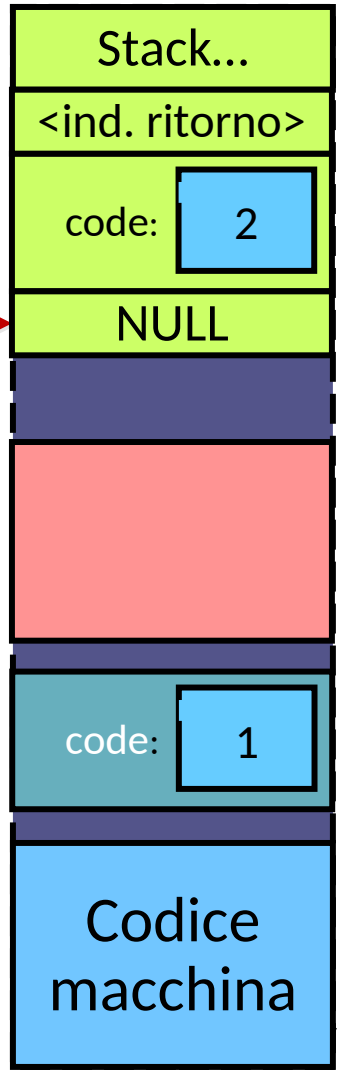
```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
 //
 delete pRC;
 pRC = NULL;
}
```

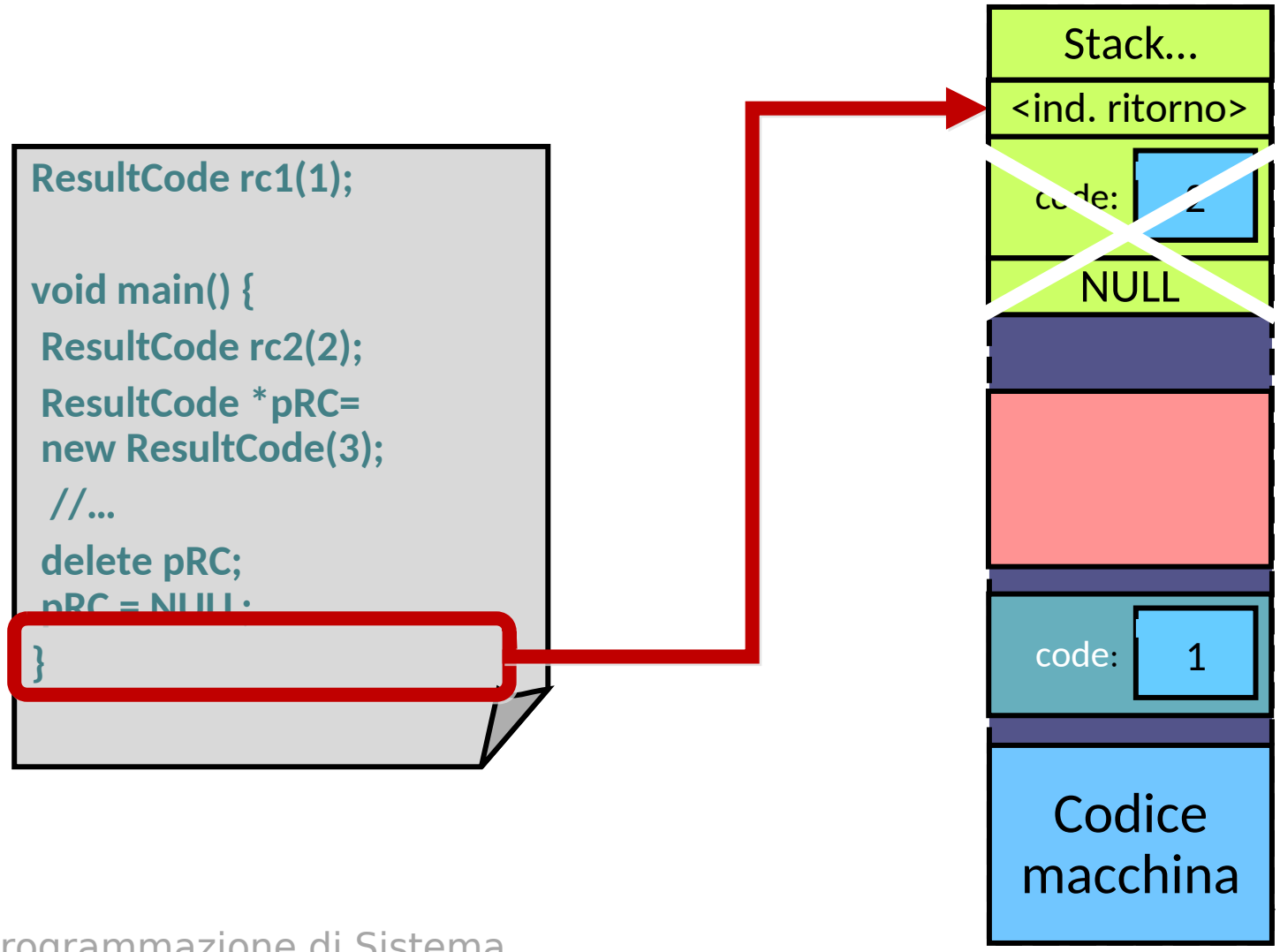


```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
 //
 delete pRC;
 pRC = NULL;
}
```

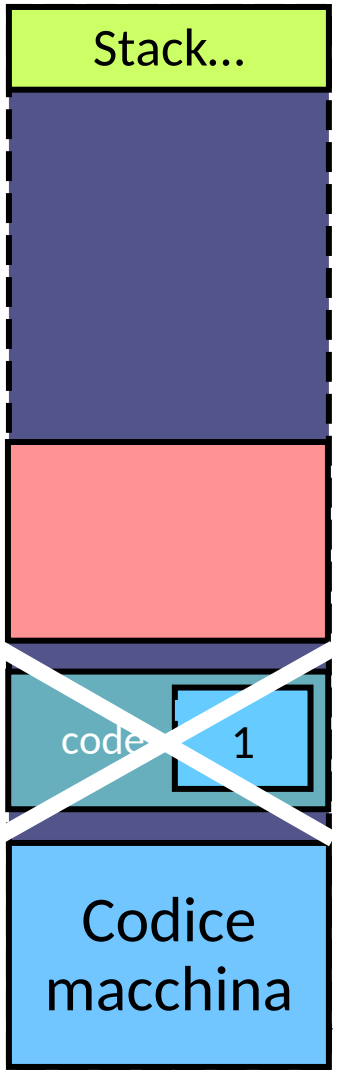


```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```





```
ResultCode rc1(1);

void main() {
 ResultCode rc2(2);
 ResultCode *pRC=
 new ResultCode(3);
  //...
 delete pRC;
 pRC = NULL;
}
```



Passaggio dei parametri

- Funzioni e metodi possono ricevere parametri e generare un valore di ritorno
  - Coerentemente con quanto specificato nella dichiarazione della funzione
  - Il linguaggio definisce più strategie per gestirli

Passaggio per valore

- All'atto dell'invocazione,
  i dati contenuti nei parametri sono duplicati e viene passata la copia alla funzione chiamata
  - Se questa modifica il parametro, l'originale resta immutato

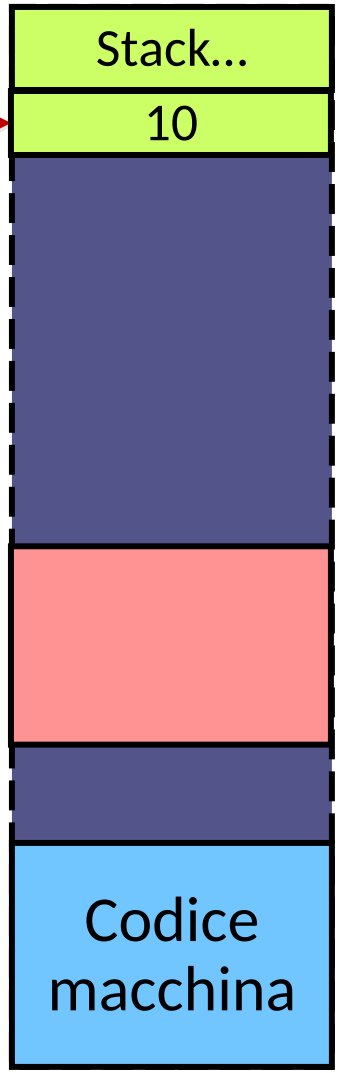
Passaggio per valore

- La funzione agisce su un copia della variabile passata
- Il valore può essere il risultato di un'espressione

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

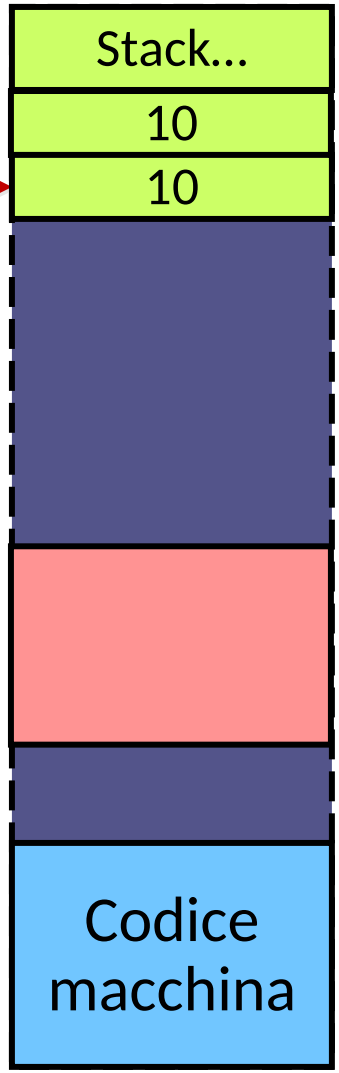

Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



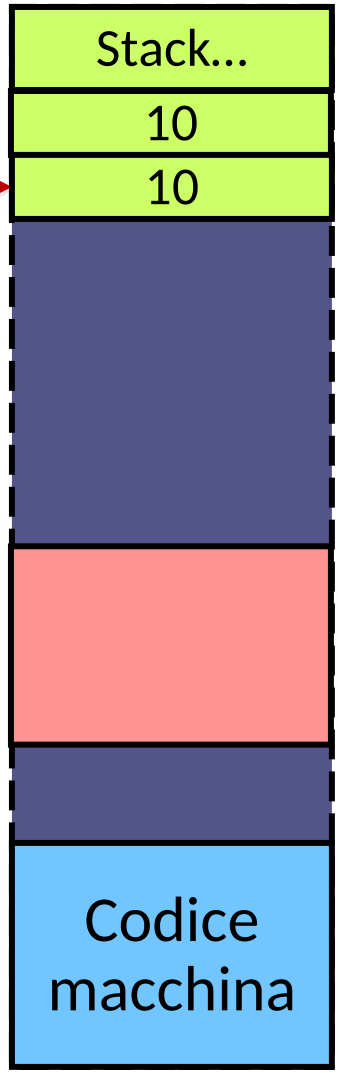
Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



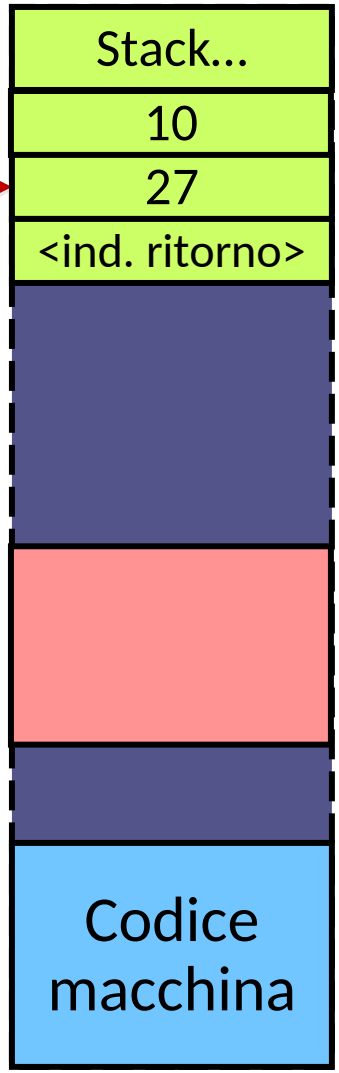
Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



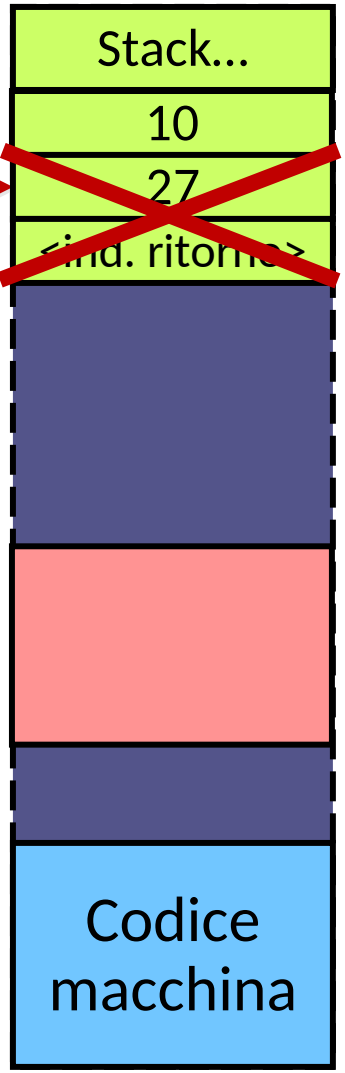
Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



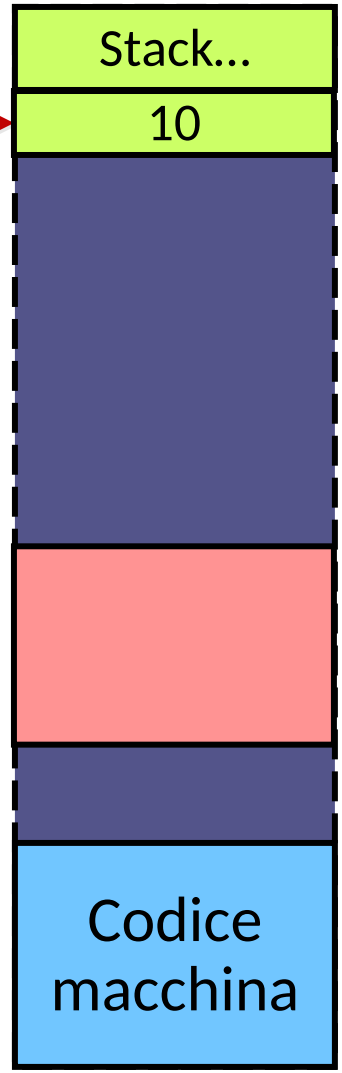
Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



Passaggio per valore

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



Passaggio per indirizzo

- Viene passata una copia dell'indirizzo del dato
  - La funzione chiamata deve esplicitamente dereferenziarlo
  - L'indirizzo può essere NULL
  - Il chiamato può modificare l'originale

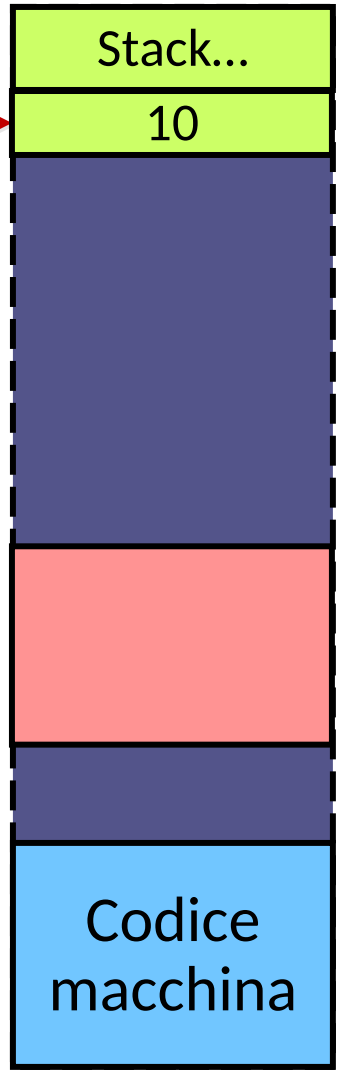
Passaggio per indirizzo

- La funzione non può fare assunzioni sull'indirizzo né sulla durata del ciclo di vita del dato

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```

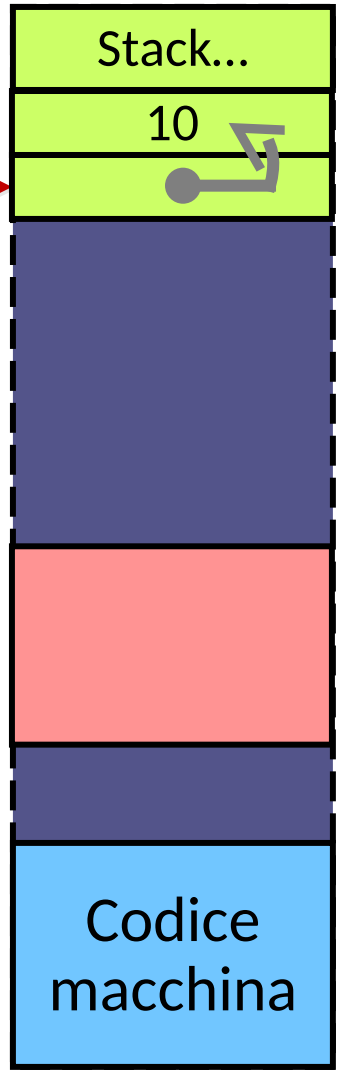

Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



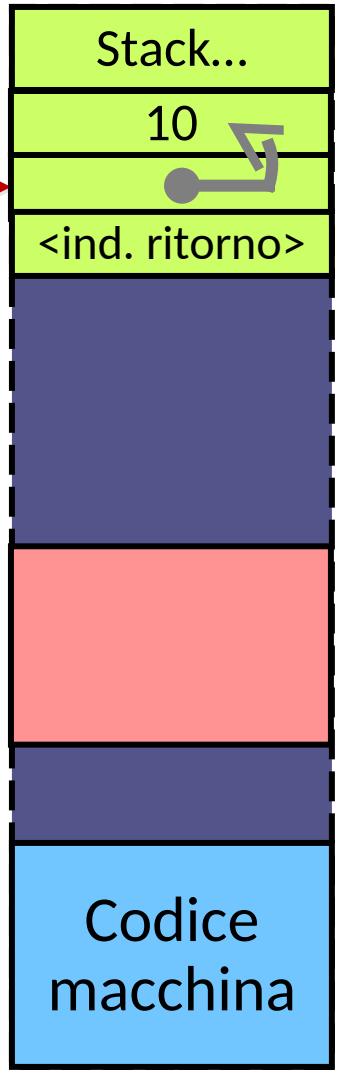
Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



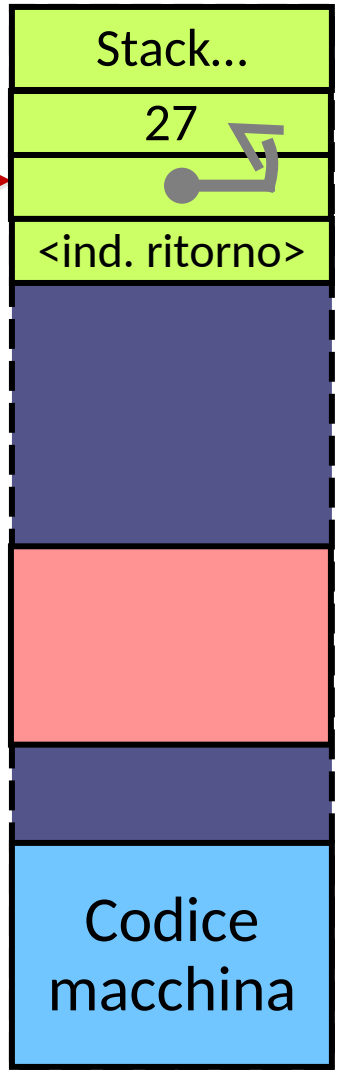
Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



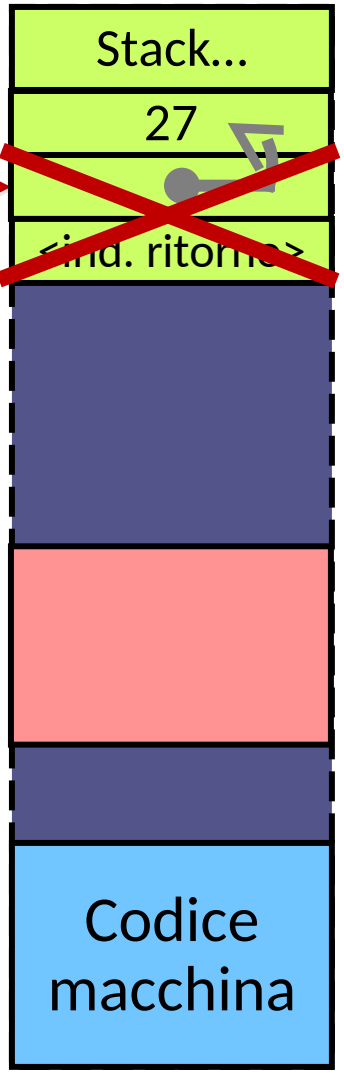
Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
    *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



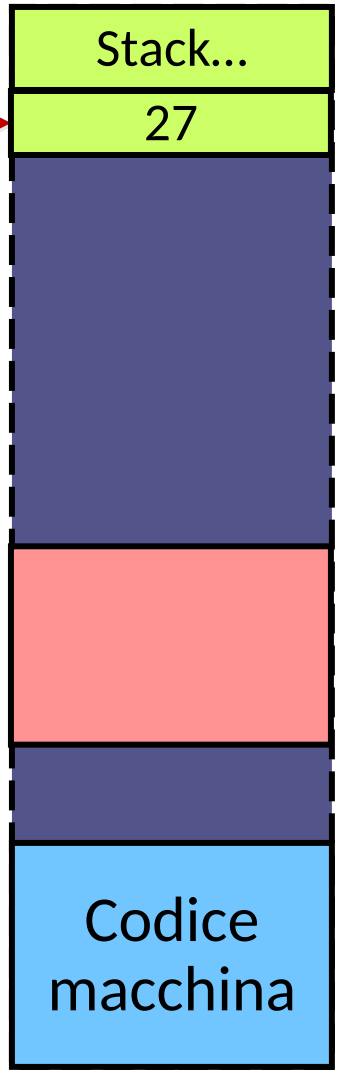
Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Passaggio per indirizzo

```
void f(int* p) {
  if (p!=NULL)
     *p=27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Passaggio per riferimento

- Viene passato un riferimento al parametro originale
  - Sintatticamente, sembra un passaggio per valore
  - Semanticamente, corrisponde ad un passaggio per indirizzo (non è mai NULL)
  - Permette ad una funzione di tornare più valori
  - Aumenta l'efficienza della chiamata
    - Se il parametro formale è preceduto da «const», la funzione ha accesso in sola lettura

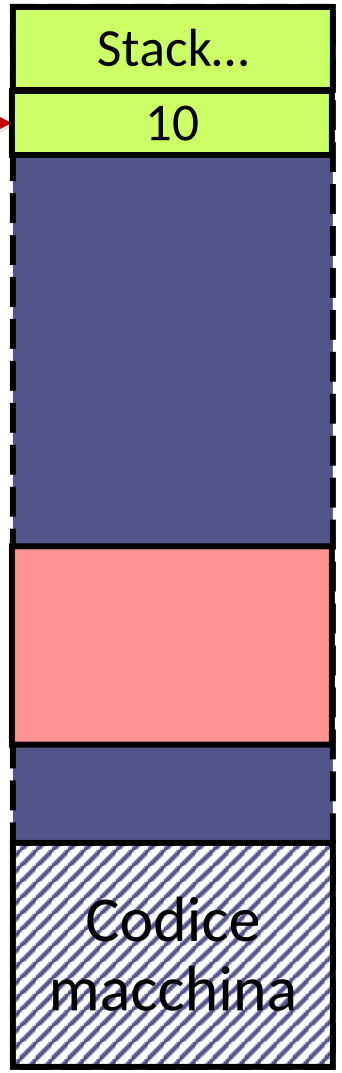
Passaggio per riferimento

- Incontrando l'invocazione della funzione, il compilatore genera un riferimento e lo passa come parametro

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```

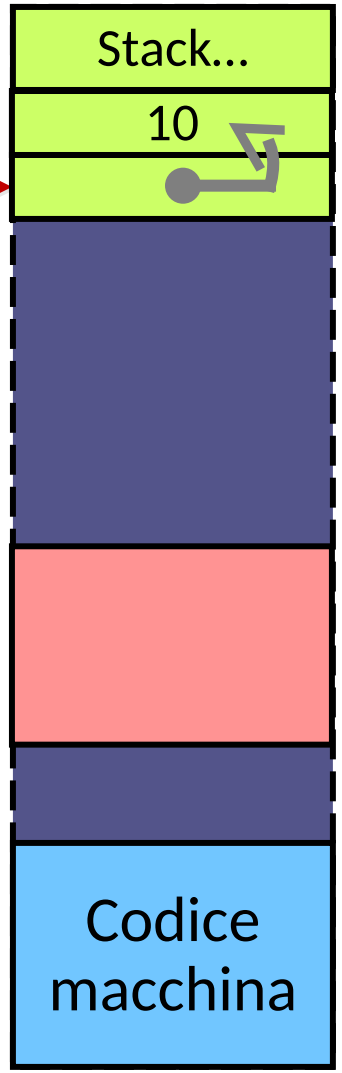
Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



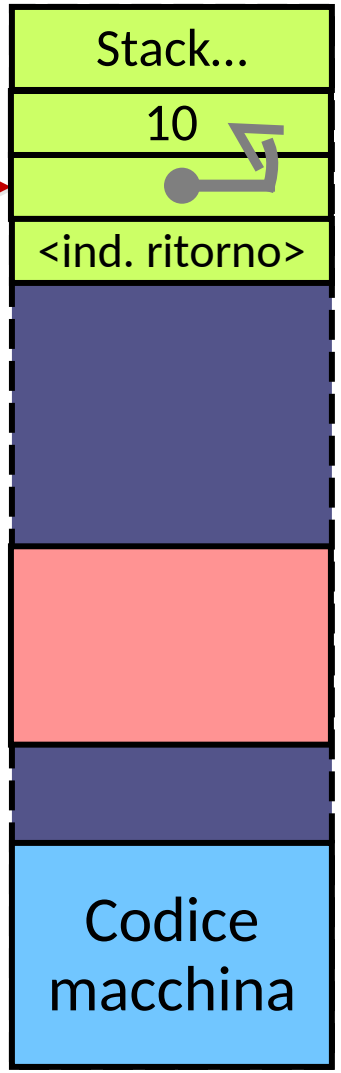
Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



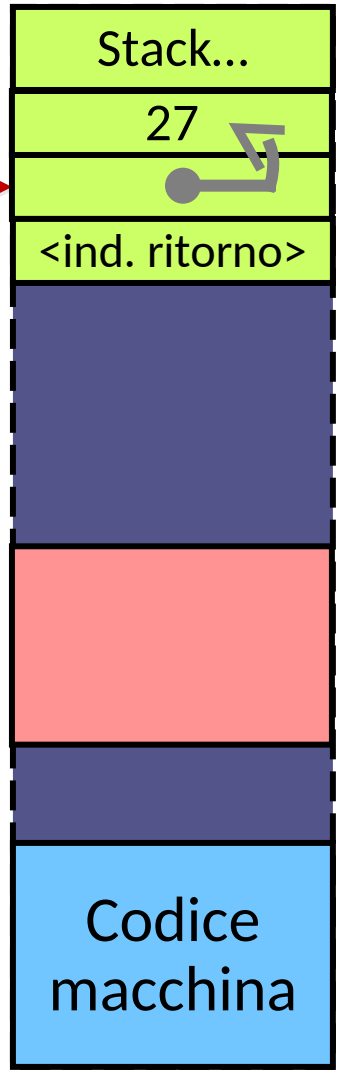
Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



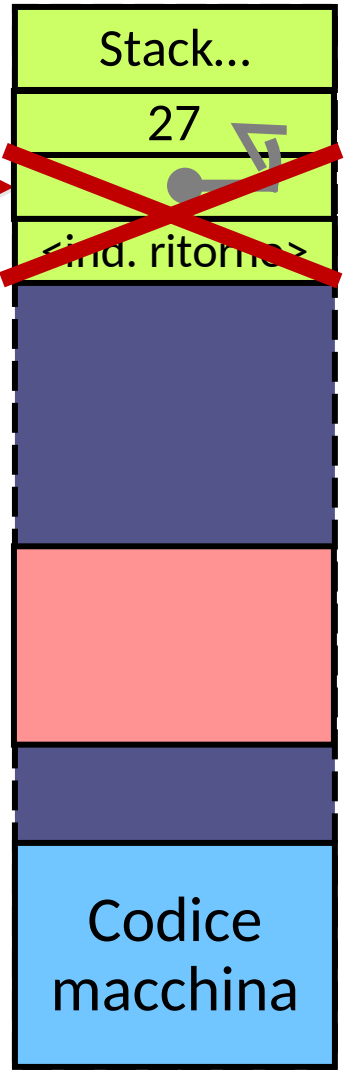
Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



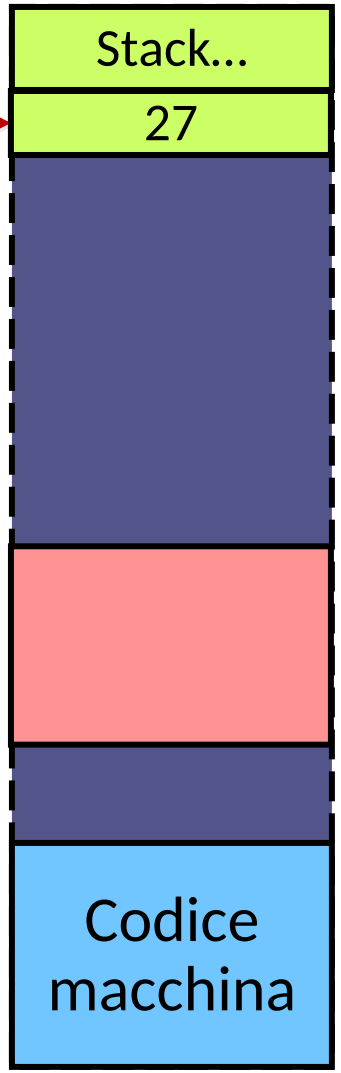
Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



Passaggio per riferimento

```
void f(int& r) {
  r=27;
};

int main() {
  int i=10;
  f(i);
  return i;
};
```



Spunti di riflessione

- Si realizzi la classe Buffer che incapsula un blocco di memoria allocata dinamicamente e la sua dimensione
  - Buffer(int size)
  - ~Buffer()
  - int getSize()
  - bool getData(int pos, int &val)
  - bool setData(int pos, int val)