NUMERO: 2371A ANNO: 2018

A P P U N T I

STUDENTE: Chiapello Nicolò

MATERIA: Programmazione di sistema - (Programmazione) - Prof. Malnati

PROGRAMMAZIONE DI SISTEMA

PROGRAMMAZIONE

Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

Chiapello Nicolò

Contenuto

Informazioni sul corso

Nome:	Programmazione di Sistema
Settore scientifico-disciplinare:	ING-INF/05
Tipo di attività formativa:	caratterizzante
Crediti:	10 CFU
Docente:	Giovanni Malnati
Livello accademico:	laurea magistrale in ingegneria informatica
Periodo:	primo anno, secondo semestre
Anno accademico:	2017 / 2018
Autore:	Chiapello Nicolò
Versione:	1.0.1

Informazioni su quest'opera

Questo lavoro vuole essere la trasposizione della spiegazione fornita in aula, da parte del docente. Esso è frutto di una stesura effettuata direttamente in aula e di successivi eventuali aggiustamenti (specialmente durante il periodo di preparazione all'esame). Non si tratta quindi di sbobinature di lezioni, ma di appunti *real-time*, con tutte le imprecisioni che questo può comportare.
Quest'opera non è stata controllata in alcun modo dai professori e quindi potrebbe contenere degli errori. Se ne trovi uno sei invitato a contattare l'autore, affinché esso venga corretto (nicochina.notes@gmail.com).
Questi appunti sono quindi forniti *as is* : costituiscono uno strumento ottimale per l'accompagnamento allo studio, ma non costituiscono una fonte accademica di nozioni.

Legenda

In questa opera è stata adottata la seguente convenzione:

- *corsivo*: concetto/entità precedentemente già definita
- **grassetto**: concetto di rilievo, in evidenza
- sottolineato: mini-titolo a cui si riferisce il testo successivo
- **giallo grassetto**: definizione di un concetto (riportato nell'indice analitico finale)
- azzurro: argomento di cui si occupa un paragrafo

Presentazione corso

Corso di *Programmazione di Sistema* nel secondo semestre del secondo anno di laurea magistrale.

Docenti

Corso tenuto dai docenti:

- Gianpiero Cabodi: *gianpiero.cabodi@polito.it*
- Giovanni Malnati: *giovanni.malnati@polito.it*

Organizzazione insegnamento

Corso diviso in due porzioni da 5 CFU ciascuno. Sono pressoché indipendenti, ma si occupano entrambi di Sistemi Operativi.
Il corso è composto dalle seguenti parti (presentate in blocchi di appunti distinti):

- teoria (5 CFU): tenuto da Cabodi, si occupa delle parti interne del Sistema Operativo (in particolare del *kernel*)
- programmazione (5 CFU): tenuto da Malnati, si occupa della scrittura di un programma che utilizza le *system calls* (in particolare programmazione concorrente)

Argomenti

Il corso verte sui seguenti argomenti:

- Cabodi: come creare il codice di un sistema operativo (NO *system manager*). Segue un "taglio verticale"
  - gestione della memoria centrale
  - file system
  - gestione dei dischi
  - gestione dei dispositivi di I/O
  - sistema operativo didattico OS/161 (*Unix-like*)
    - introduzione, compilazione e debug del kernel
    - gestione thread di kernel
    - gestione memoria
    - processi user e chiamate di sistema (*system calls*)
- Malnati: come interagire con un sistema operativo (linguaggi di script e API software)
  - piattaforma e modello di esecuzione
  - linguaggio C++
  - programmazione concorrente
  - linguaggio C#

INTRODUZIONE

Piattaforme di esecuzione

Ambienti operativi

In questo corso utilizzeremo prevalentemente C/C++ e Java.

Di C++ analizzeremo le estensioni introdotte nel 2011. Si utilizzeranno gli ambienti di sviluppo

- Visual Studio versione 12 o superiore (Windows)
- g++ 4.8.x+ (Linux)

Nel 2011 il C++ è stato profondamente rivisitato nelle sue caratteristiche: sono state aggiunte estensioni per la programmazione concorrente (rende le applicazioni *platform independent*).

Per la parte di Android analizzeremo anche Java, utilizzando i seguenti ambienti di sviluppo:

- AndroidStudio
- IntelliJ CE

Interfacciarsi con il sistema operativo

Fatta eccezione per i sistemi più elementari (come le schede ESP32), l'esecuzione di un'applicazione è mediata dal sistema operativo. Non è quindi possibile interagire direttamente con le periferiche o allocare memorie. Il SO si frappone tra applicativo e hardware, esso conosce le risorse disponibili e le assegna in modo equo a tutte le applicazioni in azione sulla stessa macchina.
Il SO effettua anche i controlli di sicurezza, per evitare interazioni dannose tra applicazioni distinte.

Per sviluppare un'applicazione bisogna quindi conoscere il sistema operativo, a cui essa è destinata, e le relative specifiche. Il sistema operativo definisce due distinti livelli di specifiche:

- specifiche a livello di codice sorgente: API che sono *platform independent*
- specifiche a livello di codice eseguibile: ABI che sono *platform dependent*

API

Le **API (Application Programming Interface)** sono un insieme di funzioni e strutture dati che vengono offerte al programmatore affinché possa accedere alle risorse sottostanti. Invocando le API si ottiene l'accesso ai servizi offerti dal sistema operativo.
Per esempio effettuando una `malloc()`, essa richiede una funzionalità del sistema operativo che attivi al gestore della memoria fisica che assegna degli indirizzi fisici alle pagine logiche e le assegna al processo che ha effettuato la `malloc()` ad alto livello.

Una particolare ABI è supportata da tutti gli strumenti che compongono la toolchain:

- **compilatore**: genera dei moduli oggetto (codice macchina) a partire da un dato codice sorgente
- **linker**: unisce i diversi moduli generati dal compilatore, a seconda delle reciproche chiamate
- **debugger**: consente l'osservazione di un programma in esecuzione
- **profiler**: osserva l'esecuzione *runtime* e aggiunge dei contatori al codice per memorizzare delle statistiche di utilizzo delle risorse
- **inspector**: permette lo *heap walking* per osservare i blocchi di memoria allocati, da chi e la loro storia

Le ABI possono divenire oggetto diretto della programmazione di sistema quando si utilizzano meccanismi quali:

- caricamento dinamico dei moduli
- emulazione delle piattaforma di esecuzione
- …

Accedere alle API

Per scrivere un programma che si interfacci col sistema operativo è necessario conoscere:

- le API
- le strutture dati coinvolte
- le convenzioni generali definite dal SO

Gli sviluppatori di SO offrono delle *helper function*, organizzate in librerie, e i file `.h` ed esse associate. Per esempio, per chiamare il sistema operativo in Windows, è necessario includere nel progetto il file `windows.h`. Esso conterrà altri file `.h` in una cascata parecchio estesa.
Per compilare questo tipo di progetti è richiesto una grossa mole di tempo. Per ovviare a questo problema, la prima compilazione genera dei file **PCH (Pre-Compiled Header)** nei quali il *parsing* delle strutture standar è stato eseguito una volta sola.

Può essere necessario collegare l'eseguibile con librerie specifiche. Questo può essere fatto in modo:

- statico
- dinamico

Convenzioni

Ciascun SO mantiene, al suo interno, delle strutture dati che descrivono lo stato del sistema. Per esempio il numero di processi in esecuzione o il numero di file *handle* aperti.
Tali strutture dati non sono accessibili al programmatore, perché all'interno del kernel. Le API permettono di accedere in modo indiretto a tali strutture attraverso **riferimenti opachi** (non il puntatore diretto alla locazione di memoria, ma un identificatore numerico temporaneamente riferito agli oggetti di interesse). Essi assumono una diversa nominazione a seconda dell'ambiente di sviluppo:

- Windows: **handle**
- Linux: **file descriptor**

2. Gestire gli errori in Linux

In ambiente Linux un fallimento è spesso indicato col valore -1, ma dipende dal contesto. È quindi necessario consultare la documentazione specifica.

Per ragioni storiche l'errore che si è verificato è depositato nella pseudo-variabile globale Error Number (`errno`). Per la programmazione concorrente è necessario che la variabile non sia globale. Si è quindi applicato il seguente stratagemma:

```
#define errno (*__errno_location ())
```

Analogamente a Windows è necessario ispezionare il codice d'errore non appena questo si verifica. Ulteriori chiamate al sistema potrebbero sovrascrivere la variabile.

Uso di stringhe

La codifica e la gestione del testo è un argomento molto complesso.
Per ragioni storiche i caratteri sono codificati nella **tabella ASCII** che contiene solo le 26 lettere del dizionario inglese, 10 cifre e 32 caratteri speciali, per un totale di 127 corrispondenze. Mancano tutti i caratteri accentati e i caratteri di lingue non convenzionali (cirillico, greco, …).
Con l'espansione dell'informatica, fu espansa la tabella, ma fu necessario codificarla mediante convenzioni universalmente riconosciute. Apple e altre grosse aziende collaborarono per creare la **codifica UNICODE** che comprendesse tutti gli alfabeti, simboli matematici/idraulici/elettrici.

Ad oggi UNICODE è arrivata la versione 8 e utilizza almeno $21bit$. Utilizzare però $4byte$ per ogni carattere è estremamente oneroso.
Per codificare un carattere esistono più rappresentazioni possibili:

- **UTF-32**: ogni simbolo occupa $32bit$
- **UTF-16**: un simbolo può occupare una o due parole da $16bit$
- **UTF-8**: un simbolo può occupare da una a quattro parole di $8bit$

Non si ha più una corrispondenza biunivoca tra dimensione in byte e numero di simboli rappresentati. È fondamentale conoscere la codifica utilizzata prima di leggere del testo, dedicando quindi i primi bit per informare sulla codifica scelta.

Le rappresentazioni a lunghezza variabile possono avere ordinamenti differenti:

- **big endian**: inizia dal byte più significativo per finire col meno significativo ($MSB \cdots LSB$)
- **little endian**: inizia dal byte meno significativo per finire col più significativo ($LSB \cdots MSB$)

Mentre la convenzione *littel endian* è utilizzata principalmente dai processori Intel, la convenzione *big endian* fu adottata da Motorola e sopravvive all'interno dei protocolli Internet.

1. Caratteri in Windows

L'ambiente Windows, così come anche Java e C#, ha deciso di adottare un carattere esteso il cui tipo è definito `wchar_t` di lunghezza $16bit$. Potrà quindi essere rappresentato solo un sottoinsieme di caratteri, quelli conformi al **BMP (Basic Multilingual Plan)**.

Modello di esecuzione

Modello di esecuzione di un programma

Ciascun linguaggio di programmazione ad alto livello corrisponde ad uno specifico modello di esecuzione.
Un **modello di esecuzione** è un insieme di comportamenti attuati dall'elaboratore a fronte dei costrutti di alto livello del linguaggio.
Tale modello non corrisponde però al modello del dispositivo reale. Esclusi i sistemi *flat* (dove si ha acesso diretto alle risorse), i sistemi normali introducono un layer che mostra al programma utente tutte le risorse disponibili.

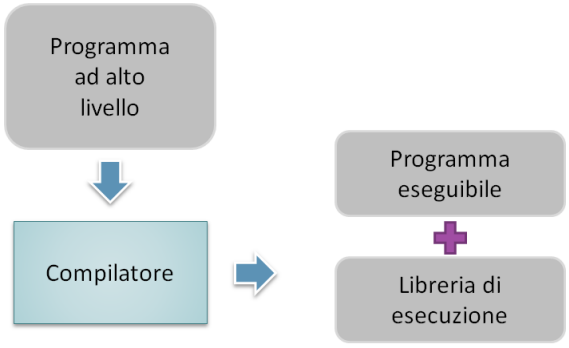
Livelli di astrazione

Il compilatore trasforma il codice ad un livello di astrazione inferiore. Tale trasformazione può avvenire da una macchina fisica (CPU), oppure subire un'ulteriore trasformazione per macchina virtuale.
Il **transpiling** è la compilazione allo stesso livello di astrazione, mentre la **compilazione** abbassa il livello. Comprende la disgregazione dei costrutti del linguaggio, ma mantenendo la semantica dell'istruzione (il compilatore si fa garante del mantenimento della semantica).
Il compilatore effettua tale traduzione, effettuando tali due azioni parziali:

- riscrivere il linguaggio in istruzioni più semplici (codice macchina)
- avvalersi di una libreria di esecuzione

Ciascun linguaggio possiede una *runtime library* che permette di iniziare un programma, molto prima del punto di accesso `main{}`.

Il processo attuato è quindi il seguente:



Librerie di esecuzione

La ***runtime library* (libreria di esecuzione)** è una libreria che offre agli applicativi dei meccanismi di base per il loro funzionamento. Essa è usata dal compilatore per implementare delle funzioni, integrate all'interno di un linguaggio di programmazione, durante di un programma (*runtime*).
Esse supportano le astrazioni del linguaggio di programmazione e forniscono un'interfaccia uniforme tra diversi SO per le funzioni ad esso demandate (`printf()`, …).

Lo standard C++ 2011 comprende i thread nella definizione del linguaggio per permettere di essere *runtime library independent*. Ciascun *vendor* dovette quindi prendersi carico dell'aggiornamento di tale libreria. Questo permise la portabilità del codice tra diverse piattaforme.
Un ulteriore punto d'ingresso, precedente al `main{}`, è l'allocazione delle varaibili globali mediante costruttori.
Lo **stack** (struttura a pila) permette di gestire chiamate annidate tra funzioni. Al suo interno sono salvate le locazioni di memoria alle quali il programma deve ritornare una volta raggiunto il `return`.
Il C++ supporta anche la gestione strutturata delle eccezioni, implementando un'ottimizzazione dello *stack* che si contrae (fino al primo blocco `try-catch{}`) quando una viene lanciata.

Alla partenza, un programma possiede un solo flusso d'esecuzione, ma può poi implementare la concorrenza: esecuzione parallela e indipendente di processi distinti. Ciascun processo concorrente possiede un suo *stack*.
Furono implementati, in C++ 2011, anche strumenti di sincronizzazione che fossere *platform independent*.

Implementazione del modello

La *runtime library* non può gestire tutte le astrazioni del modello di esecuzione, da sola. È necessario introdurre un processo di supervisione, effettuato in hardware (fosse software si passerebbe da un linguaggio compilato in un linguaggio interpretato) dalla **MMU (Memory Management Unit)**, in particolare legate all'isolamento e alle sue conseguenze.
La MMU è fondamentale per garantire che il malfunzionamento di un programma non influisca negativamente sugli altri.
L'isolamento assoluto, però, impedisce la comunicazione tra diversi processi. Bisogna quindi mediare tra le due situazioni estreme.

Processi

Per gestire l'esecuzione isolata di programmi, il SO creano **processi** distinti. Il SO crea un processo allocando della memoria ad esso (contenente lo stack) e inserisce il PID nella tabella apposita del sistema operativo (storata nella parte protetta del SO).
Per permettere all'utente di creare processi, sono forniti delle API con specifiche funzioni (in Windows `CreateProcess()` e `Clone()`).
Lo **scheduler** definisce a quale processo assegnare la CPU ad ogni **context switch** (commutazione di contesto che cambia il processo correntemente in esecuzione su una CPU). Quando un processo viene selezionato per l'esecuzione, il SO configura la CPU e ripristina lo stato del processo nell'istante in cui era stato interrotto.
Ciascun processo ha la convinzione di essere eseguito consecutivamente. Essi non hanno consapevolezza del meccanismo di *context switch* che viene operato su di essi.

I **sistemi *real-time*** si basano sulla garanzia di alcuni *commitment*, cioè il fatto che alcune azioni siano eseguite entro un determinato lasso temporale. Sono vincoli aggiuntivi dettati da un processo

Il costo per attraversare la barriera tra modo utente e supervisore è elevato. Sono quindi applicati delle ottimizzazioni, per esempio la bufferizzazione dei dati e la chiamata di una sola *system call* (un solo attraversamento).

Preparazione ed esecuzione di un processo

L'esecuzione di un programma prevede la creazione di un nuovo processo. Esso è inizializzato mediante le seguenti azioni:

- acquisire le risorse necessarie
- inizializzare lo stato

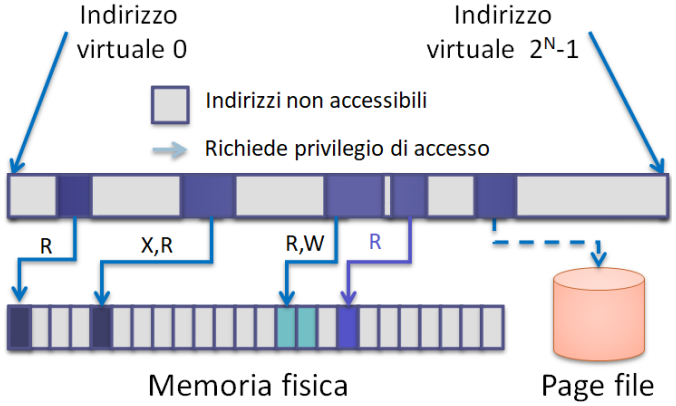
Analizzano più in dettagli si individuano alcune sotto-fasi:

- creazione dello spazio di indirizzamento
- caricamento dell'eseguibile
- caricamento delle librerie
- avvio dell'esecuzione

1. Spazio di indirizzamento

Il file eseguibile è caricato in RAM tramite **Demanding Paging**, cioè le pagine non sono caricate, ma settate *invalid* (I). Alla prima richiesta di accedere ad un'istruzione all'interno di una particolare pagina, viene attivata la *trap* **Page Fault** che carica in memoria la pagine d'interesse.
Le librerie da caricare sono spesso delle **DLL (librerie dinamiche)**. Non vengono linkate staticamente all'interno dell'eseguibile, ma sono nativamente presenti all'interno del sistema. Esse possono essere *shared*, cioè condivise tra più utenti.

Lo **spazio di indirizzamento** è un insieme di locazioni di memoria accessibili tramite indirizzo virtuale. La sua dimensione va da `0` a `MAX_INT`, ma segmentato. Ciascun segmento ha un determinato stato: *read-only*, *writable*, *missing*, …
LA MMU gestisce la traduzione tra indirizzo logico e indirizzo fisico.



Più processi distinti possono coesistere all'interno della memoria fisica, avendo però un spazio di indirizzamento virtuale distinto. Questo significa che lo stesso indirizzo, dei due processo, è mappato su un frame fisico distinto.

Il formato standard, in C e C++, dell'invocazione di `main()` è il seguente:

```
main(int argc, char** argv)
```

Il termine del processo è eseguito mediante l'invocazione della *system call* `exit()`.

In GCC/Linux, la funzione d'avvio ha come punto d'ingresso la funzione `_start()` (indicata dal linker). Tale funzione setta una serie di parametri e invoca la funzione `_libc_start_main()`.

```
int __libc_start_main(
    int(*main) (int, char**, char **),
    int argc,
    char** ubp_av,
    void(*init) (void),
    void(*fini) (void),
    void(*rtld_fini) (void),
    void* stack_end);
```

In Visual Studio/Windows, la funzione di avvio vede il punto di ingresso a seconda del tipo di progetto:

- opzione `/SUBSYSTEM:WINDOWS`: applicazione grafica, viene invocato `WinMainCRTStartup()`
- opzione `/SUBSYSTEM:WINDOWS`: applicazione a linea di comando

Ulteriore distinzione è effettuata in base al set di caratteri utilizzato:

- `ANSI`
- `UNICODE`

Il punto d'ingresso utente riflette i quattro casi:

```
int main(int argc, char* argv[]);
int wmain(int argc, wchar_t* argv[]);
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow);
int wWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR lpCmdLine, int nCmdShow);
```

Il comportamento è simile, vengono eseguite le seguenti fasi:

- inizializzazione della libreria di supporto
- invocazione dei costruttori
- preparazione degli argomenti
- funzione principale
- invocazione dei distruttori
- terminazione del processo

- **dynamic zone**: blocchi di memoria allocate dinamicamente dal programmatore, il cui ciclo di vita inizia tramite la funzione `malloc()` e termina all'invocazione della `free()`

Le zone di memoria dinamica permettono di definire dei cicli di vita intermedi tra la durata dell'intero programma e l'esecuzione di una singola funzione. Lo spazio necessario è allocato solo finché utile, poi rilasciato.

Criteri di accesso

La corrispondenza tra indirizzi virtuali e pagine fisiche è corredata di metadati che definiscono quali operazioni sono lecite sulla memoria:

- `X`: execute
- `R`: read
- `W`: write
- `C`: copy on write

Lo stato **Copy on Write** indica che la pagina è accessibile in lettura, ma al primo tentativo di scrittura viene sollevata un'eccezione che ne crea una copia per lasciare intatto l'originale. Viene quindi sostituito il puntatore prima di permettere la scrittura.
Tale modalità garantisce alta efficienza di accesso (non sempre necessaria la duplicazione) pur separando i contenuti. Tale meccanismo è molto utile nella *System Call* `fork()` che clona il processo dando al figlio lo stesso spazio di indirizzamento.

I tentativi di azioni illecite su pagine mappate oppure di lettura di pagine non mappate, provocano un'eccezione:

- Windows: **Access Violation**
- Linux: **Segmentation Fault**

Come conseguenza principale, il processo che ha scatenato tale interruzione viene terminato.

Organizzazione dello spazio di indirizzamento

Quando un processo viene creato, il suo spazio di indirizzamento viene suddiviso in diverse aree, ciascuna dotata dei propri criteri di accesso.
Esse permettono ai linguaggi di programmazione di offrire il loro modello semantico con diversi classi di variabili, ciascuna dotata di un ciclo di vita regolata da specifiche norme.

Linux e Windows possiedono due organizzazioni dello spazio di indirizzamento (su $32bit$) leggermente differenti, ma accumunate dalla presenza delle medesime aree:

Le richieste di allocazione giungono solitamente dalla *Runtime Library,* che offre l'astrazione tipica del linguaggio di programmazione, con una semantica leggermente differente.

Linguaggio:	Allocazione:	Liberazione:
C	`malloc()`	`free()`
C++	`new`	`delete`

Una porzione dello spazio di indirizzamento non è accessibile ai programmi in modalità utente, ma solo al Sistema Operativo (quando la CPU è in `ring 0`).
Tale area protetta contiene sia informazioni specifiche per ciascun processo, sia informazioni comuni a tutti i processi. Il Sistema Operativo vi memorizza le sue strutture dati che gli permettono di conoscere la configurazione dell'intera macchina.
L'area protetta occupa metà dello spazio di indirizzamento per Windows, mentre per Linux solo un quarto.

La **randomizzazione del punto di ingresso** è un meccanismo che somma, all'indirizzo iniziale del codice, un Δ variabile. Ciò è effettuato dal *loader* e impedisce ai virus di poter attaccare punti specifici del codice (se noti gli indirizzi ai quali sono staticamente allocati).

Blocco:	Permessi:		
codice eseguibile	R		X
costanti	R		
variabili globali	R	W	
stack	R	W	
heap	R	W	

Ciclo di vita delle variabili

Per ciascuna variabile è definito un **ciclo di vita**: un tempo in cui è lecito accedere alla variabile, avendo la certezza che le informazioni contenute sono conservate.
Il modello di esecuzione del C distingue diverse classi di variabili:

- globali: ciclo di vita pari a quello del processo
- locali: ciclo di vita pari a quello del suo lexical scope (es. funzione)
- dinamiche: ciclo di vita definito dal programmatore (anche superiore a quello della funzione in cui è definita)

Per le variabili globali, il ciclo di vita corrisponde all'intero ciclo di vita del programma. Questo è possibile perché il *linker* posiziona le variabili globali in un indirizzo specifico, garantendone una reperibilità statica. Tale spazio esiste dall'inizio dell'esecuzione del programma (già da prima che venga eseguita la prima istruzione) perché già mappato alla creazione dello spazio di indirizzamento.
Se inizializzata, la variabile viene mappata sul valore che essa assume.
Il segmento dedicato alle variabili globali non è accessibile all'allocazione di altri processi, quindi queste esisteranno fino al termine del processo.

In C++ il bocco ottenuto è sempre inizializzato a partire da una funzione specificata insieme alla classe, detta **Costruttore**. Essa garantisce che, una volta ottenuto il blocco di memoria, in questo vengano inserite opportune informazioni prima che l'utente possa accedervi.
Il costrutto `new` restituisce il puntatore all'oggetto inizializzato. Il contenuto diventa così accessibile al resto del programma.
Per allocare un array di `n` oggetti, C++ fornisce il seguente costrutto:

```
new NomeClasse[n]();
```

Viene allocata una sequenza di `n` oggetti e il costruttore inizializza i singoli oggetti. L'operatore `new` restituisce il puntatore all'array.

Rilascio della memoria

L'allocazione dinamica di un blocco di memoria corrisponde all'assunzione dell'impegno (da parte del programmatore) della gestione anche del suo rilascio. Esso è possibile tramite opportune funzioni di libreria.
Poiché ogni funzione di allocazione mantiene le proprie strutture dati, occorre che un blocco sia rilasciato dalla funzione duale di quella con cui è stato allocato.

Costrutto di allocazione:	Costrutto di rilascio:
`malloc()`	`free()`
`new`	`delete`
`new[]`	`delete[]`

Particolarmente critico è il rilascio di array di oggetti: è molto facile dimenticarsi le parentesi quadre, ma questo significa rilasciare solamente il primo oggetto e mantenere intatti tutti i successivi.
Se il blocco viene rilasciato con la funzione sbagliata si rischia di corrompere le strutture dati degli **allocatori**, con conseguenze imprevedibili.
È importante che le librerie di codice espongano tanto la funzione per allocare, quanto la funzione per liberare le porzioni memoria.

Non rilasciare le risorse allocate porta a saturare rapidamente lo *heap* (di dimensione fissata) fornito, costringendo il Sistema Operativo a rifiutare le richieste di ulteriori allocazioni e sollevare l'opportuna eccezione.

Puntatori e loro utilizzo

Un **puntatore** permette un accesso diretto ad un blocco di memoria.
Tale blocco di memoria può appartenere ad altri oggetti:

```
int A = 10;
int* pA = &A;
```

oppure allocato allo scopo specifico:

```
int* pB = new int(24);
```

Si ha la criticità sull'arco temporale nel quale è lecito accedere ad dato puntato (ad esempio da `pA`).
È lecito accedervi finché è lecito accedere all'oggetto puntato (in questo caso `A`), ma al momento della creazione del puntatore, si è perso il legame concettuale con l'oggetto `A`.

La responsabilità è riferita a chi e quando deve rilasciare un puntatore. Per rilasciare un puntatore non è sufficiente porlo a `NULL`, perché qualcuno potrebbe averne creato una copia.

Ambiguità dei puntatori

I puntatori possiedono delle ambiguità intrinseche:

- dimensione del blocco puntato
- intervallo temporale in cui è garantito l'accesso
- permessi di modifica
- responsabilità del rilascio
- origine del puntatore (copia od originale)

I puntatori sono problematici, ma non si può fare a meno di usarli.

Uso dei puntatori

Ci sono ragioni molto valide per utilizzare i puntatori, nonostante le loro criticità:

- accedere a informazioni fornite da altri
- accedere ad array
- accedere allo *heap*
- creare strutture dati complesse

Nel caso più semplice ed molto frequente, i puntatori possono essere uno strumento per accedere *hic et nunc* (qui e ora) ad un'informazione fornita da altri.
Esempio è la funzione `scanf()`, ad essa è passato come parametro la locazione di memoria nella quale memorizzare il risultato di ritornare. La sua implementazione interna prevede che non venga effettuata alcuna copia di quel puntatore (utilizzo esclusivamente *qui e ora*), ma non vi è garanzia di tale comportamento.
Tale passaggio di parametri (passaggio per valore dell'indirizzo), per quanto ambiguo, è però necessario per permettere a tali funzioni di ritornare più risultati, lasciando la `return` principale ad un valore numerico indicante il successo o il fallimento della funzione richiesta.
In questo tipo di utilizzo dei puntatori, la responsabilità per la gestione della memoria del dato è totalmente esterna all'osservatore.

Esempio:

> La seguente funzione tenta di leggere un dato. Al pari della `scanf()`, utilizza il valore di ritorno per garantire il successo dell'operazione e il parametro (passato come puntatore) per depositarci il valore letto.

```
int read_data1(int* result) {

 //Se il puntatore sembra valido e ci sono dati...
 if (result != NULL && some_data_available()) {

        //accedi in scrittura alla memoria
        *result = get_some_data();

        //indica operazione eseguita correttamente
        return 1;
```

In questo scenario non è chiaro di chi sia la responsabilità e quale sia il momento richiesto per il rilascio della memoria. In programmi *multi-thread* la gestione si complica ulteriormente.
L'utilizzo di un puntatore per accedere allo *heap* è il caso base di tutte le strutture dinamiche.

Esempio:

> Questa seconda versione di una funzione di lettura di dati si assume la responsabilità di allocare internamente il blocco di memoria di dimensione sufficiente e di ritornarlo al chiamante.
>
> ```
> int* read_data2() {
>     if (some_data_available()) {
>         int* ptr = (int*)malloc(sizeof(int));
>         *ptr = get_some_data();
>         //indica operazione eseguita correttamente
>         return ptr;
>     }
>     else
>         //operazione fallita
>         return NULL;
> }
>
> int* result = read_data2();
> ...
> free(result);
> ```
>
> Questa modalità pone due fardelli gravosi al chiamante: la responsabilità del rilascio della memoria (allocatore e deallocatore sono due entità distinte) e la scelta del modo corretto per farlo (in relazione alla funzione utilizzata per allocare).

I puntatori sono utilizzati, infine, anche per implementare strutture dati composte. Le liste, i grafi e le mappe sono strutture dati che richiedono una quantità di dati variabile nel tempo e nelle quali tali dati referenziano parte delle strutture stesse (scorrere lungo la collezione).

Esempio:

> Si implementa una lista che punta solo in avanti. Si fornisce solo il puntatore alla testa di questa struttura dati dinamica.
>
> ```
> struct simple_list {
>     int data;
>     struct simple_list *next;
> };
>
> struct simple_list *head;
> // head è responsabile di tutte le proprie parti quando si rilascia la lista, occorre
> liberarne tutti gli elementi
> ```
>
> Il puntatore alla testa (`head`) è responsabile di tutte le sue sottoparti: per rilasciare `head` è necessario rilasciare prima tutti i nodi della lista.

Problemi legati ai puntatori

In C, la gestione della memoria è totalmente affidata al programmatore: non si ha alcun supporto sintattico per distinguere i diversi tipi di puntatori o i diversi utilizzi che è possibile fare di un puntatore. Appaiono tutti semplicemente come `<tipo>*`.

Un **wild pointer** (puntatore incontrollato) è un puntatore non inizializzato ad alcun valore lecito, che punta quindi ad una locazione di memoria impredicibile.
Se non si inizializza un puntatore e lo si usa, il suo contenuto potrebbe puntare ovunque, causando una violazione d'accesso oppure corrompendo un'area di memoria in uso ad altre parti del programma.
Spesso i compilatori sono in grado di identificare questo tipo di problematica e li segnalano tramite dei warning.

Esempio: dangling pointer

Il seguente codice mostra un dangling pointer.
Le parentesi graffe identificano un blocco di codice all'interno di una funzione. Le variabili locali definite al loro interno terminano il loro ciclo di vita al temine del blocco e lo stack si contrae.
Quindi il contenuto di `*ptr` fuori dalle graffe potrebbe essere qualunque cosa.

```
{
    char* ptr = NULL;

    {  // inizio di un nuovo blocco
        char ch = '!';
        ptr = &ch;

    }   // fine blocco: lo stack si contrae
        // le variabili qui definite cessano di esistere

    printf("%c", *ptr);    //contenuto impredicibile
}
```

Nello stack vengono generate le variabili `ptr` e `ch`, poi viene fatta puntare `ptr` a `ch` (`ptr = &ch`). Chiudendo le graffe `ch` è eliminato, ma `ptr` continua a puntare alla stessa cella di memoria. Richiamando la `printf()` viene stampato il contenuto della cella dove prima vi era `ch`, ma adesso conterrà un altro dato.

Esempio: memory leakeage

Il seguente codice mostra un memory leakeage.
Si alloca un blocco di memoria e si perde la variabile locale che lo punta, prima di averlo rilasciato.

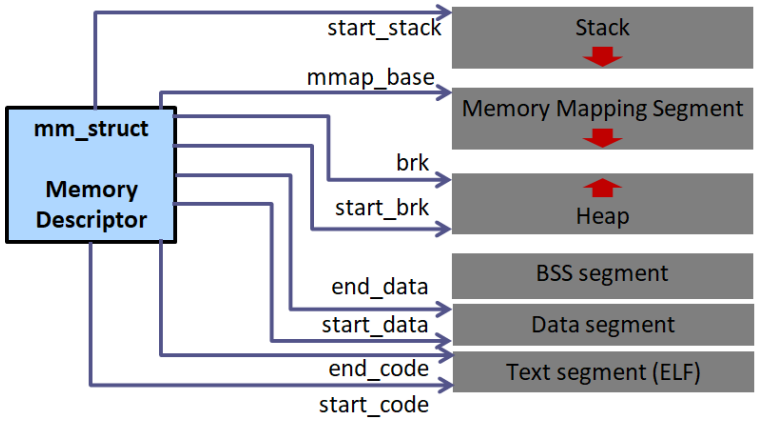
```
{
    char* ptr = NULL;

    ptr = (char*)malloc(10);        // alloco un blocco
    strncpy(ptr, 10, "Leakage!");   // lo uso
    printf("%s\n", ptr);
}                                   // ne perdo le tracce
```

Alla chiusura delle graffe `ptr` viene eliminato ed essendo l'unico riferimento alla stringa, questa non può più essere acceduta, liberata e neanche sovrascritta.

Gestione dei puntatori

Chi alloca un blocco di memoria è responsabile di mettere in atto un meccanismo che ne garantisca il successivo rilascio. Costui è detto possessore del puntatore (**pointer owner**).
Si ha un problema quando tale puntatore viene copiato. In quel caso i possessori diventano due e devono sincronizzarsi: uno solo dei due deve rilasciare il blocco. È necessario introdurre un

All'interno del *PCB* vi è la struttura `mm_struct` che contiene un riferimento alla struttura dello spazio di indirizzamento del processo a cui è riferita.
Tra i campi di tale *Memory Descriptor* vi sono i delimitatori dello heap (`brk` e `start_brk`), l'indirizzo di origine dello stack (`start_stack`) che può crescere fino all'indirizzo `mmap_base`.



Se la richiesta di spazio è superiore a quello disponibile nello heap in quel momento, il programma potrebbe spostare più avanti il `brk` utilizzando la seguente *System Call*:

```
int brk(void *end_data_segment);
```

Questa funzione permette di impostare un nuovo valore di `brk`, quindi si aumentando che diminuendo lo *heap*. Diminuire lo *heap* significa eliminare tutto il contenuto escluso (una `free()` senza consapevolezza).
Se l'estensione dello heap invaderebbe il blocco successivo, la richiesta è rifiutata e si ritorna il codice di errore `-1`.
La variante `sbrk()` richiede come parametro il valore intero di cui modificare la dimensione dello heap:

```
void *sbrk(intptr_t increment);
```

Tale *System Call* ha tutte le limitazioni della precedente.

In caso di richieste onerose (grosse dimensioni) di allocazione, la `malloc()` decide di non utilizzare lo heap per evitare problemi di compattazione. Invoca piuttosto la `mmap()` per estendere il numero di blocchi mappati utilizzando un nuovo segmento dello spazio di indirizzamento (non ancora usato) per mapparvi una nuova pagina di memoria.

2. Allocazione in Windows

In Windows ogni processo può avere più *heap*. Alla sua creazione ne possiede uno solo, creato dalla `global_alloc()` ed è quello a cui fa riferimento la funzione `malloc()` della libreria standard.
È possibile creare ulteriori heap per gestire allocazioni specifiche per particolari tipi di dati (dati omogenei da non mischiare con altri).

Per creare un nuovo si utilizza la seguente *System Call*:

```
HANDLE HeapCreate(...)
```

Essa richiede come parametri la politica di allocazione, la dimensione iniziale e quella massima. Ritorna un riferimento opaco, non l'indirizzo di riferimento del nuovo *heap*.

LINGUAGGIO C++

Introduzione al linguaggio C++

Introduzione al linguaggio

Il **linguaggio C++** è un linguaggio di programmazione vasto ed articolato che supporta diversi stili di programmazione:

- programmazione strutturata
- programmazione funzionale
- programmazione ad oggetti
- programmazione generica

Il C++ è un'estensione del C; essi sono quindi compatibili a livello di sorgente e di moduli oggetto. È quindi possibile avere una funzione C++ che chiama una funzione C.
Il C++ possiede però delle estensioni uniche che il C non supporta.

Il C++ fu sviluppato circa 30 anni fa da Stroustrup secondo le linee guida di:

- garantire espressività al programmatore: affrontare problemi complessi senza doversi occupare dei dettagli
- non penalizzare le prestazioni: a differenza di Java, il C++ permette di scegliere il livello di attenzione alle prestazioni

In particolare, sotto il profilo delle prestazioni, le funzionalità che il programmatore non utilizza non devono pesare sull'esecuzione del programma.
Per esempio in Java è possibile sviluppare un programma non polimorfo, ma la sua esecuzione subirà comunque delle penalizzazioni dovute al *polimorfismo* (controllo del tipo di oggetto puntato e di eventuali `@override` per ogni metodo invocato). Viceversa in C++ è possibile scegliere se applicare il polimorfismo (perché utile nella programmazione), oppure evitarlo (per incrementare di qualche ciclo macchina le prestazioni).

In questo corso analizzeremo il C++ prevalentemente come linguaggio ad oggetti, successivamente come linguaggio generico e infine la gestione strutturata delle eccezioni.
Il C++ offre i seguenti supporti alla programmazione ad oggetti:

- incapsulamento: i dati sono conservati all'interno di un oggetto che ne dischiude l'accesso al mondo esterno utilizzando i principi di private/protected/…
- composizione: un oggetto può contenere al suo interno altri oggetti. In Java è solo di puntatori, mentre in C++ può essere anche di parti)
- ereditarietà: una classe può specificare le caratteristiche di un'altra classe. A differenza del Java, il C++ supporta l'ereditarietà multipla
- polimorfismo: un metodo descritto da una classe può assumere valori di un qualunque tipo descritto da una sua sottoclasse

- caratteri:
  - `char` / `unsigned char`
  - `wchar_t`
- valori logici:
  - `bool`

Gli **interi** possiedono il problema dell'effettiva dimensione delle variabili (in bit). L'implementazione dipende dal singolo compilatore, quindi i programmatori spesso definiscono `typedef INT_32BIT` quando vogliono avere il controllo dello spazio effettivamente utilizzato da ogni variabile.
I risultati delle operazioni tra interi sono esatte (no approssimazione), l'unica situazione critica da gestire è l'**overflow** (si tenta di rappresentare un numero eccessivo per i bit nei quali è memorizzato).

I **reali** introducono un problema di approssimazione, sui calcoli che li riguardano, perché si utilizza un numero finito di bit per la loro rappresentazione e i restanti valori sono troncati. Per determinare l'uguaglianza tra due reali è quindi necessario fissare un intervallo ε di variazione all'interno del quale i due valori troncati sono uguali.
Il frutto di un'operazione non consentita da come risultato un **NaN (Not a Number)**, che è però diverso da sé stesso (definito dallo standard). Un reale può contenere un NaN e verificare tale situazione è complesso se non la si conosce.
Un reale può anche contenere `INFINITY` quando si divide un valore non nullo, per un valore nullo. La somma di due `INFINITY` da come risultato NaN.
In definitiva i reali sono molto diversi dagli interi: per i secondi le operazioni sono sempre giuste (a meno di overflow), mentre per i primi sono approssimate.

I **caratteri** rappresentano singole lettere o numeri nella loro rappresentazione ASCII. La struttura `wchar_t` indica un carattere nella sua rappresentazione *wide*.

Il C++ introduce i **valori logici** espressi nel tipo `bool`. Essi possono assumere solamente i valori `TRUE` o `FALSE`. Essi sono costanti.

Il linguaggio offre anche alcuni tipi derivati:
- eumerazioni
- array
- puntatori
- riferimenti
- struct
- union

Le **enumerazioni** sono un insieme definito di possibili valori che un variabile può assumere. In C++ sono resi come una serie di interi il cui scopo non sono operazioni, ma solo etichettare alcune grandezze.
La sintassi per definirle è la seguente:

```
enum enumerationName {label1, label2, ...};
```

Le **union** (già presenti nel C) sintatticamente sembrano una `struct` con la differenza è che i campi non sono sequenziali, ma sovrapposti (in alternativa).
Servono a basso livello per lavorare con l'hardware, perché permettono di interpretare una data sequenza di bit in più modi alternativi. È tanto utile quando pericolosa, perché si rischia di perdere il tipo originale e inserirvi dentro un puntatore costituisce un meccanismo di *forging* di indirizzi.
Una *union* modella un'area di memoria che può contenere valori di tipo diverso in momenti diversi.
La sintassi per definirle è la seguente:

```
union mytypes_t {
    char c; int i; float f;
};
```

Classi e oggetti

Una **classe** è un costrutto composto da:

- variabili istanza
- funzioni membro

Analogamente ad una struct, una classe è una struttura dati: un blocco di memoria consecutiva dentro cui sono riportati dei valori etichettati da un nome. Tali nomi sono detti **variabili istanza**.
I metodi sono sintatticamente appartenenti ad una classe, ma in memoria sono presenti solo i valori.
I **metodi** (anche detti **funzioni membro**) sono funzioni che possiedono un parametro implicito in più rispetto a quelli dichiarati: il parametro `&this` (puntatore all'oggetto a cui fanno riferimento).

La sintassi per la definizione di una classe è la seguente:

```
class className {
private:
    type1 field1;
    type2 field2;
public:
    type3 method1( /* paramethers*/) { /* implementation */ }
    type4 method2( /* paramethers*/ );
};
```

Importante è il punto e virgola finale (assente in Java) la cui assenza compromette la compilazione.
All'interno della classe è possibile creare delle sezioni di visibilità (`private/protected/public`) contenenti tutto ciò che si vuole associare al relativo livello di visibilità. Spesso, come diretta conseguenza dell'*incapsulamento*, si pongono le variabili istanza private (per il principio di **data hiding**) e i metodi pubblici (per esporre delle funzionalità sui dati).

Per le funzioni memobro, nella sezione dedicata ai parametri, non è necessario specificare `this`, ma viene automaticamente incluso. I metodi sono quindi implementati come normali funzioni.
L'implementazione dei metodi può essere:

- *inline*: all'interno della definizione della classe
- *separata*: nella stessa unità di compilazione o in un altro modulo collegato

L'implementazione *separata* non era possibile in Java, ma è molto utile perché permette di tenere la definizione di una classe in un file `.h` e la sua implementazione in un file `.cpp`.
Il compilatore, per implementazioni *inline*, non effettua neanche una chiamata a funzione, ma esegue direttamente il codice specificato per evitare l'*overhead* della chiamata. È quindi preferibile

```
public:
    int doWork();
};
#endif
```

Le direttive per il precompilatore sono indicate con `#` (cancelletto) e permettono di decidere se compilare o meno una porzione di codice.
Tutti i file `.cpp` che utilizzano `MyClass` devono includere il `File.h` con il rischio che questo venga incluso due volte (da due file sorgenti diversi). La direttiva `#ifndef` permette di evitare di leggere la seconda `#include<File.h>` come un tentativo di ridefinire `MyClass` ed evitare così di sollevare un'eccezione.
Definendo `MYCLASS` dentro al costrutto condizionale `#ifndef`, la definizione della classe è letta solamente una volta. Il secondo tentativo (quando la costante `MYCLASS` è già stata definita) non avrà neppure compilato quel frammento di codice perché il precompilatore ha rimosso quelle righe.
Nei compilatori moderni questa intera protezione è sostituita da una sola istruzione:

```
#pragma once
```

File.cpp:

```
#include "File.h"

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

I file di codice che vogliono utilizzare la classe `MyClass` devono includere il file `.h` nella quale questa è stata definita.
In `File.cpp` viene implementato il metodo `doWork()` che era già stato annunciato nella definizione della classe.

Per definire il metodo è necessario farlo precedere dal nome della classe separato dallo ***scope operator*** (`::`) per distinguerlo da una semplice funzione.

Disposizione in memoria

Un **oggetto** è un'istanza di una classe. Per ogni classe è possibile istanziare più oggetti.
Ad ogni oggetto corrisponde un blocco di memoria contenente tutti gli attributi non statici (pubblici, privati e protetti) nell'ordine in cui sono stati definiti ed eventuali altre informazioni utili al compilatore (riferite a polimorfismo e similari).

In Java gli oggetti esistono solo nello *heap* (tramite costrutto `new`), ma in C++ questi possono essere allocati in varie aree della memoria, a seconda del tipo di variabile di cui si tratti:

- memoria globale
- stack (*local store*)
- heap (*free store*)
- all'interno di un altro oggetto (a sua volta in una delle tre aree precedenti)

In assenza di un costruttore esplicito, il compilatore provvede a crearne uno di default che inizializza tutte le variabili al loro **valore di default**:

- boolean: `FALSE`
- interi: `0`
- reali: `0.0`
- puntatori: `NULL`
- oggetti: valore di ritorno del loro costruttore senza parametri

A livello sintattico è possibile inserire, tra le parentesi dei parametri e dell'implementazione, dei due punti seguiti dall'elenco dei campi e dal valore a cui inizializzarli prima di svolgere il codice interno al costruttore:

```
MyClass(): field(0) {}
```

Tale struttura è analoga alla sintassi classica:

```
MyClass() { field = 0 }
```

ma la prima è preferita per lasciare il meno possibile all'interno delle graffe e ridurre le fonti di confusione.

Esempio:

Di seguito è implementata una classe con due distinti costruttori (ciascuno con una diversa sintassi). La presenza di due diversi costruttori è permessa dalla differenza tra i parametri che essi richiedono (sono quindi distinguibili).

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

private:
    int code;
};
```

Distruttore

A differenza del Java, in C++ esiste il **distruttore**: un metodo che libera le risorse eventualmente possedute. Compie il processo inverso del costruttore e, analogamente a lui, possiede il nome della classe a cui appartiene, ma preceduta da ~ (tilde, richiamabile su Windows come `alt+126`).
Il distruttore non ha parametri ed è chiamato solo dal compilatore, non è mai chiamato esplicitamente dal programmatore. Farlo sarebbe un errore grave perché non è possibile invocare un metodo di un oggetto che non esiste più (l'indirizzo di ritorno non ha più significato).
Un distruttore si deve occupare di chiudere eventuali file aperti, portare a `NULL` eventuali puntatori e liberare le locazioni di memoria dinamicamente allocate.

Nel C++ moderno si utilizzano pochi distruttori, perché non si utilizzano puntatori nativi, piuttosto *Smart Pointer* che non richiedono una distruzione parametrizzata.

L'**operatore `new`** esegue due compiti distinti e consecutivi:

1. acquisisce, dall'allocatore della *libreria di esecuzione*, un blocco di memoria di dimensioni opportune
2. inizializza tale blocco invocando l'opportuno costruttore.

L'**operatore `delete`** esegue i compiti duali, in ordine inverso:

1. invoca il distruttore dell'oggetto per rilasciarne eventuali risorse
2. rilascia la memoria occupata dall'oggetto attraverso la libreria di esecuzione

Gli operatori `new`/`delete` non si occupano esplicitamente di invocare costruttori/distruttori, ma essi sono invocati contestualmente per dare un senso più generale all'intera operazione di allocazione/rilascio della memoria.

Esempio:

Frammento di codice di allocazione e di rilascio di un oggetto. Si utilizza un puntatore ad un oggetto, quindi si tratta di una variabile dinamica.

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

Il frammento contiene un errore logico, perché l'invocazione di new può modificare l'ultimo errore ottenibile tramite `GetLastError()`. È preferibile quindi memorizzare l'ultimo errore in un avariabile temporanea prima di invocare il `new`.

Il `new` e il `delete` devono essere contenuti in funzioni diverse, oppure in due diversi flussi alternativi di una stessa funzione (dentro un `if{}`), altrimenti non ha senso utilizzare una variabile dinamica.

Dopo invocare il `delete`, è buona abitudine porre il puntatore a `NULL`, per evitare accessi non più permessi alla memoria.

È possibile allocare **array dinamici** di oggetti tramite il costrutto:

```
MyClass *vet_ptr = new MyClass[count];
```

In C++ per ciascun oggetto viene invocato il costruttore, mentre Java istanzia un vettore di puntatori a `NULL`.

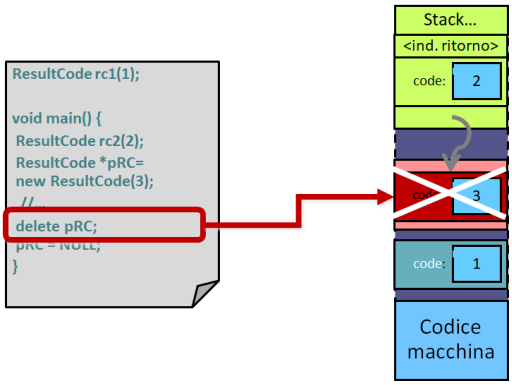
Per rilasciare il blocco (in C++) è necessario utilizzare l'operatore duale `delete[]`. Il compilatore, dato un puntatore, non ha modo di conoscere se esso è riferito ad un singolo oggetto, oppure ad un array. È quindi necessario aggiungere le parentesi quadre:

```
delete[] vet_ptr;
```

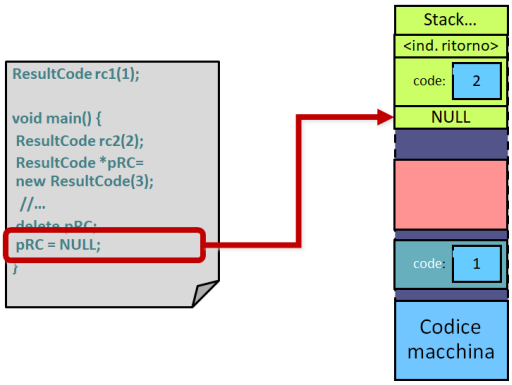
Il compilatore conosce così che la posizione prima di quella fornita contiene un intero che rappresenta il numero di elementi dell'array. Fa quindi un passo indietro, impara tale numero ed è così in grado di liberare tutti gli oggetti e il contatore stesso.

È importante che non venga modificata la cella di memoria contente il contatore (tramite aritmetica dei puntatori o simili), perché la `delete` non sarebbe più in grado di rilasciare l'array. Per questo motivo, in modalità debug, il blocco richiesto per l'allocazione di un array è aumentato di un Δ prima e uno dopo. In tali blocchi aggiuntivi scrive un **fingerprint** (codice specifico personale), nella `delete` controlla il *fingerprint* e se questo non è stato sovrascritto è indice di assenza di errori.

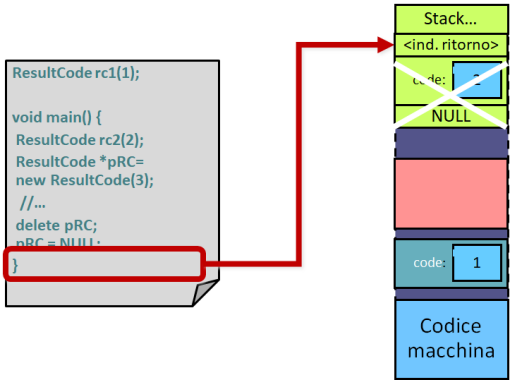
Il `delete pRC` rilascia la memoria precedentemente allocata e invoca il distruttore:



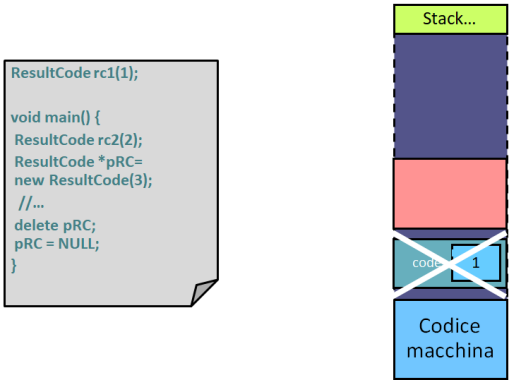
Porre `pRC` a `NULL` significa pulire la posizione di `pRC` nello stack:



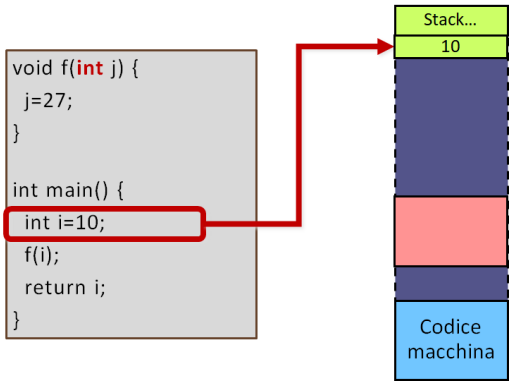
La parentesi graffa di chiusura della funzione sottende molto codice. Vengono rilasciate tutte le variabili locali (lo stack si contrae), si estrae l'indirizzo di ritorno memorizzato all'inizio e di esegue l'effettivo `return`.



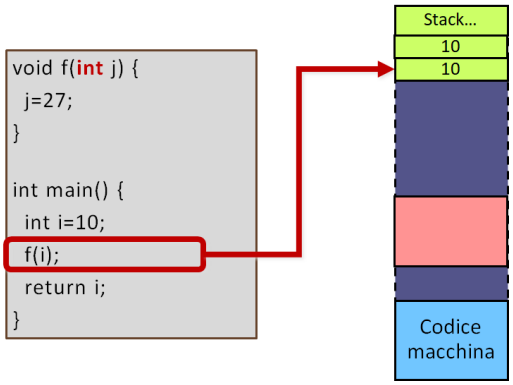
Al termine del programma viene rimossa anche la variabile globale:



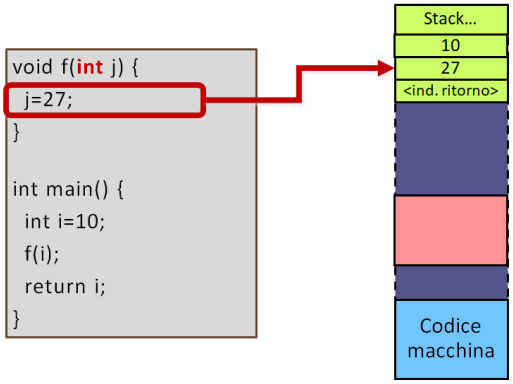
La situazione iniziale è la seguente:



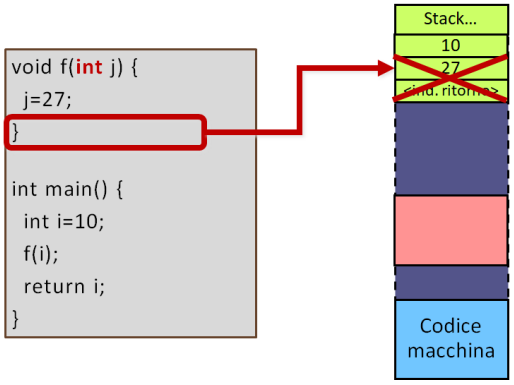
All'invocazione della funzione, lo stack viene abbassoto di 4*byte* perché la variabile `i` venga duplicata:



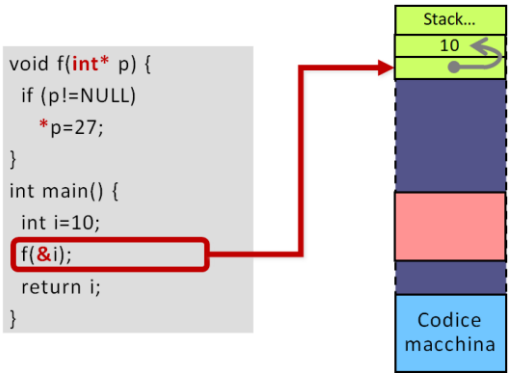
Viene memorizzato l'indirizzo a cui tornare dopo l'esecuzione id `f()` e se ne lancia l'esecuzione. La funzione `f()` agirà sul parametro (come indicato da codice):



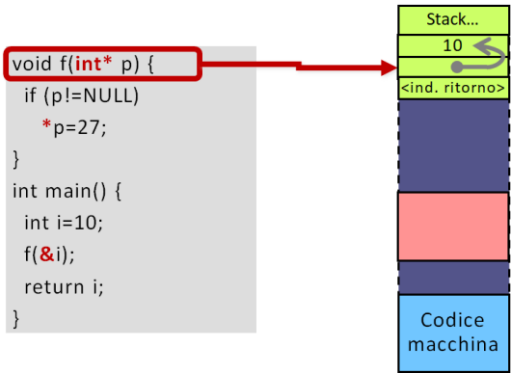
Una volta terminata la funzione, lo stack si contrae (da funzione chiamata/chiamante a seconda della convenzione) fino a tornare alla situazione iniziale (precedente all'invocazione di `f()`):



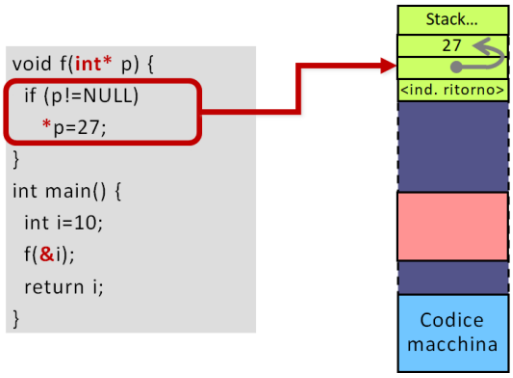
La chiamata per indirizzo richiede (come parametro attuale) la creazione di un puntatore alla cella di memoria di interesse:



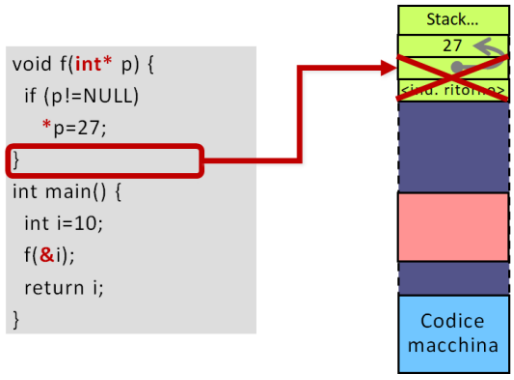
La chiamata di `f()` aggiunge allo stack l'indirizzo di ritorno e passa l'esecuzione ad f() stessa, che vede il parametro:



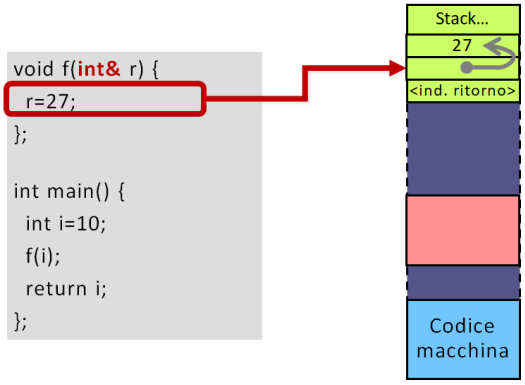
Viene poi eseguito il codice specifico di `f()` che, essendo `p` non nullo, lo dereferenzia per scrivere un nuovo valore nella cella puntata:



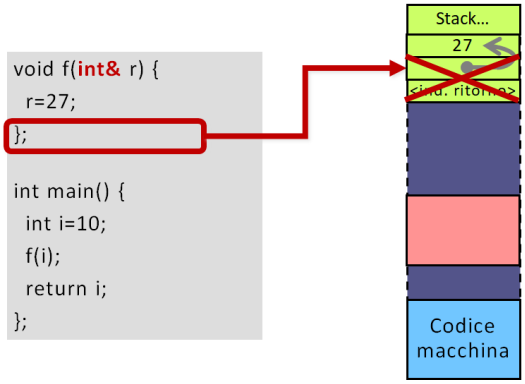
Al termine della funzione `f()` lo stack si contrae (distruggendo l'indirizzo di ritorno e il parametro), ma la cella originale di `i` esiste ancora, con il valore aggiornato da `f()`:



La chiamata di `f()` necessita della scrittura, nello stack, dell'indirizzo di ritorno. All'interno del codice di `f()` non è necessario controllare che il puntatore sia nullo e neanche referenziarlo esplicitamente (già fatto dal compilatore):



La terminazione di `f()` fa contrarre lo stack, con conseguente perdita di informazioni, ma il valore aggiornato di `i` sopravvive ed è visibile anche da `main()`:



Con gli oggetti, passare *by reference*, permette di migliorare l'efficienza perché evita di copiare tutto il contenuto per averlo come parametro, ma solo pochi byte del puntatore. Per evitare che il chiamato modifichi tale oggetto passato, è possibile dichiarare il parametro come `const` che garantisce che esso sia *read-only* (non permesse azioni di modifica).

Eccezioni

Una **eccezione** è un meccanismo che permette di trasferire il flusso di esecuzione ad un punto precedente del codice, dove si ha la possibilità di gestirlo.
Esse permettono di indicare due informazioni:

- descrivere l'errore che è avvenuto (meglio di un semplice codice)
- indicare come gestire la situazione anomala

L'eccezione è una struttura dati che contengono una descrizione formale dell'errore che è avvenuto.

In Java c'è la seguente gestione:

- classe `Throwable`:
  - classe `Error`: situazioni ingestibili
  - classe `Exception`: situazioni gestibili
    - classe `RuntimeException`: situazioni comuni che non è necessario specificare
    - classe `CheckedException`: situazioni trattate sul posto (try-catch) oppure ribaltate sul chiamante

Esempi di `Error` sono `StackOverflow` oppure `OutOfMemory`, mentre esempio di `RuntimeException` è `IndexOutOBound` che non è necessario specificare (*statement* `throws`) perché troppo comuni perché sia necessario complicare il codice.
Gestire un errore semplicemente stampando a video un messaggio di errore è una pessima gestione perché non risolve il problema e spesso i *device* non possiedono neppure una *consolle* su cui stampare il messaggio.

Il C++ permette di lanciare un'eccezione con un tipo qualunque (numerico, string, ecc…) mentre Java pone il limite forte che vengano lanciate solo figlie derivate da `Throwable`.
Dal C++ 2011, per retrocompatibilità, fu mantenuta la possibilità di lanciare qualunque tipio, ma lo sconsiglia fortemente e fornisce la classe `std::Exception` come superclasse dell'albero di ereditarietà per gestire i possibili tipi di gestione. Il programmatore è portato a derivare la classe `std::Exception` in una più specifica che contiene tutte le informazioni specifiche per gestire l'errore che si desidera gestire.
Un'eccezione è spesso corredata da una stringa che descrive sommariamente l'errore che è avvenuto. Il messaggio dev'essere genericamente comprensibile per permettere interoperabilità tra diversi team di sviluppatori che operano sullo stesso codice in tempi diversi (lingua inglese obbligatoria).

Signature e conseguenze delle eccezioni

Gestire un'eccezione impone di interrompere il *workflow* attuale per saltare al codice necessario per svolgere operazioni che riportino la situazione alla normalità.
È necessario saltare tutto il codice intermedio ed evitare il rischio di dimenticarsi di propagare una indicazione di errore.
Si notifica al sistema la presenza di un'eccezione creando un dato qualsiasi (meglio sottoclasse di `Exception`) e passandolo come valore alla seguente *keyword*:

```
throw
```

Classe `exception`

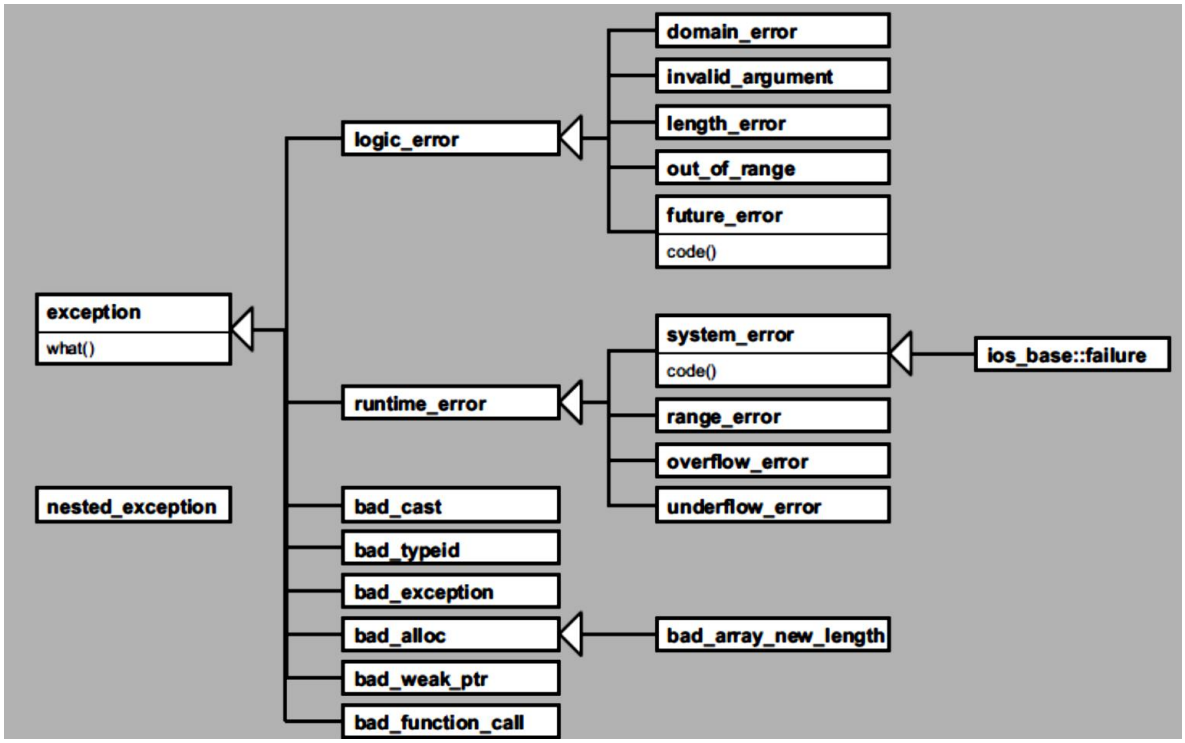
Il C++ mette a disposizione la seguente classe:

```
std::exception
```

che incapsula una stringa, che viene inizializzata nel costruttore, a cui si accede al suo contenuto con la funzione membro `what()`.
Un'eccezione serve per documentare un errore in fase di debug, non per fornire informazioni all'user.

La classe `std::exception` ha le seguenti sottoclassi:

- `std::logic_error`: indica incoerenza del codice
  - `domain_error`
  - `invalid_argument`
  - `length_error`
  - `out_of_range`
- `std::runtime_error`: indica situazioni inattese
  - `overflow_error`
  - `range_error`
  - `underflow_error` (si tenta di fare una *pop* su una pila già vuota)
  - `system_error` (si tenta di creare un thread, ma ce ne sono troppi)



Le API del sistema operativo non gestiscono eccezioni e si limitano a ribaltarle sul chiamante. Per questo è necessario inserire un codice come descrittore.

Le eccezioni sono incluse nella librerie:

```
#include <exception>
```

Le librerie standard del C++ si chiamano senza il ".h" finale (per distinguerli da quelli del C) mentre gli *header file* del programmatore sono chiamati con il ".h" e le doppie virgolette di delimitazione.
Per le sottoclassi è necessario includere le seguenti librerie:

La sintassi è la seguente:

```
int retry = 2;
while (retry) {
    try {
           //azioni varie
           break; //termina
    } catch (exception& e) {
           ReleaseExtraRes();
           retry--;
           if (!retry)
                  throw;
    }
}
```

Il comando `throw` all'interno di una clausola `catch{}` comporta l'inoltro dell'eccezione al livello superiore.

3. Registrare un messaggio

Si usa un meccanismo opportunamente predisposto per la registrazione degli errori (log di sistema). Esempio è il flusso standard di errore `std::cerr`.
Spesso si scrive nel log il messaggio dell'ecceazione stessa, col metodo `what()`. Essendo una notifica e non una gestione, l'eccezione dev'essere propagata tramite lo *statement* `throw`.

La sintassi è la seguente:

```
try {
    ...
} catch (exception& e) {
    logger::log(e.what());
    throw;
}
```

Cosa non fare

Assolutamente da evitare è scrivere un blocco `catch{}` che non esegua nessuna strategia di riallineamento e lascia proseguire un programma la cui esecuzione è parzialmente fallita.
Stampare un messaggio di errore e poi continuare è un errore grave.

Contrazione dello stack

Mentre in Java esiste il blocco `finally` (qualunque clausola `catch` è stata attivata, esegui comunque il blocco `finally`), in C++ esso non esiste: si utilizza la contrazione dello stack e la relativa chiamata dei distruttori.
Il **paradigma RAII (Resource Acquisition Is Initialization)** consiste nell'acquisire le risorse alla creazione dell'oggetto (costruttore) che le utilizza e nel rilasciarle quando questo viene liberato (distruttore). Questo paradigma permette di garantire che le risorse acquisite siano sempre liberate opportunamente. Può essere usato per eseguire azioni anche in presenza di eccezioni, ma è necessario fare attenzioni a non sollevarne altre.
Tale strategia sarà molto utilizzata per la concorrenza (acquisizione e rilascio di lock).

Composizione di oggetti

Nella programmazione ad oggetti, gli oggetti possono essere composti tra di loro. Tale composizione impatta sulla duplicazione: copia e movimento. Ciascuna di esse si suddivide in creazione e assegnazione. Sono concetti nuovi (nessuna equivalenza diretta in Java), ma permettono di capire a fondo C++.

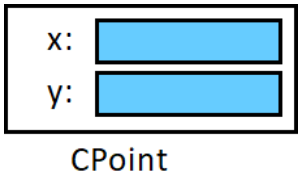
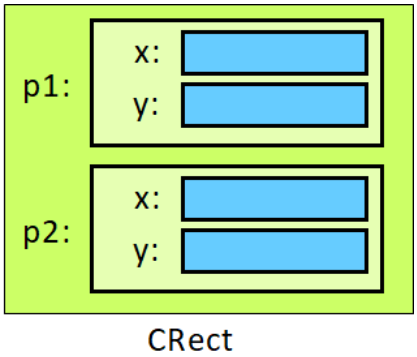
Oggetti composti

In C++ la **composizione di oggetti** può essere ottenuta in due modi diversi:

- parte integrante della struttura dati
- facendo riferimento tramite puntatore

Se l'oggetto contenuto è parte della struttura dati, esso è legato alla vita del contenitore, mentre utilizzando puntatori la vita del contenuto può essere slegato dal ciclo di vita del contenitore.
In entrambi i casi è il costruttore che si occupa di inizializzare opportunamente tali oggetti. Per utilizzare queste due modalità si utilizzano diverse sintassi e varia anche la disposizione in memoria.

Esempio: contenere un oggetto

> Definire un oggetto `CRect` che contiene direttamente due istanze della classe `CPoint`.
>
> ```
> class CPoint {
> public:
>     CPoint(int x, int y);
> private:
>     int x, y;
> };
>
> class CRect {
> public:
>     CRect(int x1, int y1, int x2, int y2);
> private:
>     CPoint p1, p2;
> };
> ```
>
> 
>
>
> 
>
>
> Mentre per il costruttore si possono utilizzare l'inizializzazione esplicita, per il rettangolo è necessario che esistano già i punti `p1` e `p2`. La sintassi per assumere l'esistenza di alcuni oggetti implica l'utilizzo di ":" seguita dal nome della variabile locale che deve già essere assegnata.
>
> ```
> #include "geom1.h"
> CPoint::CPoint(int x, int y) {
>     this->x = x;
>     this->y = y;
> }
> CRect::CRect(int x1, int y1, int x2, int y2) : p1(x1, y1), p2(x2, y2) {}
> ```
>
> Le parte non citate sono costruite dai loro costruttori di default (senza parametri)

In Java un **containment** (un oggetto che contiene un altro oggetto) è realizzabile solo tramite puntatori, mentre in C++ è possibile utilizzare lo stesso spazio in memoria.
In C++ il costruttore presuppone l'istanziazione degli oggetti parte (oggetti contenuti).

Esempio: contenere un puntatore ad un oggetto

> Riprendere l'esempio precedente, ma utilizzare gli oggetti contenuti come puntatori, invece che come oggetti parte.

Mentre la copia non intacca alcun dato precedentemente presente, l'assegnazione deve occuparsi preventivamente di liberare l'oggetto precedentemente presente.

Esempio:

Data la classe `CPoint`, eseguire operazioni di assegnazioni e di copia distinguendo opportunamente tra le due:

```
class CPoint {
    int x, y;
public:
    CPoint(int x, int y);
};


CPoint p1(10, 5);
CPoint p2(3, 3);
f(p1, p2); //costruzione di copia
```

Dove la funzione `f()` è la seguente:

```
void f(CPoint A, CPoint B) {
    A = B; //assegnazione

    CPoint C(A);      //costruzione di copia
    CPoint D = B;     //costruzione di di copia!
}
```

Non è la presenza dell'uguale (=) a discriminare tra copia ed assegnazione, ma solo se la destinazione esisteva già preventivamente.

In Java l'analogo del codice dato non sarebbe neanche compilabile.

Il compilatore C++ si occupa autonomamente, se non presente, di creare un costruttore di copia e una operatore di assegnazione.

Se non presenti, il compilatore crea:

- **costruttore di copia**: costruttore di un oggetto che accetta come parametro un riferimento ad un oggetto dello stesso tipo
- **operatore di assegnazione**: *operator overloading* dell'operatore "=" che accetta come "argomento" un oggetto dello stesso tipo

Copiare i puntatori

Di base, il compilatore ricopia il contenuto di tutte le variabili istanza dell'oggetto sorgente nell'oggetto destinazione. Questo funzionamento opera per tutte le variabili, eccetto che per i puntatori. Tale comportamento è problematico per i puntatori in virtù delle ambiguità dei puntatori precedentemente descritte.

Per questo motivo è preferibile evitare di avere un puntatore in una classe. Inoltre gestire un puntatore nativo richiede un grande ammontare di codice. La soluzione sono gli **Smart Pointer**: un gruppo di oggetti diversi che implementano correttamente tutte le funzionalità e lasciano allo sviluppatore scegliere solo la semantica. Sintatticamente sono utilizzati come dei puntatori normali, ma la loro copia/assegnazione richiama il costruttore di copia o il suo operatore di assegnazione che memorizzano le informazioni necessarie a gestirle (es. per puntatori condivisi, il numero di oggetti che hanno puntatori a quella zona di memoria).

```
    CBuffer(const CBuffer& source) {
        this->size = source.size;
        this->ptr = new char[size];
        memcpy(this->ptr, source.ptr, size);
    }
};
```

Utilizzando la `memcpy()` si ha un'esecuzione più efficace rispetto ad un `for{}` (si evitano le condizioni di copia e l'inizializzazione).

Operatore di assegnazione

L'**operatore di assegnazione** è la funzione membro `operator=` che indica al compilatore come comportarsi quando si assegna un nuovo valore ad un oggetto già esistente (che ha risorse da liberare).

In C++ è possibile dare una semantica agli operatori classici (+, -, =, …) per tipi non primitivi, specificando la *keyword*:

```
operator <operatorType>
```

così che il compilatore sappia quale metodo chiamare quando viene applicato tale operatore per tipi non classici.

Sfruttando tale caratteristica del C++, la sintassi dell'operatore di assegnazione, è la seguente:

```
MyClass& operator= (const MyClass& source);
```

L'*operatore di assegnazione,* quando invocato, deve distruggere il precedente contenuto dell'oggetto destinazione prima di inizializzarlo con la copia dell'oggetto sorgente. Rispetto al *costruttore di copia*, l'*operatore di assegnazione* ha la responsabilità di ripulire la zona di memoria in cui copiare. Esso è, a livello logico, l'unione di costruttore di copia e del distruttore (lo sviluppatore deve implementarlo correttamente).
L'operatore di assegnazione ritorna poi un riferimento all'oggetto destinazione tramite un

```
return *this;
```

Esempio:

L'operatore di assegnazione, per la classe `CBuffer` precedentemente definita, è implementato dalla seguente funzione:

```
CBuffer& operator=(const CBuffer& source) {
    if (this != &source) {
        delete[] this->ptr;
        this->ptr = null;
        this->size = source.size;
        this->ptr = new char[size];
        memcpy(this->ptr, source.ptr, size);
    }
    return *this;
}
```

È fondamentale effettuare il controllo iniziale sull'identità:

```
if (this != &source)
```

per evitare che sia lo stesso oggetto (non necessario alcuna copia). Inoltre è necessario assegnare a `NULL` il puntatore dell'oggetto corrente:

Nella dichiarazione di una funzione non è obbligatorio specificare il nome dei parametri formali, mentre è invece necessario nella definizione, per farvi riferimento, identificandoli in modo univoco.

Assegnazione e risorse

L'operatore di assegnazione deve rilasciare tutte le risorse possedute per non creare *leakage* e si sovrappone al distruttore. Bisogna quindi duplicare il codice e mantenerlo uguale anche in evoluzioni successive.

La **regola dei tre** sostiene che, se una classe dispone di una qualunque di queste funzioni membro, occorre implementare le altre due:

- costruttore di copia
- operatore di assegnazione
- distruttore

È buona abitudine eliminare il prototipo di distruttore che Visual Studio propone alla creazione del codice di una classe. Se non si hanno particolari risorse da gestire, è meglio lasciare al compilatore il compito di implementare il distruttore di *default*.

Copia e movimento

Il *movimento* non ha alcuna analogia con Java e fu introdotta solo con C++11. Copiare un oggetto potrebbe essere un'operazione dispendiosa, si preferisce quindi utilizzare tale concetto di *movimento*, cioè il riutilizzo delle parti di un oggetto che sta per essere distrutto.

Movimento

Il **movimento** è un'operazione che implica lo "svuotare" un oggetto (che sta per essere distrutto) del suo contenuto e "travasarlo" in un altro oggetto.
Tale riutilizzo dei blocchi di memoria permette di evitare il rischio di *memory overflow* (quando entrambe le copie sono presenti in memoria) ed è una situazione frequentissima nel codice (es. `return` di una variabile locale).

Non è possibile scegliere quando applicare il movimento, ma è il compilatore sceglie in autonomia quando utilizzarlo. I candidati al movimento sono i seguenti:

- variabili locali al termine del blocco in cui sono state definite
- risultati delle espressioni temporanee
- oggetti anonimi costruiti a partire dal tipo

In generale è movibile tutto ciò che non ha un nome e può comparire solo a destra di "=" nelle assegnazioni (`RVALUE`).

Chiamando il distruttore sulla `source` (vecchio oggetto) è importante che il `MoveConstructor()` "smonti" bene le parti per evitare deallocazioni non volute di memoria.

Utilità del movimento

Il vantaggio del movimento è la riduzione del costo del passaggio di parametri per valore. Questo sia per i parametri in ingresso, sia per i dati restituiti in output.

Esempio:

> Anche la funzione `string`, come tutte le classi standard di libreria, implementa un costruttore di movimento per permettere chiamate efficaci a funzioni, anche in presenza di stringhe molto lunghe.
>
> ```
> string f() {
>     string x("…");    // Dichiara una stringa
>     string a(x);      // Non si può muovere, x: LVALUE, si COPIA x in a
>     string b(a + x);  // Il risultato di a+x viene mosso in b
>     string c(funzioneCheRitornaString()); //il risultato è mosso in c
>     return c;         // c viene mosso nel risultato
> }
> ```

La **classe `string`** gestisce, al suo interno, un buffer di caratteri di dimensione arbitraria. Si occupa lei di gestire l'eventuale riallocazione. Fornisce metodi specifici di comparazione, append e concatenazione.
Il compilatore permette di assegnare ad un oggetto `string`, la notazione standard del C (delimitata da virgolette) chiamando in automatico il costruttore con il parametro specificato. Si ha quindi totale sovrapposizione (nell'esempio precedente) tra oggetto `string` e notazione `"…"`.
La stringhe introducono un'astrazione molto comoda per il programmatore, ma rischiano di essere lunghe e quindi onerose per gli operatori di copia. Per questo motivo è stato necessario introdurre il costruttore di movimento (per questioni di efficienza) dal C++11. Nelle versioni precedenti l'uso delle `string` (e tutte le librerie standard) non era considerato un buon stile di programmazione, ora hanno reso inutili l'uso dei puntatori nativi.

Operatore di movimento

Esistono alcuni oggetti che non è possibile copiare, ma per i quali è possibile effettuarne il movimento.
Per esempio lo *smart pointer unico* (`std::unique_ptr<>`) non vuole interferenze nel suo ciclo di vita, cioè il suo possesso non dev'essere condiviso. È però possibile cedere completamente tutto l'oggetto ad un altro attore.
L'utilizzo del movimento rende possibile creare una funzione che ritorna un oggetto di tipo `unique_ptr<>`, senza violarne le regole.

Se un oggetto contiene:

- valori elementari: movimento equivale alla copia
- risorse esterne: il movimento costituisce un vantaggio rispetto alla copia

- omettere (2): in caso di eccezione successiva, il distruttore potrebbe rilasciare due volte la memoria
- rischi di (3): se l'oggetto è grosso, potrebbe non esserci la memoria richiesta e `new[]` lancia un'eccezione lasciando un oggetto corrotto

Per tali motivazioni si è introdotto il **paradigma *Copy&Swap***. Si introduce una funzione `swap()` che non appartiene alla classe, ma è dichiarata `friend`, quindi può accedere agli attributi `private` della classe (gli si danno i permessi). Tale funzione `swap()` scambia il contenuto delle risorse tra le due istanze. Viene sfruttata la funzione standard `std::swap()` che scambia i riferimenti elementari contenuti negli oggetti.

```
friend void swap(intArray& a, intArray& b) {
    std::swap(a.mSize, b.mSize);
    std::swap(a.mArray, b.mArray);
}
```

L'operatore di assegnazione si implementa con un parametro di tipo valore (`that`). La logica dell'operatore di copia provvederà a fare un duplicato nel parametro `that`. In caso di eccezione, `*this` non viene modificato (l'eccezione avviene sul tentativo di copia, prima dell'invocazione di `swap()`). Questa coppia (copia+swap) garantisce la correttezza dell'operazione, assumendo la semantica di copia+distruzione oppure del movimento, a seconda della situazione in cui ci si trova.

```
intArray& operator=(intArray that) {
    //that è passato per valore, copiato o mosso a seconda del contesto in cui è usato
    swap(*this, that);
    return *this;
}

//costruttore di movimento
intArray(intArray&& that) : mSize(0), mArray(NULL) {
    swap(*this, that);
}
```

Funzione `move()`

Una funzione molto utile, definita nel file `<utility>`, è la funzione `move()`:

```
type&& std::move(T&& t);
```

Essa costituisce un supporto per trasformare un oggetto generico in un riferimento *RValue*, così da poterlo utilizzare nella costruzione o assegnazione per movimento.
La funzione `move()` forza l'oggetto passato come parametro ad essere considerato come riferimento *RValue* rendendolo indisponibile per usi ulteriori (viene eseguito uno `static_cast<T&&>(t)`).
Il *casting* standard del C (anteporre il tipo di destinazione tra parentesi) è deprecato, perché non permette di specificare la semantica.

Esempio:

Definita una stringa e un vettore di stringhe, il metodo `move()` inserisce la stringa nel vettore eliminando contestualmente il contenuto di `str`:

```
std::string str("hello");
std::vector<std::string> v;

v.push_back(str);             // v = ["hello"], str = "hello"
v.push_back(std::move(str));  // v = ["hello", "hello"], str = ""
```

Ereditarietà multipla

In C++, a differenza del Java, supporta l'**ereditarietà multipla**, cioè una classe può ereditare da una o più classi. Si può specializzare, in un solo modello, più concetti.
Il Java supporta un'emulazione dell'ereditarietà multipla tramite l'utilizzo delle *interfacce*. Implementare un'interfaccia significa implementare i suoi metodi, quindi la classe può essere castabile all'interfaccia.
È molto pericoloso utilizzare l'ereditarietà multipla perché può provocare delle confusioni. Essa è utilizzata negli *stream* (`iostream` implementa tanto `istream` quanto `ostream`).
Mentre in Java ogni oggetto eredita da `Object`, in C++ ogni classe è radice del suo albero di ereditarietà se non specifica esplicite specializzazioni. Non esiste quindi la *keyword* `super`, perché potrebbe non esistere una classe genitrice oppure potrebbero esisterne molteplici.

Esempio:

Data la classe genitrice `Base`:

```
class Base {
public:
    Base();
    void baz();
    ~Base() {}
};
```

la classe derivata `Der`, la specifica:

```
class Der : public Base {
public:
    Der():Base() {
        Base::baz()
    }
    ~Der() {}
};
```

è quindi possibile utilizzare il metodo del genitore, dall'oggetto figlio:

```
Der *obj = new Der();
obj->baz();
delete obj;
```

In C++ non esiste la parola chiave `super`, non funzionerebbe con ereditarietà multipla

Polimorfismo

Il **polimorfismo** è la capacità di un'espressione, descritta dalla classe `B`, di asumere valori di qualunque tipo descritto da `A`, sottoclasse di `B`.
Se una classe `A` estende in modo pubblico la classe `B` (contenente puntatori), è lecito assegnare ad una variabile `v` (di tipo `B*`) un puntatore ad un oggetto di tipo `A`:

```
A <<is_a>> B
```

L'ereditarietà implica la relazione *is a* da figlio verso padre, ma non è valido il viceversa (il padre *is not a* figlio).

È possibile che il figlio effettui un *override* del metodo del padre, cioè ne modifica l'implementazione a seconda di esigenze specifiche.

La classe `Base` ha il metodo `bar()` astratto (non ha un'implementazione). In Java, al contrario, i metodi non polimorfici vanno esplicitamente dichiarati `final`.

Mentre Java rende tutti i metodi come polimorfici di default (ciascuna classe figlia può sovrascrivere qualunque metodo delle classi genitrici a meno dell'uso della *keyword* `final`), in C++ si ha di *default* il comportamento contrario: tutto è non polimorfico, a meno di esplicita segnalazione.
Questo perché il polimorfismo è costoso in termini di performance: è necessario esplorare *runtime* l'albero dell'ereditarietà per trovare la sovrascrizione di livello più specifico.
In generale il C++ segue la linea guida di far pagare (a livello computazionale) solo ciò che serve realmente. Questo ha lo svantaggio di limitare al minimo le funzionalità offerte.

Distruttori virtuali

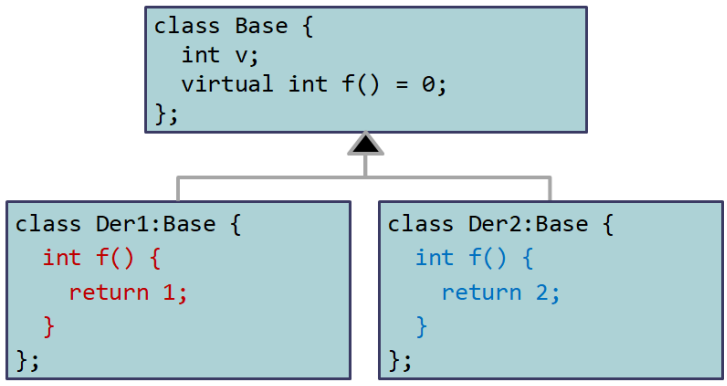
Anche i distruttori devono essere specificati come virtuali, se la classe contiene anche solo un metodo virtuale. Se non sono presenti variabili puntatori, non è necessario specificare `virtual` il distruttore.
Anche i distruttori sono non polimorfici per default.
Se una classe con metodi virtuali non ha un distruttore virtuale, è possibile che venga chiamato il distruttore sbagliato.

Esempio:

Dato la super-classe `Base`, dell'esempio precedente:

```
class Base {
public:
    Base() {}
    virtual void foo() { ... }
    virtual void bar() = 0;
    virtual ~Base() { ... }
};
```

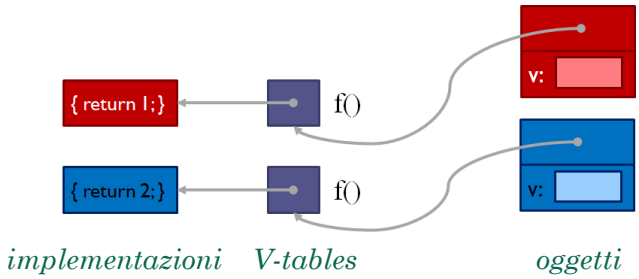
Se si effettuano due specializzazioni (`Der1` e `Der2`) di `Base`:



L'esecuzione del seguente codice

```
Base* b1 = new Der1();
Base* b2 = new Der2();

printf("%d\n", b1->f());     //1
printf("%d\n", b2->f());     //2
```

L'invocazione di `f()` su `b1` stampa `1`, mentre lo stesso metodo su `b2` stampa `2`. Il compilatore invoca il metodo corretto analizzando le specifiche *V-Table*.



La *V-Table* permette di supportare l'ereditarietà multipla perché la classe figlia ha tante V-Table quante sono le classi genitrici, più una per gli eventuali metodi virtuali che essa stessa definisce.
Per oggetti ad eredità semplice si utilizzano le prime *N* entries, della V-Table, per i metodi virtuali della superclasse e le restanti per i metodi virtuali propri della classe figlia. In caso di ereditarietà multplia non si può introdurre aleatorietà cambiando l'indice delle varie funzioni a seconda della classe a cui ci si riferisce. Il compilatore gestisce quindi una serie di V-Table separate per non introdurre incomprensioni.

Esempio:

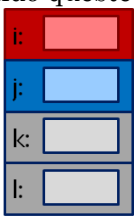
Date due superclassi:

```
class CBase1{
      int i;
};
```


```
class CBase2 {
      int j;
};
```

la classe che le eredita entrambe, essendo queste NON virtuali, possiede una sola *V-Table*:

```
class CDer :
    public CBase1,
    public CBase2
{
    int k, l;
};
```



L'utilizzo di V-Table multiple ha un grosso impatto sull'operatore di *casting*.

Conversione tra tipi

Il **casting** è un meccanismo che il compilatore mette a disposizione per alterare la gestione dei tipi. Permette di convertire il tipo di un dato, anche a costo di perdere delle informazioni. È possibile effettuare *casting* tra qualunque tipi di dati, il problema è la semantica del valore così ottenuto.

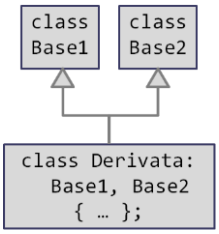
Il C++ supporta il **type-cast** in stile C, cioè utilizzando il tipo destinazione tra parentesi. Analogamente all'utilizzo dei puntatori, anche il casting C-like è ambiguo, specialmente presenza di ereditarietà ed ereditarietà virtuale.
Il C++ fornisce operatori più sicuri:

```
static_cast<T>(p);
dynamic_cast<T>(p);
const_cast<T>(p);
reinterpret_cast<T>(p);
```

Sono operatori generici, quindi tra parentesi acute richiedono il tipo di destinazione, mentre tra parentesi tonde è passato il valore da convertire. Il tipo di origine è dedotto automenticamente dal valore passato come parametro.

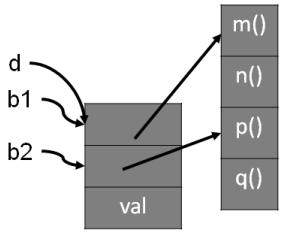
Esempio:

Dato il seguente schema di ereditarietà:




```
Derivata *d = new Derivata();
Base1 *b1;
Base2 *b2;

b1 = static_cast<Base1 *>d;
b2 = static_cast<Base2 *>d;
```

in memoria il puntatore `d` indica la seguente struttura dati (composta dalle V-Table di `Base1` e di `Base2`):



L'allocazione dei metodi è contigua, ma considerata a chunk distinti (`b1` accede solo ai primi due metodi). Il cast esplicito (stile C) non aggiorna il puntatore alle V-Table provocando quindi la perdita di informazioni.

Anche se sia `d` che `b1` puntano alla stessa cella (`d==b1`), non è possibile effettuare un ulteriore cast di `b1` a `b2`:

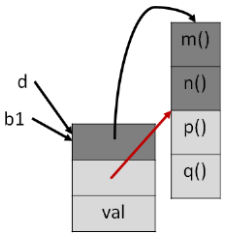
```
b2 = static_cast<Base2*>(b1);
```

perché sono due classi *unrelated*: tra questi due tipi non c'è nè un asse ereditario, né un importatore e neppure un esportatore.

Il tentativo di casting al contrario provoca delle grosse problematiche:

```
d = static_cast<Derivata*>b1;
```

Passare da specifico a generico va sempre bene, mentre il contrario, benché ammesso dal compilatore, delega al compilatore la responsabilità di farlo a dovere. Se `b1` non è effettivamente di tipo `Derivata`, si tenta di accedere a funzioni non sue, con conseguenze impredicibili.



È possibile cercare di fare **downcasting** (scendere lungo l'asse ereditario), ma è una grossa responsabilità del programmatore: il compilatore non è in grado di accorgersene.

2. Dynamic cast

L'**operatore `dynamic_cast<T>(p)`** si appoggia al compilatore e al *runtime*: delega il controllo se il *casting* sia fattibile. Perché sia fattibile in pratica è necessario aggiungere, oltre alla V-Table, anche l'informazione della classe che l'oggetto implementa.

In Java tale proprietà, **reflection**, è automatica grazie al metodo `getClass()` che ritorna la classe reale dalla quale l'oggetto è stato implementato. In C++ questo non è implementato automaticamente, per diminuire lo spazio occupato da un oggetto (evitare *fat objects*).

Gli sviluppatori C++ sono liberi di scegliere come implementare il `dynamic_cast<>`. La tecnica più diffusa è il **RTTI (RunTime Type Identification)**: si aggiunge un tag (una sorta di numero hash)

4. Const cast

L'**operatore `const_cast<T>(p)`** ha dei limiti e per questo va usato con molto giudizio. Spesso il parametro passato ad un *caster* è di tipo `const`, quindi non modificabile, ma il *const cast* permette di ignorare tale specifica `const` e di modificarlo lo stesso.
È molto pericoloso perché si modifica un dato che è stato fornito perché rimanesse immutato. Per esempio il codice *embedded* è ricco di `const`, perché scritto nella EPROM.

Oggetti funzionali

In C++ esiste il **funtore (oggetto funzionale)**: un'istanza di una qualsiasi classe che abbia ridefinito la seguente funzione membro:

```
operator() (<arguments>) {<implementation>}
```

Esso ridefinisce il comportamento dell'operatore "()" che corrisponde alla sua chiamata. Il codice specificato è quindi il codice della funzione che il *funtore* rappresenta.
Questo permette di utilizzare tale *funtore* come una normale funzione, quando lui in realtà è un oggetto (istanza di una classe).

Esempio:

Una classe che definisce la funzione `operator()`

```
class FC {
public:
    int operator() (int v) {
        return v * 2;
    }
};
```

È possibile istanziare un oggetto di tale classe, quindi un funtore:

```
FC fc;
int i = fc(5); // i vale 10
i = fc(2);     // i vale 4
```

Tale funtore somiglia ad una funzione, ma in realtà è un oggetto.

In una classe è possibile definire **molteplici definizioni** di `operator()`, purché essi differiscano per i parametri (numero e tipo) che richiedano.
Un *oggetto funzionale* può contenere variabili membro e queste possono essere utilizzate all'interno delle funzioni `operator()` per tenere traccia di uno **stato**.
Per questo motivo è possibile memorizzare uno stato che influisce sull'esecuzione del codice e fa cadere così la definizione di funzione strettamente matematica (ad ogni input corrisponde un output indipendentemente da fattori esterni). Per questo motivo tali oggetto sono detti *funzionali* (non propriamente funzioni nel senso matematico del termine).

Esempio:

Data la seguente classe `Accumulatore`:

```
class Accumulatore {
    int totale;
public:
    Accumulatore() :totale(0) {}
    int operator()(int v) {
        totale += v;
        return v;
    }
    int totale() { return totale; }
};
```

Essa può essere utilizzata, memorizzandone lo stato, come segue:

```
void main() {
    Accumulatore a;
    for (int i = 0; i<10; i++)
        a(i);
```

Per ottenere lo stesso comportamento, è possibile definire una **funzione lambda**: permettono di dichiarare una funzione (senza assegnargli un nome) dichiarandole esattamente nel punto in cui devono essere utilizzate. Non ha validità al di fuori di quello specifico contesto.
Esse non richiedono di definirne un nome ed evitano di definire una funzione esplicita. Le *funzioni labda* non hanno *contesto*, se non specificato appositamente nelle parentesi quadre.

Esempio:

L'esempio precedente può anche essere implementato mediente l'utilizzo di una *funzione lambda*:

```
int main() {
    std::vector<int> v;
    //...
    std::for_each(v.begin(), v.end(),
        [](int i) { std::cout << i << " "; }
    );
}
```

Tale soluzione permette di definire direttamente *inline* il codice necessario, senza dover definire la funzione esterna.

Le funzioni lambda restituiscono un valore mediante una sola istruzione `return`. Il suo tipo può essere dedotto esplicitamente dal compilatore, altrimenti occorre esplicitarlo nella definizione.

Esempio:

La seguente *funzione lambda* ritorna, se possibile, il risultato di una frazione; in caso contrario un *NaN (Not a Number)*:

```
[](int num, int den) -> double {
    if (den == 0)
        return std::NaN;
    else
        return (double)num / den;
}
```

Come una nurmale funzione possiede quindi diversi operatori `return` in onore dei vari flussi di esecuzione.

Catturare delle variabili

Le funzioni lambda sono riconoscibili dall'uso di "`[]`", poi "`()`" e infine "`{}`".
Le parentesi quadre introducono la notazione lambda e al loro interno è possibile specificare le variabili il cui valore (o il cui riferimento) si vuole rendere disponibile nella funzione.
Tali valori sono catturati (per valore o per riferimento) dalla funzione nell'istante in cui la *funzione lambda* è specificata e mantengono tale valore anche se le variabili originali variano successivamente.
Sono possibili due tipi di cattura, specificati per ogni variabile:

- cattura per valore `[x,y]`: viene creata una copia delle variabili e la λ-function potrà essere chiamata anche quando tali variabili saranno uscite dal loro *scope*
- cattura per riferimento `[&x,&y]`: eventuali cambiamenti influenzano l'originale, ma bisogna prestare attenzione al rischio di riferimenti pendenti

Il **operator overloading** è definito dalla *keyword* `operator`, seguito dall'operatore da ridefinire, dai parametri accettabili e dalla sua implementazione (tra parentesi graffe).

Esempio:

Per ridefinire il comportamento dell'operatore "+", sulla classe `Frazione`, si definisce il seguente codice:

```
class Frazione {
public:
    Frazione(int num = 0, int den = 1) {
        this->num = num;
        this->den = den;
    }
    Frazione operator + (const Frazione &r) {
        Frazione   temp;
        temp.den = this->den * r.den;
        temp.num = r.den * this->num + this->den * r.num;
        return temp;
    }
private:
    int num;
    int den;
};
```

La classe `Frazione` contiene due variabili intere `private`: numeratore (`num`) e denominatore (`den`).
L'operatore "+" viene ridefinito in modo da somare due istanze di tale classe e ritornare un nuovo oggetto dello stesso tipo.
Tale somma è utilizzabile come segue:

```
Frazione a(1, 3);
Frazione b(3, 5);
Frazione c;
c = a + b;
```

dove l'istruzione `(a+b)` corrisponde in realtà al seguente codice:

```
Frazione temp;
temp.den = this->den *b.den;
temp.num = b.den * this->num + this->den * b.num;
return temp;
```

Oltre che i normali operatori `()` è possibile anche ridefinire il comportamento delle parentesi quadre ("`[]`"). Invece che ridefinire come un operatore, è possibile ridefinire come un vettore.
Questo permette di utilizzare qualunque tipo come indice di una struttura dati. Non è un vero array, ma si emula (sintatticamente) il comportamento di un array, invocando in realtà una funzione.