



Piattaforme di esecuzione: cenni introduttivi

Programmazione di Sistema
A.A. 2016-17

Argomenti

- Ambienti operativi
 - Windows
 - Linux
 - Android
- Interfacciarsi con il sistema operativo
- Gestione degli errori



Linguaggi e ambienti di sviluppo

- Linguaggio C e C++, con estensioni dello standard 2011 (C++11)
 - VisualStudio 2012+ (Windows)
 - g++ 4.8.x+ (Linux)
- Linguaggio Java
 - AndroidStudio



Interfacciarsi con il sistema operativo

- Fatta eccezione per i sistemi più elementari, l'esecuzione di un'applicazione avviene nel contesto di un S.O.
 - Che offre un insieme di servizi, funzionalità, convenzioni che ne permettono il funzionamento
- Per sfruttare tali servizi, un'applicazione deve conformarsi alle specifiche del S.O.
 - Sia a livello di codice sorgente
 - Che a livello di codice eseguibile



API - Application Programming Interface

- Definisce un insieme di funzioni e di strutture dati che vengono offerte, così come sono, al programmatore
 - Invocandole, ottiene l'accesso ai servizi offerti dal sistema sottostante
- Le API effettivamente offerte dai sistemi operativi (System Call) sono prevalentemente annigate in funzioni di libreria
 - Che ne astraggono l'utilizzo
 - Es.: C Standard Library



ABI - Application Binary Interface

- Definisce quale formato debba avere un prodotto software per essere compatibile con il S.O.
 - E a quali convenzioni debba sottostare
- Comprende aspetti quali
 - Convenzioni di chiamata e passaggio dei parametri
 - Uso dei registri del processore
 - Innalzamento di privilegio e invocazione del S.O.
 - Collegamento tra moduli e struttura dei file binari



ABI - Application Binary Interface

- È supportata dagli strumenti che compongono la toolchain
 - Compilatore
 - Linker
 - Debugger
 - Profiler
 - Inspector
- Può essere oggetto diretto della programmazione di sistema
 - Quando si utilizzano meccanismi quali il caricamento dinamico di moduli, l'emulazione delle piattaforme di esecuzione, ...



Interfacciarsi con il S.O.

- Per sviluppare un programma che si interfacci direttamente con il sistema operativo, occorre
 - Conoscere le API da chiamare
 - Le strutture dati coinvolte
 - Le convenzione generali definite dal S.O.



Accedere alle API

- Occorre dichiarare le funzioni e definire i tipi dei parametri
 - Attraverso l'inclusione di appositi file header
 - Nel caso di Windows, di solito equivale ad includere il file Windows.h
- In alcuni casi, può essere necessario collegare l'eseguibile con librerie specifiche
 - Ad esempio, in Linux, un programma concorrente deve essere collegato alla libreria PThread



Convenzioni

- Ogni S.O. mantiene, al proprio interno, un insieme di strutture dati che descrivono lo stato del sistema
 - Tali strutture NON sono esposte direttamente al programmatore
- Le API permettono di accedere in modo indiretto a tali strutture attraverso riferimenti opachi
 - Detti HANDLE in Windows
 - FileDescriptor in Linux



Gestione degli errori

- Le funzioni invocate possono avere successo o fallire
 - Occorre verificarne sempre l'esito
 - Il modo di farlo dipende dal S.O.



Gestire gli errori in Windows

- Per sapere se la chiamata di una API ha avuto successo o meno, occorre basarsi sul tipo di ritorno
 - Adottando una strategia opportuna
- **BOOL (int)**
 - Restituisce 0 in caso di fallimento, qualunque altro valore in caso di successo
- **HANDLE (void*)**
 - Restituisce 0 o -1 in caso di fallimento: consultare la documentazione online
- **PVOID (void*)**
 - NULL indica un fallimento
 - Un puntatore valido indica successo
- **LONG/DWORD (unsigned long)**
 - Dipende dal significato del valore ritornato: consultare la documentazione online

Gestire gli errori in Windows

- Appurato che si è verificato un errore, occorre capire quale sia
 - Si invoca la funzione DWORD GetLastError();
 - Poiché ulteriori chiamate potrebbero nascondere l'errore precedente, occorre invocarla il prima possibile
- Il valore ritornato è un numero a 32 bit
 - Per conoscerne il significato è possibile utilizzare la utility ErrorLookup di VisualStudio
 - O invocare la funzione FormatMessage(...)



Gestire gli errori in Linux

- I dettagli con cui una API indica il fallimento della richiesta sono variabili
 - Prevalentemente, un valore pari a -1 indica il fallimento
 - Occorre controllare la documentazione online
- Si accede al codice di errore ispezionando il contenuto della pseudo-variabile globale errno
 - `#define errno (*__errno_location ())`
- Come nel caso di Windows, occorre ispezionare il contenuto di tale variabile non appena si verifica l'errore
 - Ulteriori chiamate al sistema potrebbero sovrascriverla



Gestire gli errori in Linux

```
if (fsync (fd) == -1) {  
  
    // fprintf chiama altre system call,  
    // che sovrascrivono errno  
    fprintf (stderr, "fsync failed!\n");  
  
    if (errno == EIO) // NON VA BENE!!!  
        fprintf (stderr,  
                 "I/O error on %d!\n", fd);  
}
```



Uso di stringhe

- La codifica e gestione del testo è un argomento molto più complesso di quanto appaia a prima vista
 - La rappresentazione dei caratteri a 8 bit (char) è insufficiente per la maggior parte degli alfabeti

Codifica dei caratteri

- Il consorzio Unicode ha definito una rappresentazione standard che prevede alcuni milioni di simboli rappresentati da numeri interi
 - La loro codifica richiede almeno 21 bit
- Rappresentazioni possibili
 - UTF-32: ogni simbolo occupa 32 bit
 - UTF-16: un simbolo può occupare una o due parole da 16 bit
 - UTF-8: un simbolo può occupare da una a quattro parole di 8 bit



Codifica dei caratteri

- Le rappresentazioni a lunghezza variabile possono avere ordinamenti differenti
 - Big endian, little endian
- I primi 128 valori Unicode coincidono con la codifica ASCII
- Nei primi 65000 valori sono codificati la maggior parte dei sistemi di scrittura correnti
 - Per evitare la complessità di gestire stringhe a lunghezza variabile, spesso si adotta una codifica a 16 bit, ignorando caratteri rari



Caratteri in Windows

- Due tipi base
 - char: 8bit, codifica ASCII estesa
 - wchar_t: 16 bit, codifica UNICODE
 - Basic Multilingual Plan 0
- Tipo generico
 - TCHAR: definito nel file tchar.h
- Tipi derivati
 - LPSTR: sequenza terminata da \0 di char
 - LPWSTR: sequenza di wchar_t
 - LPTSTR: sequenza di TCHAR
 - Le versioni con «LPC-» indicano sequenze immutabili



Caratteri in Windows

- Tutte le API del sistema che prevedono l'uso di caratteri e stringhe sono offerte in due versioni
 - ASCII, usa suffisso "-A"
 - WCHAR, usa suffisso "-W"
- Si invoca solo la versione generica, senza suffisso

Esempio

```
BOOL CreateDirectoryA(  
    LPCSTR lpPathName,  
    LPSECURITY_ATTRIBUTES lpSecAtt);  
BOOL CreateDirectoryW(  
    LPCWSTR lpPathName,  
    LPSECURITY_ATTRIBUTES lpSecAtt);  
#ifdef UNICODE  
#define CreateDirectory CreateDirectoryW  
#else  
#define CreateDirectory CreateDirectoryA  
#endif // !UNICODE
```



Caratteri in Linux

- Per default, il compilatore GCC codifica eventuali caratteri non-ASCII con la codifica UTF-8
 - Poiché la stringa che ne deriva termina correttamente con \0, questo non crea problemi al kernel
- La codifica UTF-8 può creare problemi in caso di algoritmi semplicistici
 - La funzione *strlen(str)*, ad esempio, non indica più il numero di caratteri effettivamente presenti, ma solo il numero di byte non nulli
 - Si determina il numero di caratteri con la funzione *mbstowcs(NULL, str, 0)*
(MultiByteStringTOWideCharacterString)



Caratteri in Linux

- Il tipo `wchar_t` esiste, ma ha lunghezza pari a 4 byte
 - Può ospitare qualsiasi carattere Unicode
 - Sequenze costanti di `wchar_t` sono precedute da «l »

```
wchar_t* s= L"Ω€®™åßðƒ∞Δªøπ°¬Σμ"
```



Il modello di esecuzione

Programmazione di Sistema
A.A. 2017-18

Argomenti

- Modello di esecuzione di un programma
- Implementazione del modello
- Preparazione ed esecuzione di un processo



Modello di esecuzione di un programma

- Ogni linguaggio di programmazione propone uno specifico modello di esecuzione
 - Insieme di comportamenti attuati dall'elaboratore a fronte dei costrutti di alto livello del linguaggio
- Tale modello per lo più non corrisponde a quello di un dispositivo reale
 - Occorre introdurre uno strato di adattamento che implementi il modello nei termini offerti dal dispositivo soggiacente

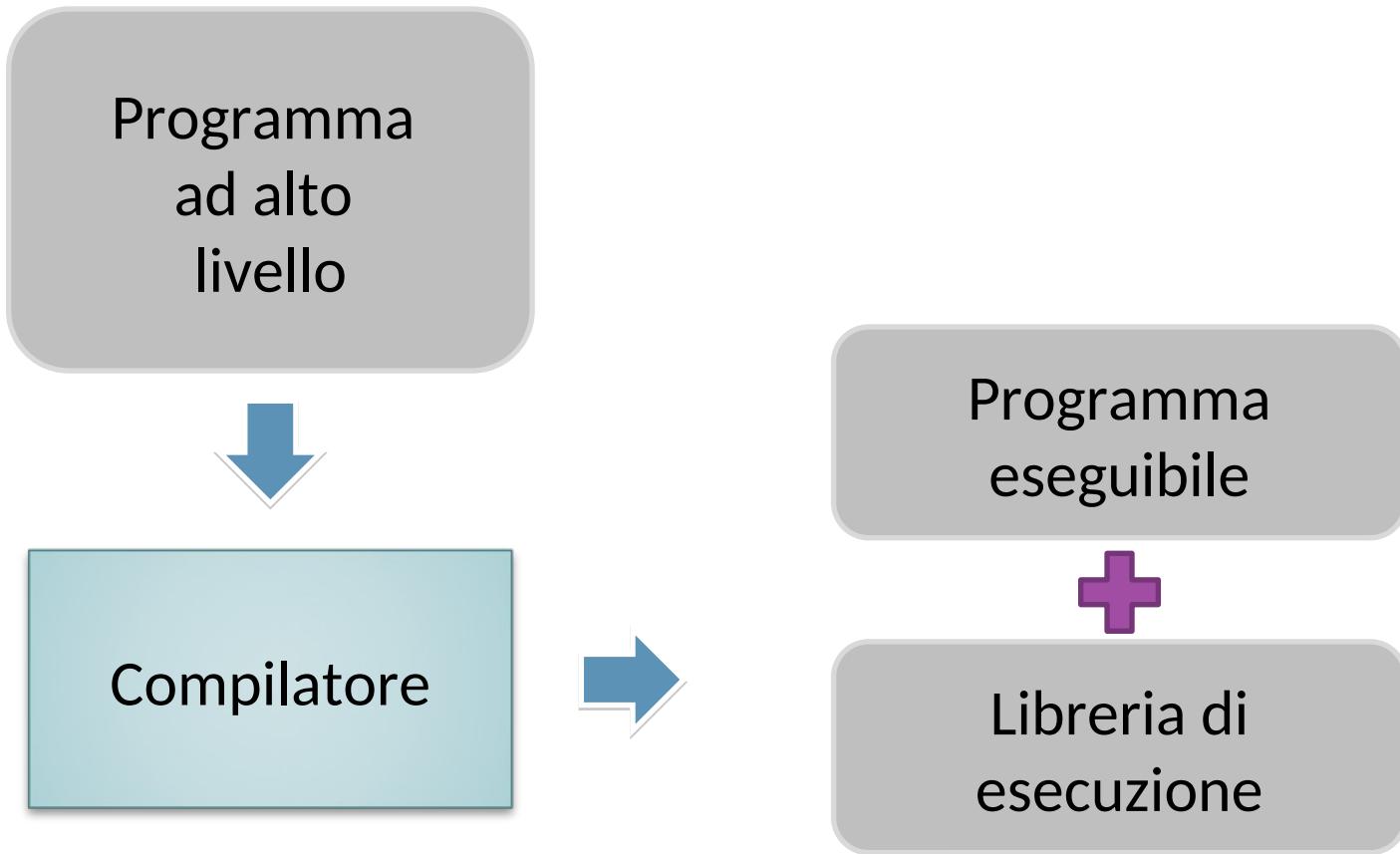


Livelli di astrazione

- Il programma originale viene trasformato in un nuovo programma dotato di un modello di esecuzione più semplice
 - Che può essere eseguito da una macchina fisica (CPU) o subire un'ulteriore trasformazione (esecuzione virtuale)
- Il compilatore ha il compito di eseguire tale traduzione
 - In parte, riscrivendo il linguaggio in istruzioni più semplici (codice macchina)
 - In parte, avvalendosi di una libreria di esecuzione



Livelli di astrazione



Librerie di esecuzione

- Offrono agli applicativi meccanismi di base per il loro funzionamento
 - Supportano le astrazioni del linguaggio di programmazione
 - Forniscono un'interfaccia uniforme tra i diversi S.O. per le funzioni ad essi demandate
- Costituite da due tipi di funzioni
 - Alcune, invisibili al programmatore, sono inserite in fase di compilazione per supportare l'esecuzione (es.: controllo dello stack)
 - Altre offrono funzionalità standard, gestendo opportune strutture dati ausiliarie e/o richiedendo al S.O. quelle non altrimenti realizzabili (es.: malloc, fopen, ...)



Modello di esecuzione nei linguaggi C e C++

- I programmi sono pensati come se fossero gli unici utilizzatori di un elaboratore, completamente dedicato loro (isolamento)
 - Le eventuali interazioni si riducono al più all'uso di risorse persistenti come il file system
- Un programma C/C++ assume di poter accedere a qualsiasi indirizzo di memoria
 - All'interno del quale può leggere o scrivere dati o dal quale può eseguire codice macchina



Esecuzione sequenziale, senza limitazioni

- Un programma è formato da un insieme di istruzioni eseguite, una per volta, nell'ordine indicato dal programmatore
 - Non ci sono limiti sulle istruzioni da eseguire, sul tempo richiesto e sulla memoria necessaria



Flusso di esecuzione

- Il programma è costituito da un flusso di esecuzione il cui punto di partenza è predefinito
 - La funzione main() nel linguaggio C
 - I costruttori delle variabili globali in C++
- Una struttura a pila, lo stack, permette di gestire chiamate annidate tra funzioni
 - Supporta la ricorsione e l'uso di variabili locali
 - In C++, supporta anche la gestione strutturata delle eccezioni



Concorrenza

- A partire dallo standard ISO/IEC 9899:2011, in un programma possono essere attivati ulteriori flussi di esecuzione
 - Eseguiti parallelamente al primo
 - Ciascuno dotato di un proprio stack

Implementazione del modello di esecuzione

- La libreria di esecuzione non può – da sola – implementare tutte le astrazioni del modello di esecuzione
 - In particolare quelle legate all'isolamento ed alle sue conseguenze
- Se più programmi sono in esecuzione, il (mal-) funzionamento di uno non deve avere conseguenze sugli altri
 - Occorre che il S.O. crei un meccanismo di isolamento semi-permeabile



Processi

- Per gestire l'esecuzione isolata dei programmi, i S.O. introducono il concetto di **processo**
 - Costituisce il contesto di esecuzione di un programma
 - Viene modellato da un'opportuna struttura dati interna al S.O.
- Il sistema operativo sovraintende alla creazione dei processi
 - e alterna, nel tempo, l'esecuzione dei flussi di elaborazione ad essi associati

Processi

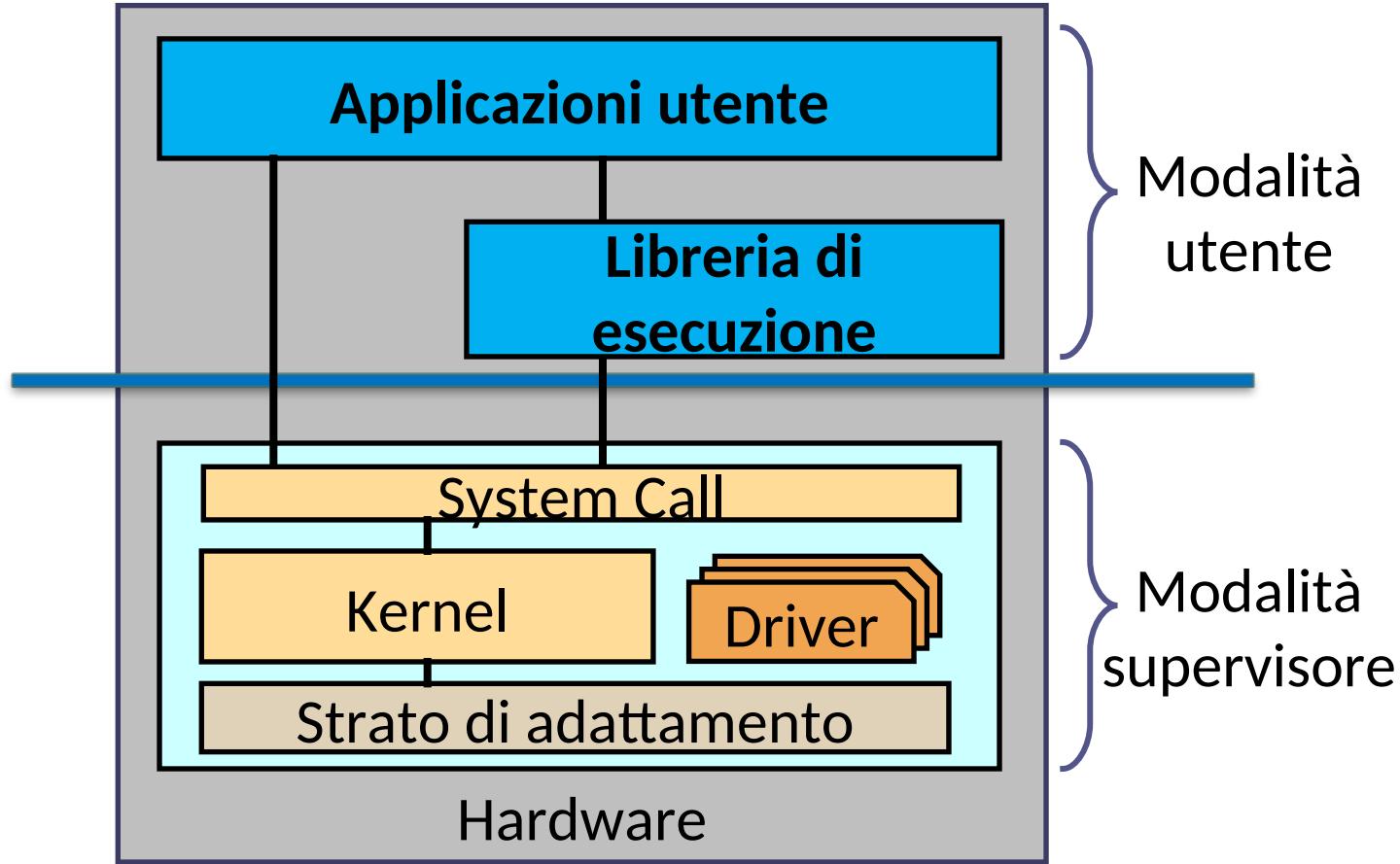
- Quando un processo viene selezionato per l'esecuzione, il S.O. :
 - configura il processore con le relative risorse (organizzazione della memoria, privilegi, periferiche, ...)
 - ripristina lo stato ad esse associato

Sistema Operativo

- Richiede un supporto hardware per distinguere due modalità di esecuzione
 - modalità **utente** – esecuzione di un sottoinsieme delle istruzioni offerte dalla CPU
 - modalità **supervisore** – accesso illimitato a tutte le funzionalità del sistema



Sistema operativo



Sistema operativo

- In modalità supervisore
 - Crea e gestisce le strutture dati che modellano i processi, il loro spazio di indirizzamento e le altre risorse che posseggono
 - Interagisce con le periferiche, il file system e la rete
- In modalità utente
 - Esegue il codice associato ai diversi processi
 - Sottostando ad un insieme di vincoli
 - Accesso parziale alla memoria ed alle istruzioni della CPU
 - Impossibilità di accedere direttamente alle periferiche



Innalzamento di privilegio

- Per permettere l'esecuzione di programmi dotati di una qualche utilità
 - i S.O. offrono meccanismi per accedere, in modo controllato, ad un insieme predefinito di funzionalità (System Call)
- Per garantire l'assenza di violazioni del modello di esecuzione
 - l'innalzamento di privilegio comporta la sostituzione dello stato della CPU e delle strutture di supporto all'esecuzione tramite istruzioni sicure



Innalzamento di privilegio

- Per i processori IA-32 ci sono due meccanismi
 - Interrupt software (trap) - più lento, funzionante anche su processori obsoleti
 - Le coppia di istruzioni SYSENTER/SYSEXIT - più efficienti, richiedono processori più recenti
- Il costo per attraversare la barriera tra modo utente e supervisore è comunque elevato
 - Nelle architetture IA-32 Windows/Linux l'operazione può richiedere 500 cicli
 - Una chiamata semplice, ne richiede 5



Esecuzione di un programma

- L'esecuzione di un programma comporta la creazione di un nuovo processo
 - Acquisendo le risorse necessarie
 - E inizializzandone lo stato in modo opportuno
- La creazione di un processo è articolata in diverse sotto-fasi
 - Creazione spazio di indirizzamento
 - Caricamento dell'eseguibile in memoria
 - Caricamento delle librerie
 - Avvio dell'esecuzione

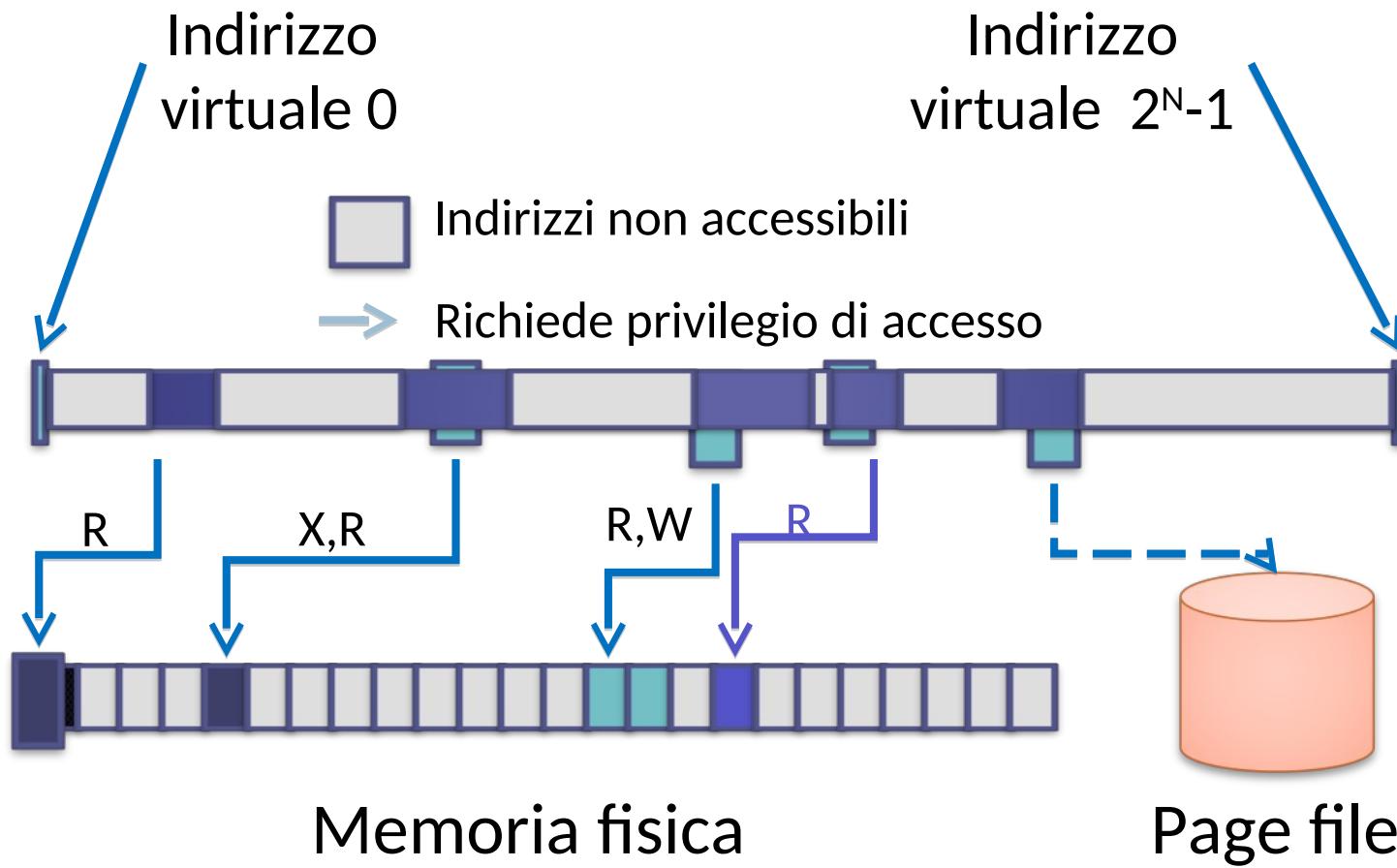


Spazio di indirizzamento

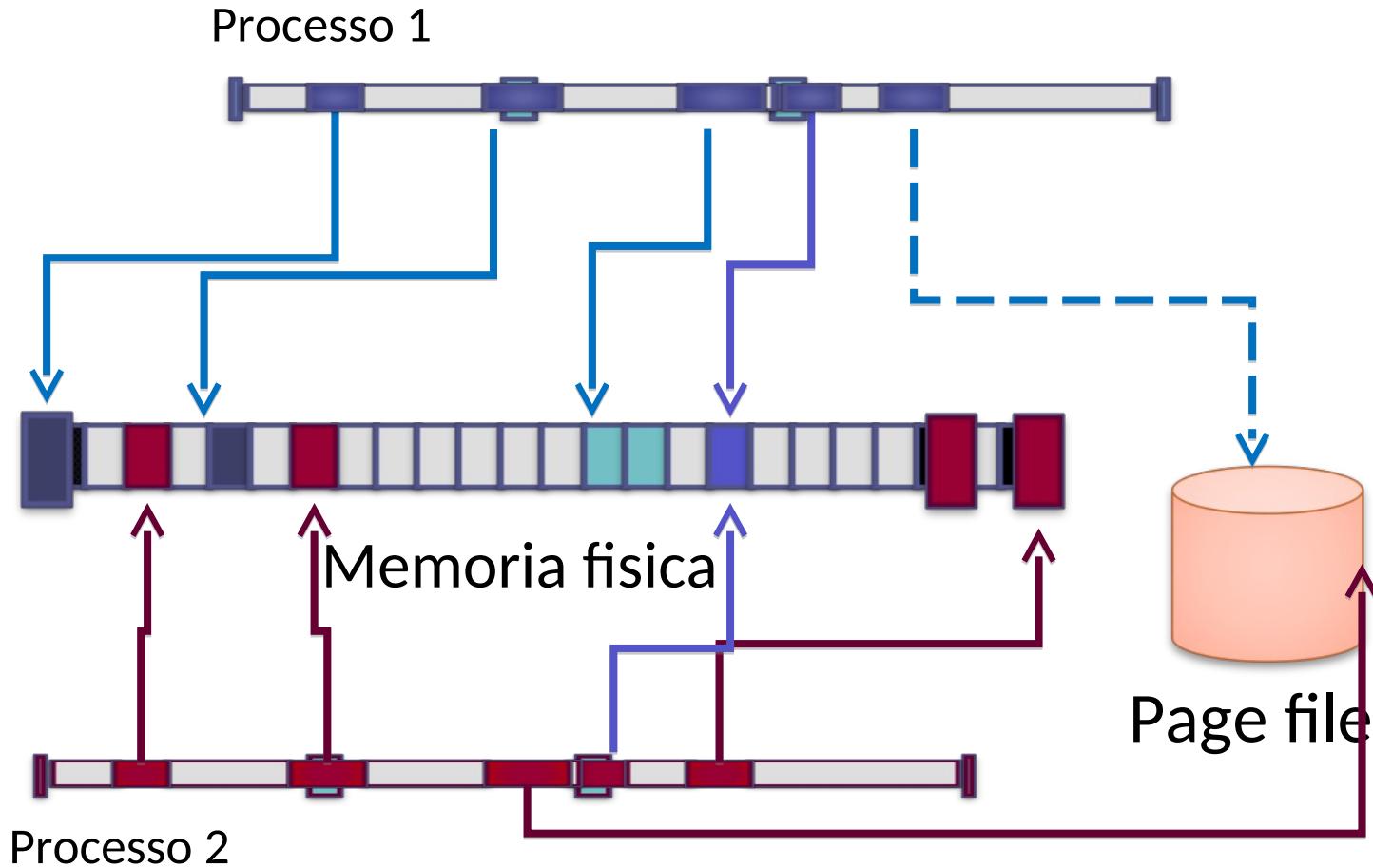
- L'esecuzione di un programma avviene nel suo spazio di indirizzamento
 - Insieme di locazioni di memoria accessibili tramite indirizzo virtuale
 - Sottoinsieme delle celle indirizzabili, gestito dal sistema operativo
- Attraverso funzionalità offerte dal blocco MMU del processore, gli indirizzi virtuali vengono tradotti in indirizzi fisici
 - Cambiando questa corrispondenza, possono essere creati innumerevoli spazi tra loro separati



Spazio di indirizzamento



Spazio di indirizzamento



Paginazione

- I processori supportano il meccanismo di page_fault
 - Tentativi di accesso ad indirizzi non mappati generano un'interruzione
 - Appoggiandosi ad una memoria persistente, si possono indirizzare contenuti molto più grossi della memoria fisica disponibile

File mapping

- Il S.O. può associare blocchi di indirizzi virtuali al contenuto di un file
 - In caso di accesso ad uno di essi, il S.O. trasferisce il corrispondente contenuto del file in una pagina fisica libera e la associa agli indirizzi relativi

Caricamento del codice eseguibile

- Il loader è il componente del S.O. responsabile di inizializzare lo spazio di indirizzamento con il contenuto del programma da eseguire
 - E di tutte le sue dipendenze
- Il file eseguibile è dotato di una struttura interna
 - Formato ELF in Linux
 - Formato PE2 in Windows
- Suddiviso in sezioni
 - Contenenti informazioni omogenee (codice, dati, ...)



Caricamento del codice eseguibile

- Nello spazio di indirizzamento si allocano una sezione per il codice ed una per i dati
 - Mappate sulle corrispondenti sezioni del file eseguibile
 - Ulteriori sezioni vengono create per stack e heap



Caricamento delle dipendenze

- Una sezione del file eseguibile è dedicata ad elencare gli ulteriori eseguibili necessari
 - Librerie dinamiche
 - Esse vengono ricorsivamente mappate nello spazio di indirizzamento
- Tutti i riferimenti a variabili/ funzioni delle dipendenze vengono aggiornati con gli indirizzi effettivi
 - Rilocazione degli indirizzi dei simboli contenuti nella tabella di importazione



Avvio dell'esecuzione

- Quando il processo di caricamento termina, è possibile avviare l'esecuzione
 - Un'apposita funzione della libreria di esecuzione è delegata a tale scopo
- Il comportamento della funzione di avvio dipende dal linguaggio di programmazione e dal S.O.
 - In generale ha il compito di garantire l'esistenza e la consistenza di tutte le necessarie strutture dati di supporto



La funzione di avvio

- Configurazione della piattaforma di esecuzione
 - Inizializzazione dello stack, dei registri e delle strutture dati per la gestione delle eccezioni
- Invocazione dei costruttori degli oggetti globali
- Invocazione della funzione principale
 - `main(int argc, char** argv)`
- Invocazione dei distruttori degli oggetti globali
 - Nell'ordine opposto a quello dei costruttori
- Terminazione del processo
 - Rilasciando l'intero spazio di indirizzamento e tutte le risorse in esse allocate, attraverso l'invocazione di un'opportuna system call (`exit`)



La funzione di avvio in GCC/ Linux

- Il linker indica come punto di ingresso la funzione `_start()`
 - Ha il compito di preparare una serie di parametri e di invocare 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  
) ;
```



La funzione di avvio in VisualStudio/Windows

- Il linker seleziona tale funzione in base al tipo di progetto
 - L'opzione /SUBSYSTEM:WINDOWS indica un'applicazione grafica e invoca WinMainCRTStartup()
 - E' l'opposto di /SUBSYSTEM:CONSOLE
- Un ulteriore distinzione viene fatta in base al set di caratteri usato
 - ANSI o UNICODE
- Il punto di ingresso utente riflette i quattro casi
 - main(...),wmain(...),WinMain(...),wWinMain(...)



La funzione di avvio in VisualStudio/Windows

- Il comportamento è simile
 - Inizializzazione della libreria di supporto
 - Invocazione dei costruttori
 - Preparazione degli argomenti
 - Funzione principale
 - Invocazione dei distruttori
 - Terminazione del processo



Spunti di riflessione

- Si crei, in linux, un'applicazione che stampi «Hello sysprog!» e poi termini
 - Senza usare la libreria di esecuzione (opzione `-nostdlib`)
 - Si usi come punto di ingresso la funzione `_start()`



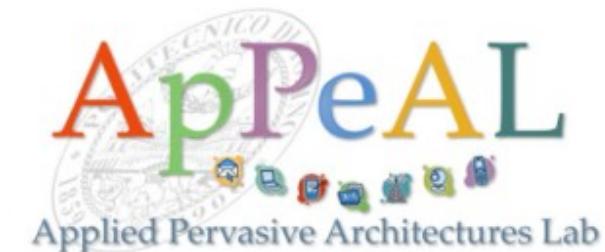


Allocazione della memoria

Programmazione di Sistema

A.A. 2017-18

Programmazione di Sistema



Argomenti

- Allocazione statica e dinamica
- Puntatori e loro utilizzo
- Allocazione in Linux
- Allocazione in Windows

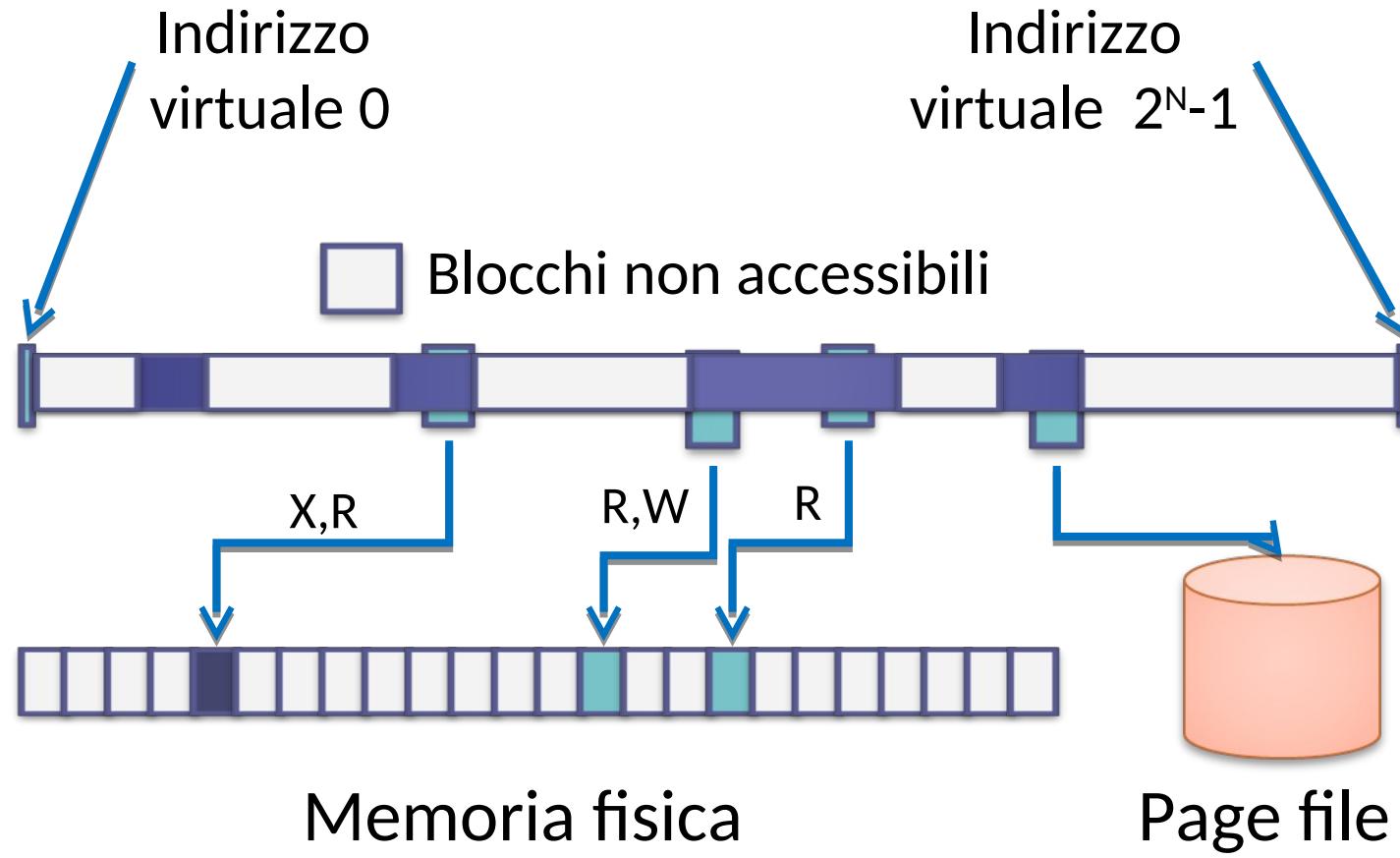


Spazio di indirizzamento

- L'esecuzione di un programma avviene nel suo spazio di indirizzamento
 - Insieme di locazioni di memoria accessibili tramite indirizzo virtuale
 - Sottoinsieme delle celle indirizzabili, gestito dal sistema operativo



Spazio di indirizzamento



Classi di allocazione

- Il sistema operativo/la libreria di esecuzione offrono aree diverse
 - In funzione dell'utilizzo e del ciclo di vita
 - Leggibili, scrivibili, eseguibili



Criteri di accesso

- La corrispondenza tra indirizzi virtuali e pagine fisiche è corredata di metadati
 - Definiscono quali operazioni sono lecite sulla memoria
 - execute, Read, Write, Copy_on_write
- Accessi a locazioni non mappate o in violazione dei criteri indicati comportano l'interruzione della CPU
 - Il processo viene terminato
 - Access violation (Windows) o segmentation fault (Linux)

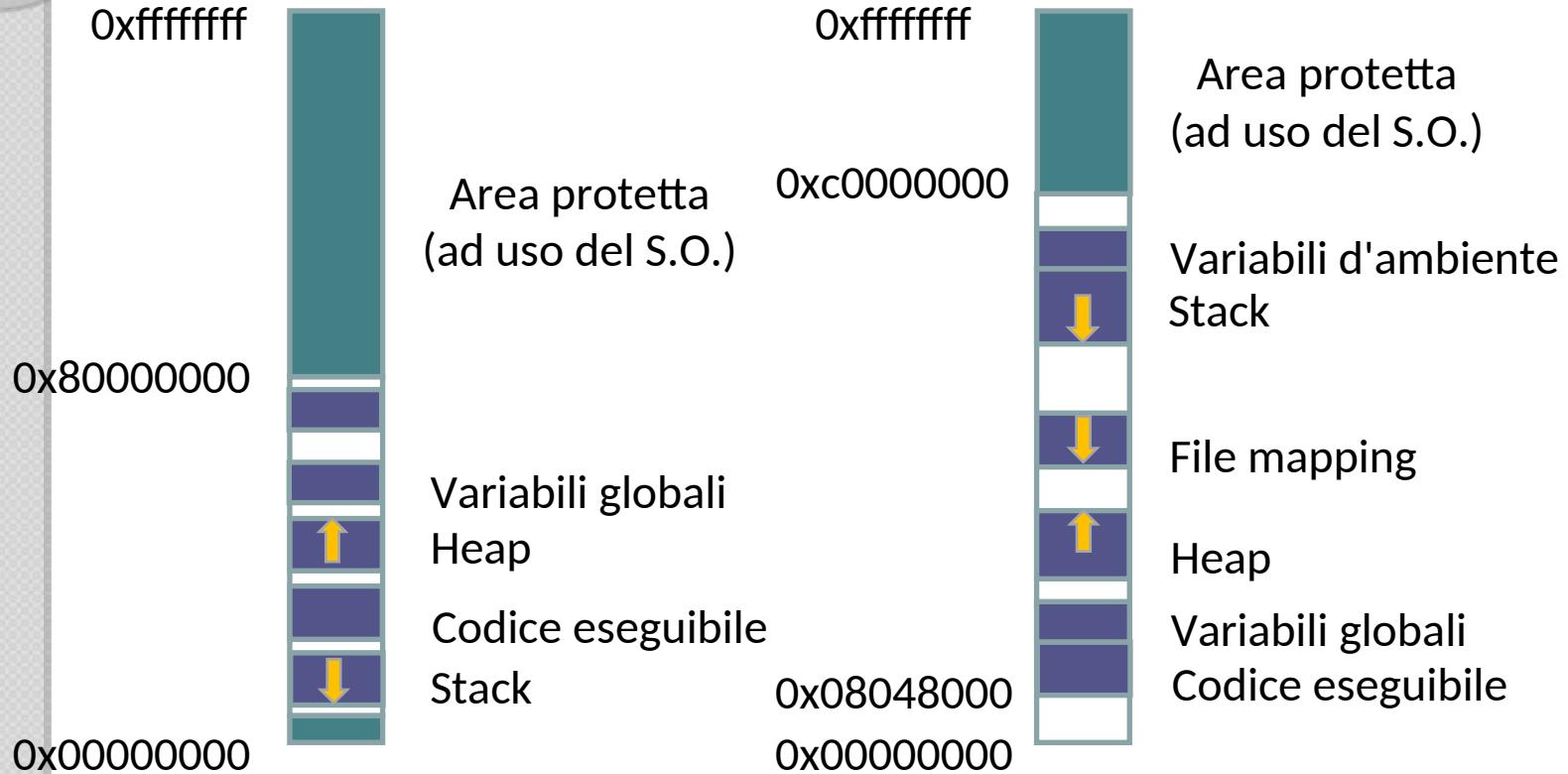


Uso della memoria

- Quando un processo viene creato, il suo spazio di indirizzamento viene popolato con diverse aree
 - Ciascuna dotata di propri criteri di accesso
 - Funzionali a supportare i modelli di esecuzione dei linguaggi C/C++



Spazio di indirizzamento in Windows e Linux



Organizzazione dello spazio di indirizzamento

- Codice eseguibile
 - Contiene le istruzioni in codice macchina
 - Accesso in lettura ed esecuzione
- Costanti
 - Accesso in sola lettura
- Variabili globali
 - Accesso lettura/scrittura
- Stack
 - Contiene indirizzi e valori di ritorno, parametri e variabili locali
 - Accesso lettura/scrittura



Organizzazione dello spazio di indirizzamento

- Free store o heap
 - Insieme di blocchi di memoria disponibili per l'allocazione dinamica
 - Gestiti tramite funzioni di libreria (`malloc`, `new`, `free`, ...) che li frammentano e ricompattano in base alle richieste del programma



Ciclo di vita delle variabili

- Il modello di esecuzione del C distingue diverse classi di variabili
 - Globali, locali, dinamiche
- Ciascuna classe ha un proprio ciclo di vita
 - Intervallo di tempo in cui è garantito l'accesso alle informazioni

Variabili globali e locali

- Le variabili globali hanno un indirizzo fisso, determinato dal compilatore e dal linker
 - Accessibili in ogni momento
 - All'avvio del programma, contengono l'eventuale valore di inizializzazione
- Le variabili locali hanno un indirizzo relativo alla cima dello stack
 - Ciclo di vita coincidente con quello della funzione/blocco in cui sono dichiarate
 - Valore iniziale casuale



Variabili dinamiche

- Hanno un indirizzo assoluto, determinato in fase di esecuzione
 - Accessibili solo tramite puntatori
 - Il programmatore ne controlla il ciclo di vita
 - Il valore iniziale può essere inizializzato o meno
- L'uso di questo tipo variabili presuppone un'infrastruttura di supporto che offra meccanismi di allocazione e di rilascio
 - Fornita dalla libreria di esecuzione e dal S.O.



Allocazione della memoria

- La libreria standard C offre vari meccanismi per ottenere un puntatore dinamico
 - `void *malloc(size_t s)`
 - `void *calloc(int n, size_t s)`
 - `void *realloc(void* p, size_t s)`
- In C++ viene definito il costrutto `new` `NomeClasse`
 - Alloca nello heap un blocco di dimensioni opportune
 - Invoca il costruttore della classe sul blocco
 - Restituisce il puntatore all'oggetto inizializzato



Allocazione della memoria

- Per allocare sequenze di oggetti, C++ offre il costrutto `new[] NomeClasse`
 - Si indica il numero di oggetti consecutivi da allocare tra le quadre
 - Inizializza i singoli oggetti
 - Restituisce il puntatore all'array



Rilascio della memoria

- Opportune funzioni di libreria permettono di restituire i blocchi precedentemente allocati
 - Organizzandoli in una lista in base alla dimensione e altri criteri
- 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
 - `malloc/free`, `new/delete`, `new[]/delete[]`
- Se il blocco viene rilasciato con la funzione sbagliata
 - si rischia di corrompere le strutture dati degli allocator, con conseguenze imprevedibili



Puntatori

- Permettono l'accesso diretto ad un blocco di memoria
 - Appartenente ad altri oggetti
 - `int A=10; int* pA = &A;`
 - Allocato allo scopo
 - `int* pB = new int(24);`
- Possono essere invalidi
 - Il valore 0
 - La macro NULL ((void *)0)
 - La parola-chiave nullptr (C++11)
- Quando si usa un puntatore, occorre stabilire quali responsabilità/permessi sono associati al loro uso



Le ambiguità dei puntatori

- Quanto è grosso il blocco puntato?
- Fino a quando è garantito l'accesso?
- Se ne può modificare il contenuto?
- Occorre rilasciarlo?
- Lo si può rilasciare o il blocco è accessibile tramite una copia del puntatore?



Usi dei puntatori

- Come strumento per accedere qui ed ora ad un'informazione fornita da altri
 - La responsabilità per la gestione della memoria del dato è totalmente esterna all'osservatore
- Caso più semplice e alquanto frequente

Puntatori

```
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;  
    } else  
        //operazione fallita  
        return 0;  
}
```



Puntatori

- Per accedere ad array monodimensionali di dati
 - Il compilatore trasforma gli accessi agli array in operazioni aritmetiche sui puntatori
 - Si perde di vista l'effettiva dimensione della struttura dati
- Occorre fare attenzione a non spostare il puntatore al di fuori dell'effettiva zona di sua pertinenza



Puntatori

```
{  
  
char* ptr = "Quel ramo del lago di Como...";  
  
//conta gli spazi  
int n=0;  
//usa l'aritmetica dei puntatori  
for (int i=0; *(ptr+i)!=0; i++) {  
  
    //usa il puntatore come fosse un array  
    if ( isSpace(ptr[i]) ) n++;  
}  
...  
}
```



Puntatori

- Come modo per accedere ad un dato memorizzato sullo heap
 - Il ciclo di vita del dato è slegato da quello del blocco di codice che lo ha allocato
 - Chi è responsabile del suo rilascio e quando va fatto?
- E' il caso base di strutture dati dinamiche

Puntatori

```
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);
```

Puntatori

- Strumento per implementare strutture dati composte
 - Liste, grafi, mappe, ...
 - La struttura nel suo complesso è responsabile della gestione di tutte le sue parti

```
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
```

Problemi legati ai puntatori

- In C, la gestione della memoria è totalmente affidata al programmatore
 - Non ci sono supporti sintattici per identificare l'uso associato ad un puntatore



Responsabilità del programmatore

- Limitare gli accessi ad un blocco
 - Nello spazio
 - Non accedere alle locazioni che lo precedono o che lo seguono
 - Nel tempo
 - Non accedere al blocco al di fuori del suo ciclo di vita
- Non assegnare a puntatori valori che corrispondono ad indirizzi non mappati
 - Possibile nel caso di cast o di assegnazione improprie
- Rilasciare tutta la memoria dinamica allocata
 - Una e una sola volta



Rischi

- Accedere ad un indirizzo quando il corrispondente ciclo di vita è terminato, ha effetti impredicibili
 - *Dangling pointer*
 - La memoria indirizzata può essere inutilizzata, in uso ad altre parti del programma o non mappata
- Non rilasciare la memoria non più in uso, spreca risorse del sistema
 - *Memory leakage*
 - Se si continua ad allocare senza mai rilasciare, si può saturare lo spazio di indirizzamento



Rischi

- Se si assegna ad un puntatore un indirizzo non mappato e si usa il puntatore, viene generata un'interruzione
 - Il S.O., per lo più, termina il processo
- Se non si inizializza un puntatore e lo si usa, il suo contenuto potrebbe puntare ovunque
 - *Wild pointer*
 - Causando una violazione d'accesso
 - Oppure corrompendo un'area di memoria in uso ad altre parti del programma



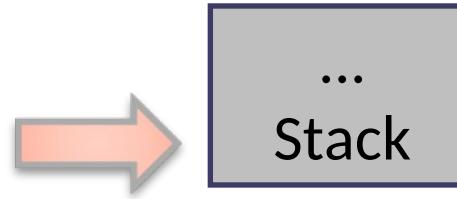
Dangling Pointer

```
{  
    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  
}
```



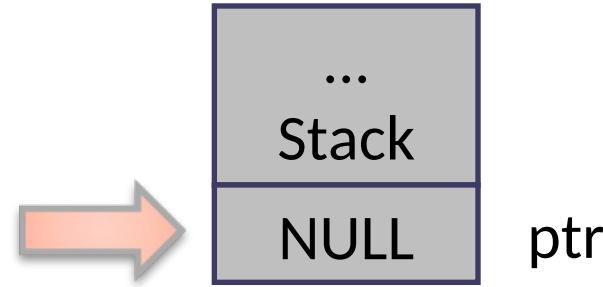
Dangling Pointer

```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
  
}  
  
printf("%c", *ptr);  
  
}
```



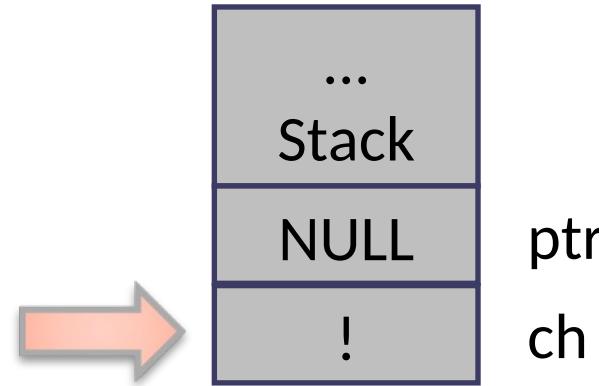
Dangling Pointer

```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
  
}  
  
printf("%c", *ptr);  
  
}
```



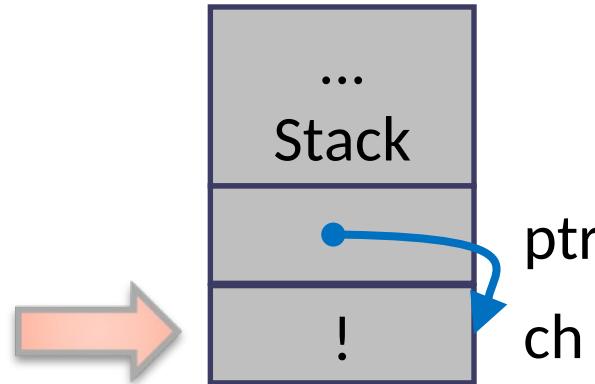
Dangling Pointer

```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
  
}  
  
printf("%c", *ptr);  
  
}
```



Dangling Pointer

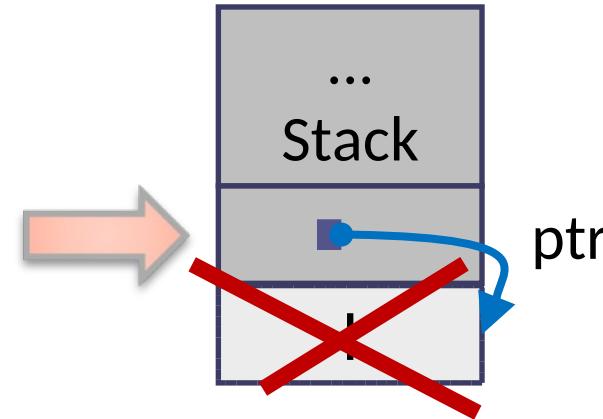
```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
}  
  
printf("%c", *ptr);  
}
```



Dangling Pointer

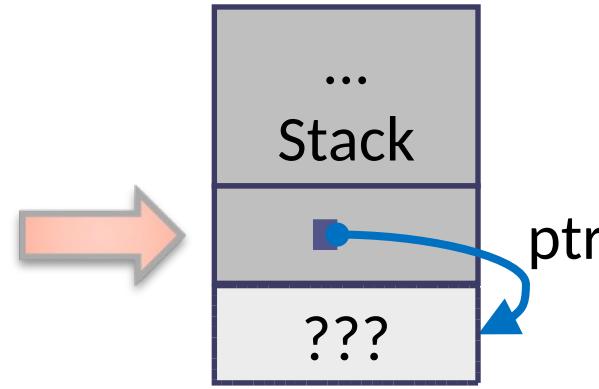
```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
  
}  
printf("%c", *ptr);
```

```
}
```



Dangling Pointer

```
{  
    char* ptr = NULL;  
  
{  
    char ch='!';  
    ptr = &ch;  
  
}  
  
printf("%c", *ptr);  
  
}
```

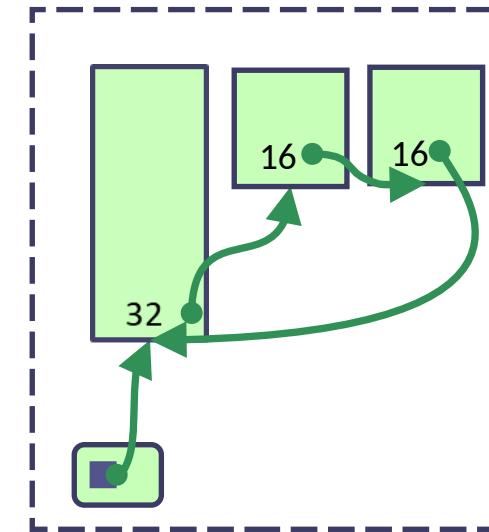
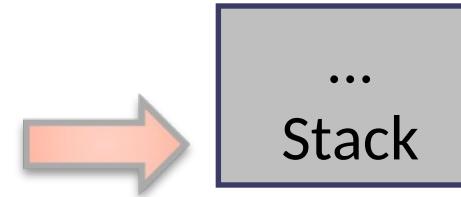


Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10); //Alloco un blocco  
  
    strncpy(ptr,10,"Leakage!"); //Lo uso  
  
    printf("%s\n", ptr);  
  
}  
                                //Ne perdo le tracce
```

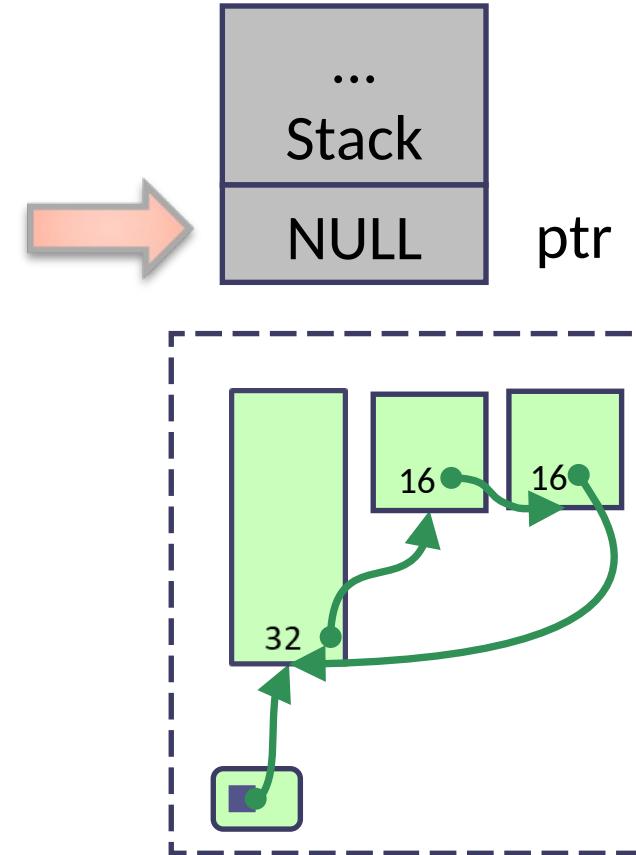
Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
}
```



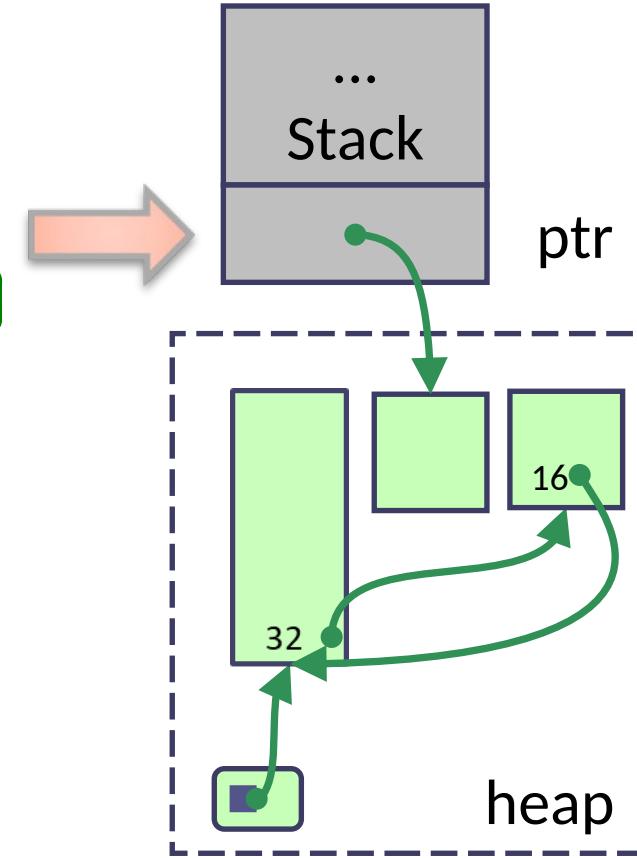
Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
  
}
```



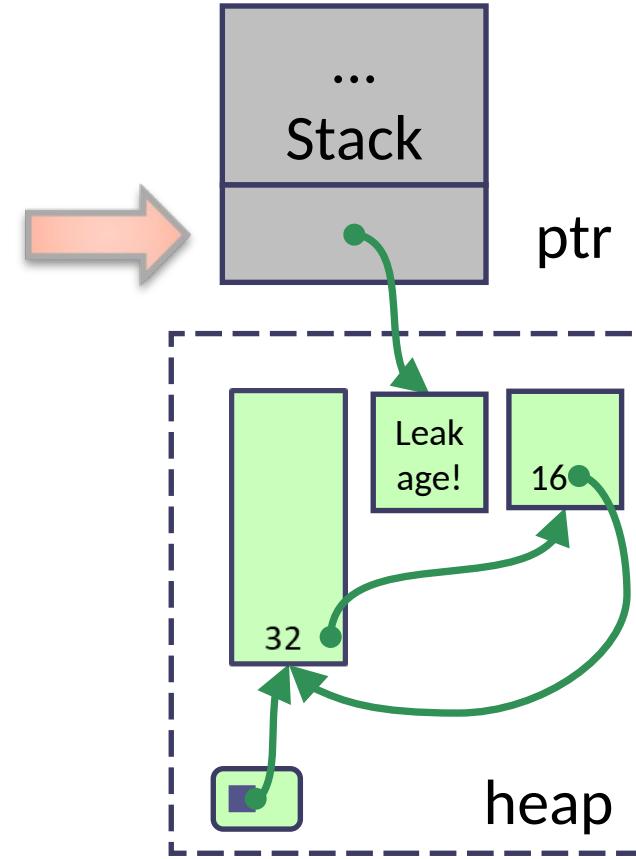
Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
  
}
```



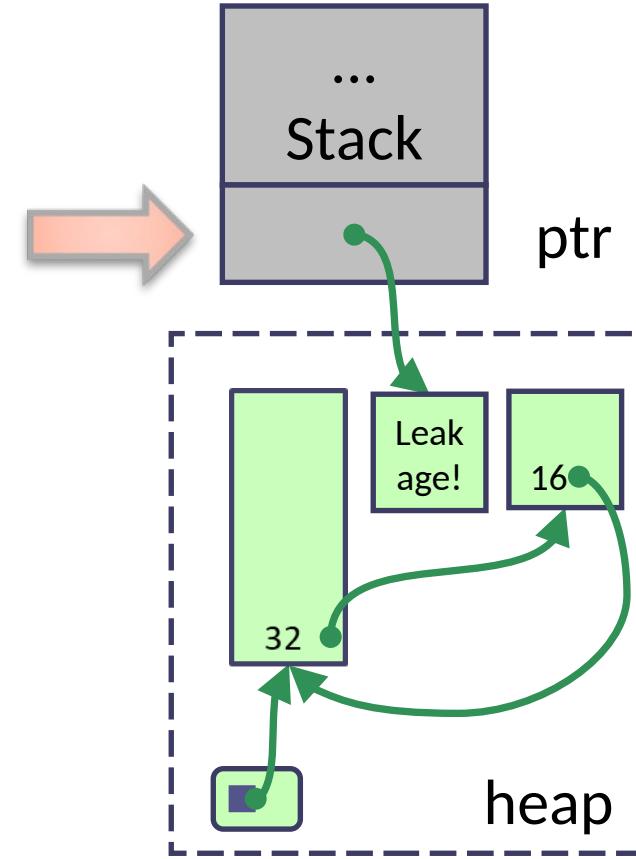
Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
  
}
```



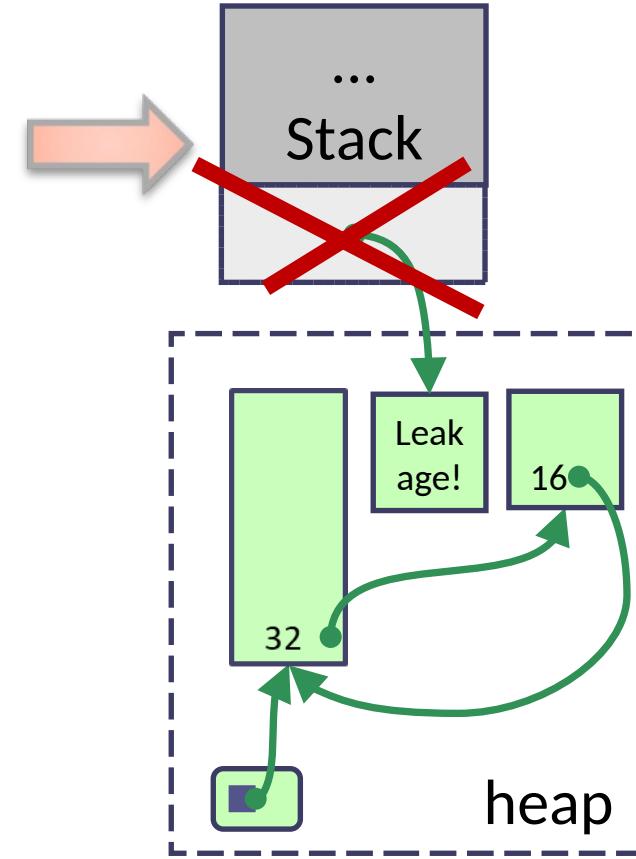
Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
  
}
```



Memory leakage

```
{  
    char* ptr = NULL;  
  
    ptr = (char*) malloc(10);  
  
    strncpy(ptr,10,"Leakage!");  
  
    printf("%s\n", ptr);  
  
}
```



Gestire i puntatori

- Chi alloca un blocco di memoria è responsabile di mettere in atto un meccanismo che ne garantisca il successivo rilascio
 - Viene detto “possessore” del puntatore
- Cosa capita se un blocco di codice copia un puntatore?
 - Si trova coinvolto, suo malgrado, nel ciclo di vita
 - Occorre introdurre un meccanismo per gestire efficacemente la semantica di un puntatore



Possesso della memoria

- Il vincolo di rilascio risulta problematico per via di un'ambiguità del linguaggio
 - Dato un indirizzo non nullo, non è possibile distinguere a quale area appartenga né se sia valido o meno
- Ogni volta in cui si alloca un blocco dinamico e se ne salva l'indirizzo in una variabile puntatore
 - Quella variabile diventa proprietaria del blocco
 - Ha la responsabilità di liberarlo



Proprietà della memoria

- Non tutti i puntatori posseggono il blocco a cui puntano
 - Se ad un puntatore viene assegnato l'indirizzo di un'altra variabile, la proprietà è della libreria di esecuzione
- Il problema si complica se un puntatore che possiede il proprio blocco viene copiato
 - Quale delle due copie è responsabile del rilascio



Tecniche di sopravvivenza

- Utilizzo di strumenti per la diagnosi dei processi
 - Valgrind (linux)
 - Dr.Memory (windows)
- Incapsulamento dei puntatori in apposite strutture dati
 - SmartPointer in C++



Spunti di riflessione

- Si crei un programma C che allochi e non rilasci un blocco di memoria
- Si cerchi di scoprire tale perdita utilizzando i programmi valgrind e Dr.Memory

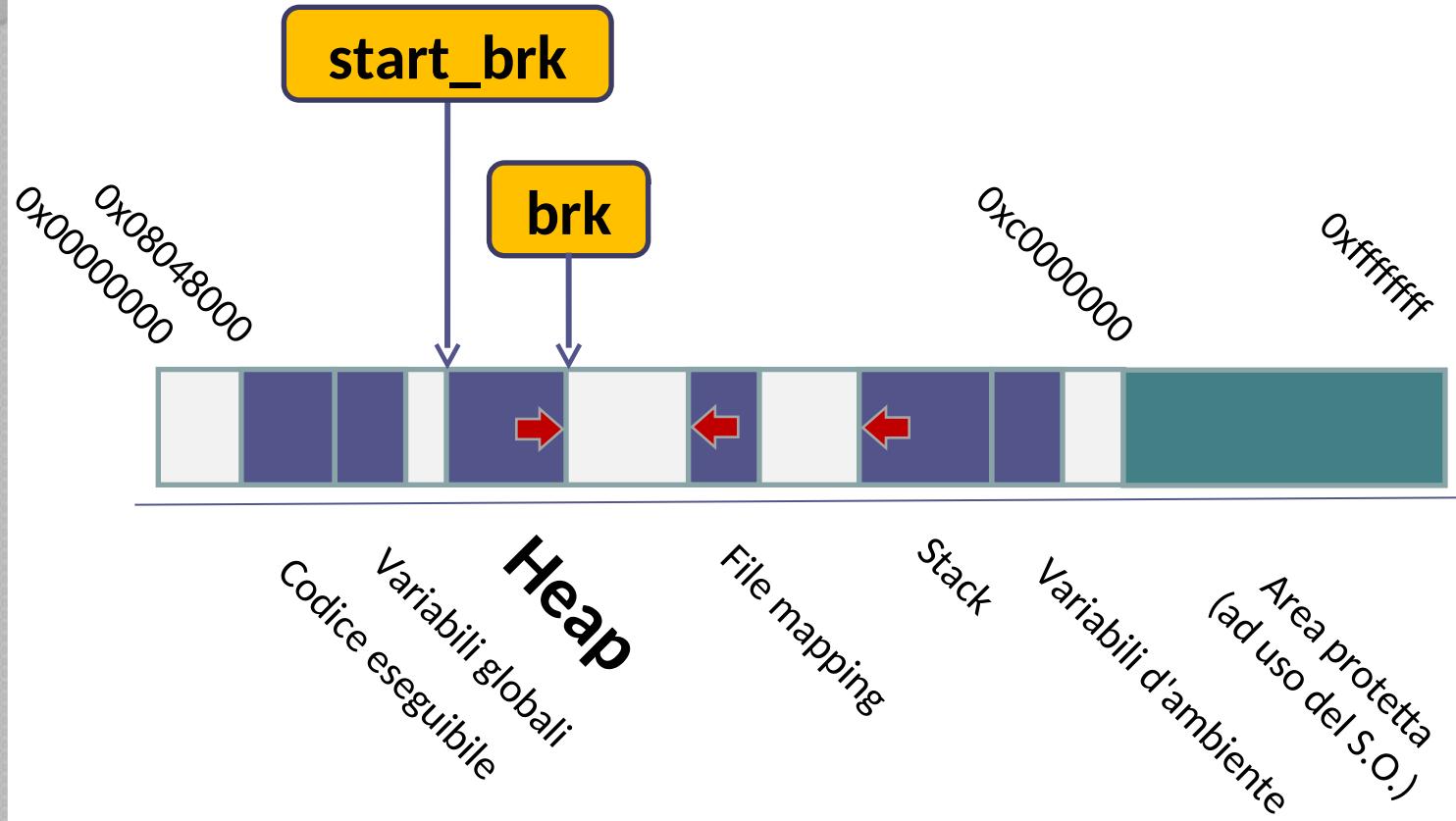


Allocazione in Linux

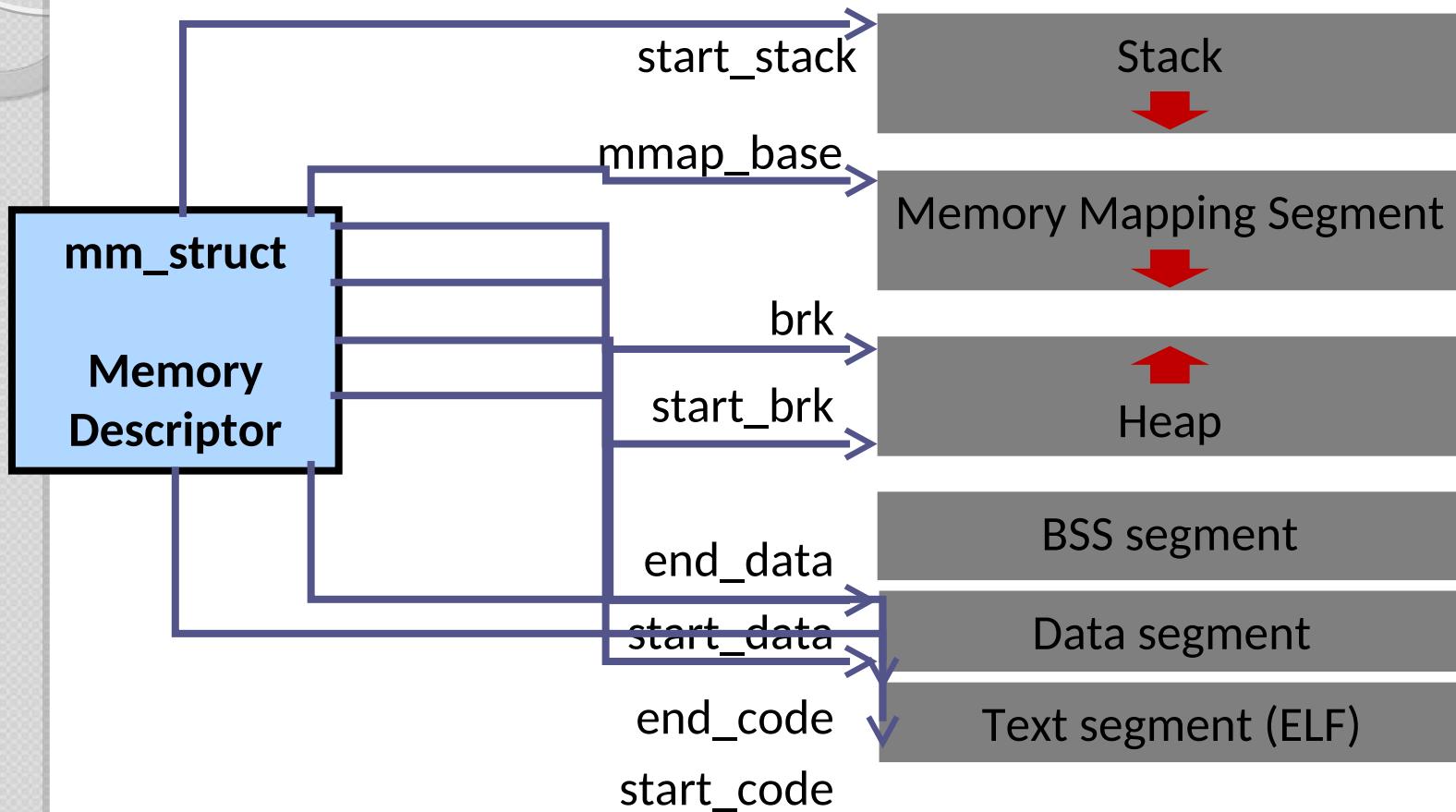
- Quando il processo viene inizializzato, viene creato un blocco di memoria a disposizione dello heap
 - Le funzioni di allocazione cercheranno di utilizzare la memoria disponibile per soddisfare le richieste
- Se nessuno dei blocchi liberi ha dimensioni sufficienti, occorre richiedere al S.O. altra memoria
 - I dettagli di tale operazione dipendono dall'implementazione



Lo spazio di indirizzamento in Linux



La rappresentazione interna



Allocare altro spazio

- Il sistema mantiene compatta l'area dello heap
 - Delimitandola con due puntatori interni al sistema
 - `start_brk` indica l'inizio dello heap
- int brk(void *end_data_segment);
- Fissa la dimensione del segmento dati specificando il valore limite, chiamato program break
 - La memoria disponibile varia di conseguenza
 - In caso di errore ritorna -1



La system call sbrk()

- Sposta la locazione del program break della quantità indicata
 - Incrementando o diminuendo l'area dello heap
 - In caso di errore ritorna `(void*)-1`
- ```
void *sbrk(intptr_t increment);
```



# Politiche di allocazione

- Per evitare di raggiungere tale limite, se il blocco richiesto ha dimensioni elevate (>128KByte)
  - malloc() richiede la creazione di un blocco separato attraverso la system call mmap()



# Allocazione in Windows

- Un processo può gestire più heap
  - Quando il processo viene creato, c'è uno heap di base, usato da malloc
  - Altri heap possono essere creati e distrutti per implementare politiche particolari



# HeapCreate & HeapDestroy

- **HANDLE HeapCreate(...)**
  - Crea un nuovo heap, specificandone le opzioni e dimensioni iniziali e massime
  - Restituisce un identificatore opaco (handle) che lo rappresenta
- **BOOL HeapDestroy(HANDLE h)**
  - Rilascia tutto lo spazio di indirizzamento associato allo heap indicato
  - Rende inaccessibile tutte le eventuali aree di memoria precedentemente ottenute da questo heap



# HeapAlloc & HeapFree

- `void *HeapAlloc(HANDLE h, DWORD options, SIZE_T s)`
  - Alloca, dalla regione dello heap, un'area di dimensione *s*
  - Restituisce il puntatore relativo o NULL
- `BOOL HeapFree(HANDLE h, DWORD options, void* ptr)`
  - Rilascia un blocco precedentemente allocato
  - Restituisce FALSE in caso di errore





# 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[10];
- Puntatori
  - int\* ptr;
  - Un puntatore può essere invalido (NULL, nullptr)
  - Gli operatori \* e -> dereferenziano un puntatore



# Tipi derivati

- Riferimenti

- int i=0;  
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

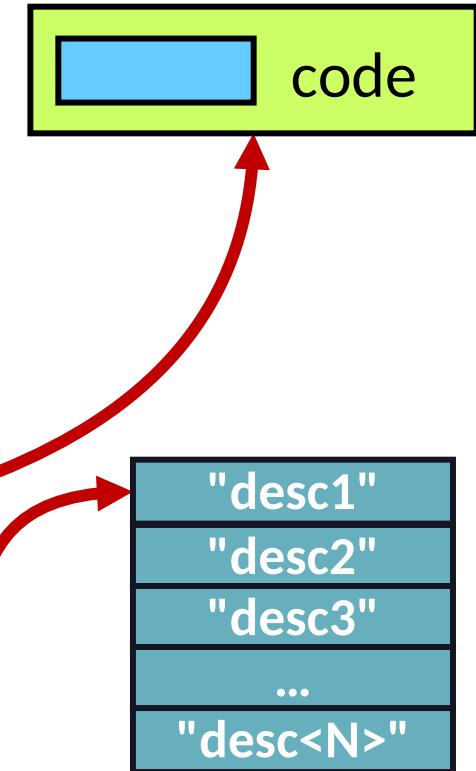
- Ad ogni oggetto corrisponde un blocco di memoria
  - Contiene gli attributi non statici (pubblici, privati e protetti) ed eventuali altre informazioni utili al compilatore
- Un oggetto può essere allocato in varie aree
  - Memoria globale
  - Stack (local store)
  - Heap (free store)
  - All'interno 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

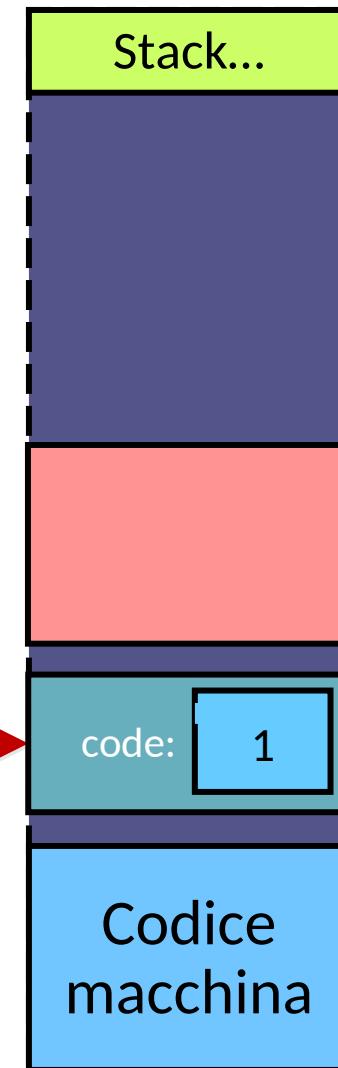
- Si possono allocare dinamicamente array di oggetti
  - Tramite il costrutto `ptr = new[count]  
ClassName`
- Occorre rilasciare il blocco tramite l'operatore duale
  - `delete[] ptr;`

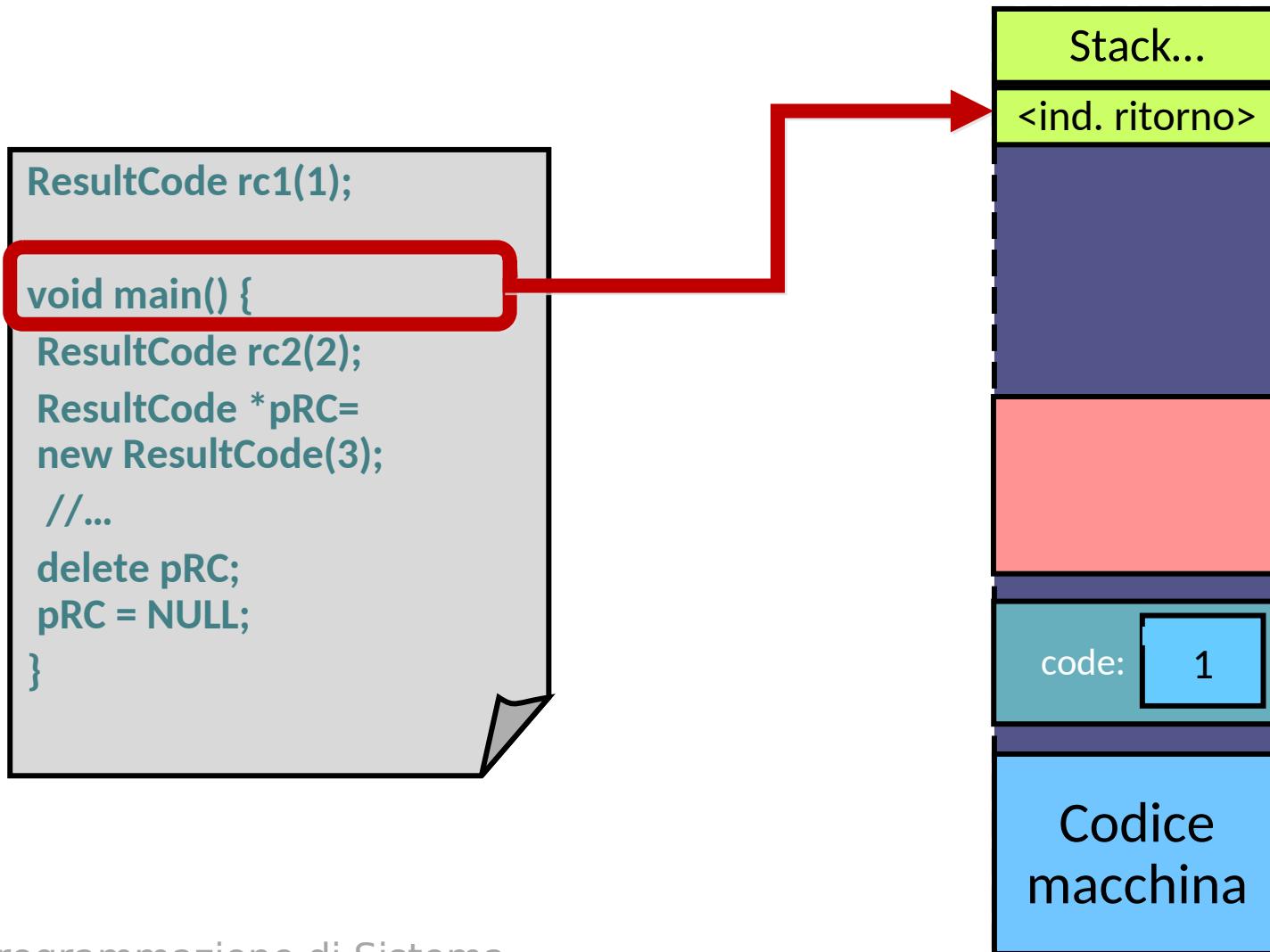




```
resultCode rc1(1);

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

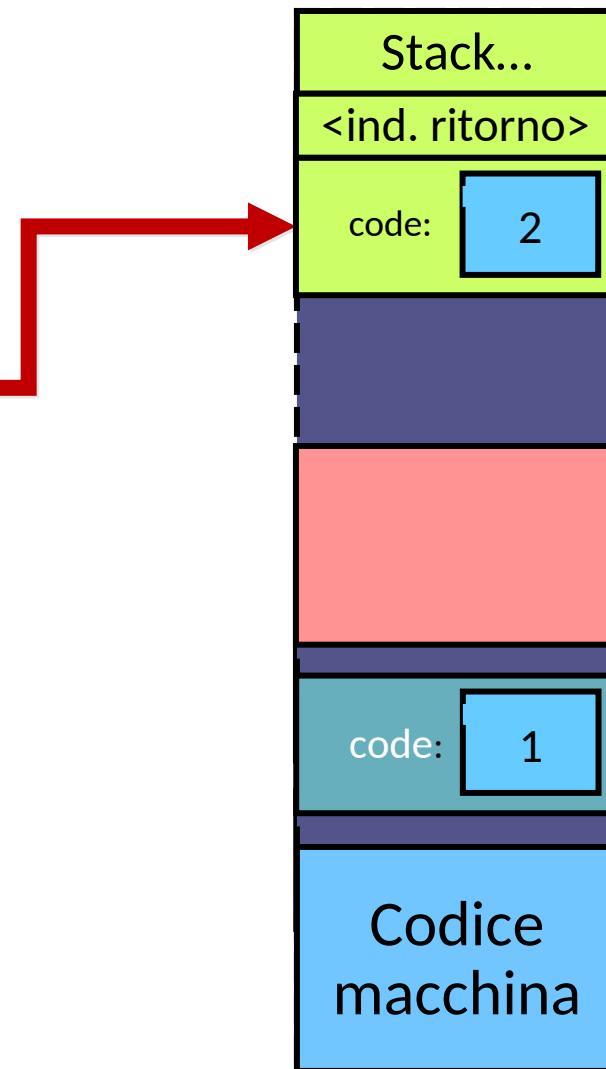






```
resultCode rc1(1);

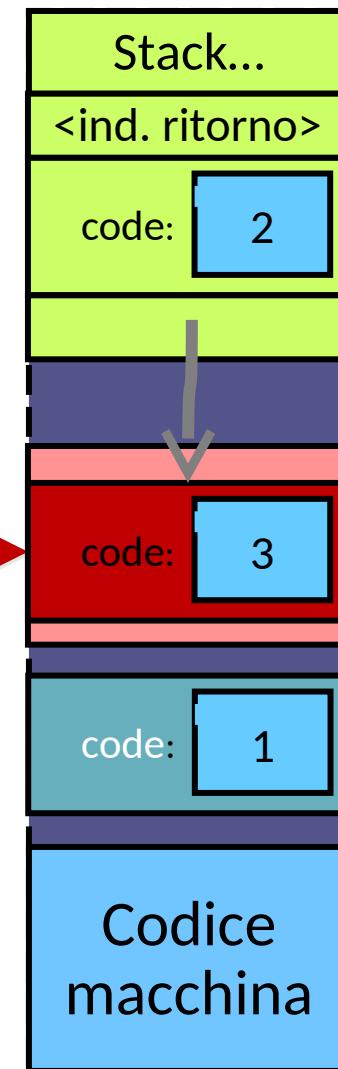
void main() {
 resultCode rc2(2);
 resultCode *pRC=
 new resultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```





```
resultCode rc1(1);

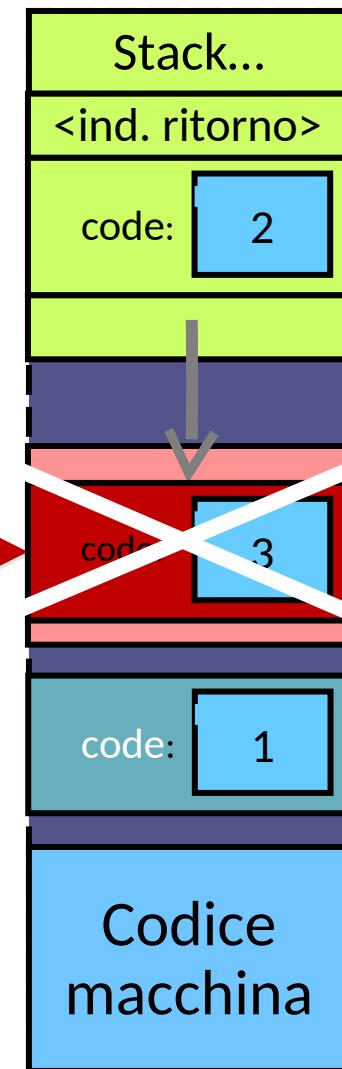
void main() {
 resultCode rc2(2);
 resultCode *pRC= ...
 new resultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```





```
resultCode rc1(1);

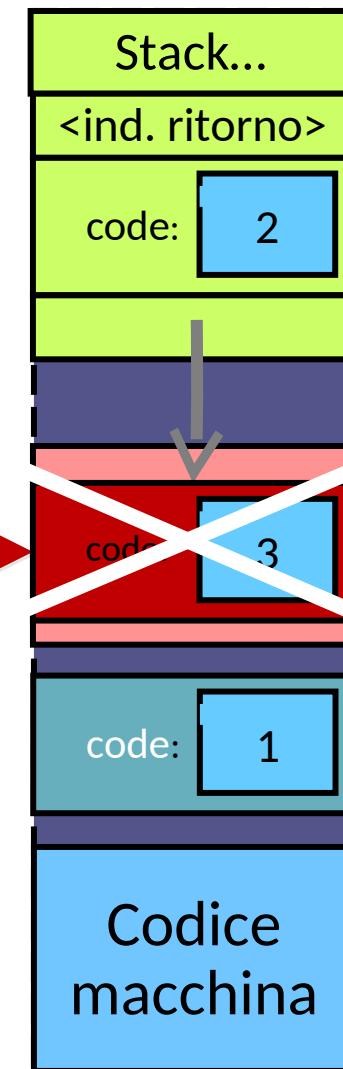
void main() {
 resultCode rc2(2);
 resultCode *pRC=
 new resultCode(3);
 //
 delete pRC;
 pRC = NULL;
}
```





```
resultCode rc1(1);

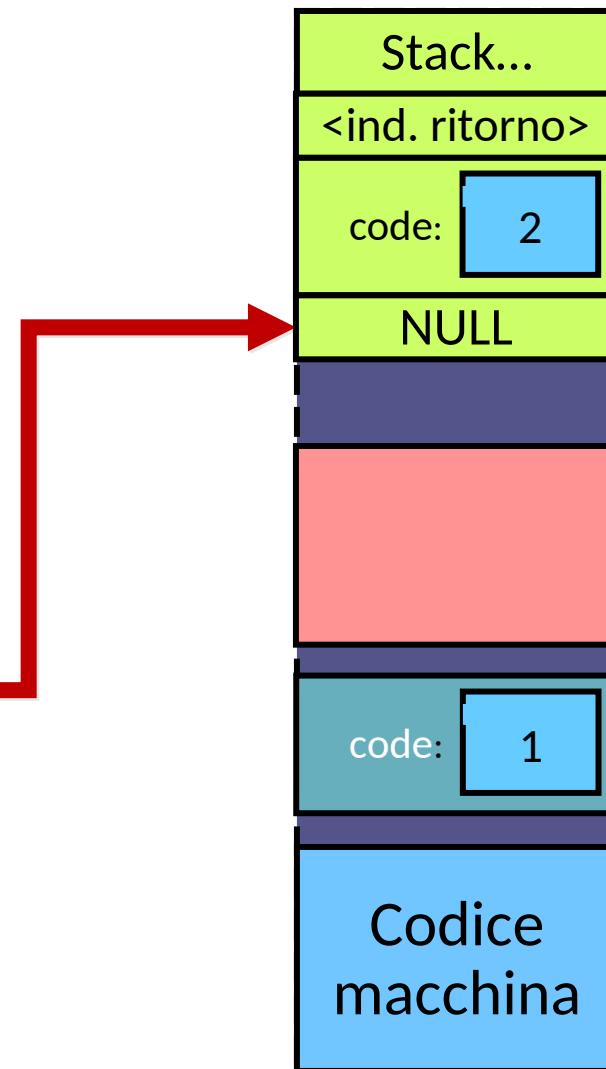
void main() {
 resultCode rc2(2);
 resultCode *pRC=
 new resultCode(3);
 //
 delete pRC;
 pRC = NULL;
}
```





```
resultCode rc1(1);

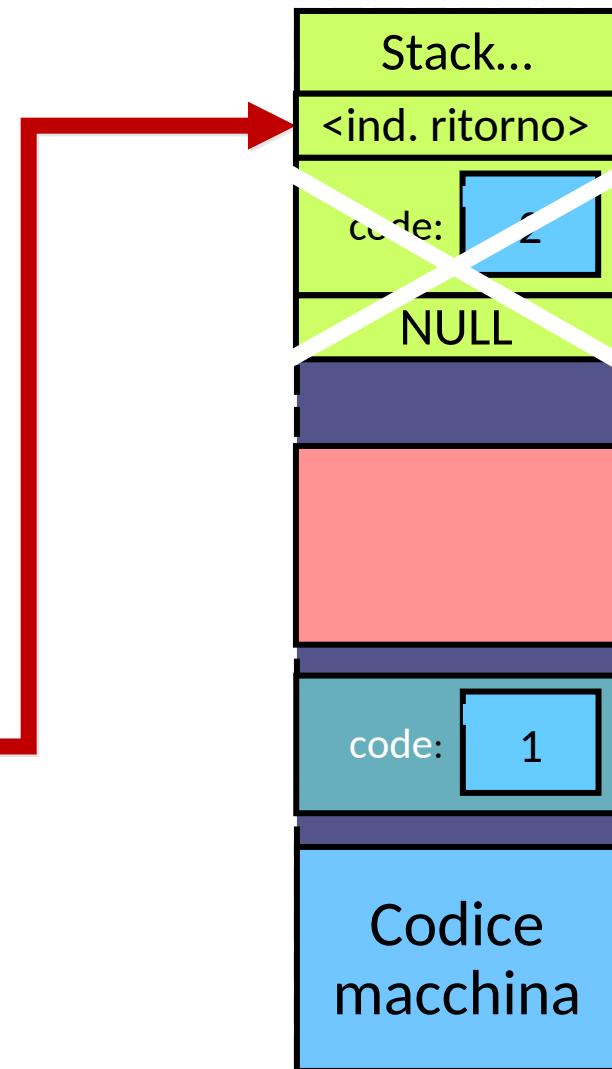
void main() {
 resultCode rc2(2);
 resultCode *pRC=
 new resultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```





```
resultCode rc1(1);

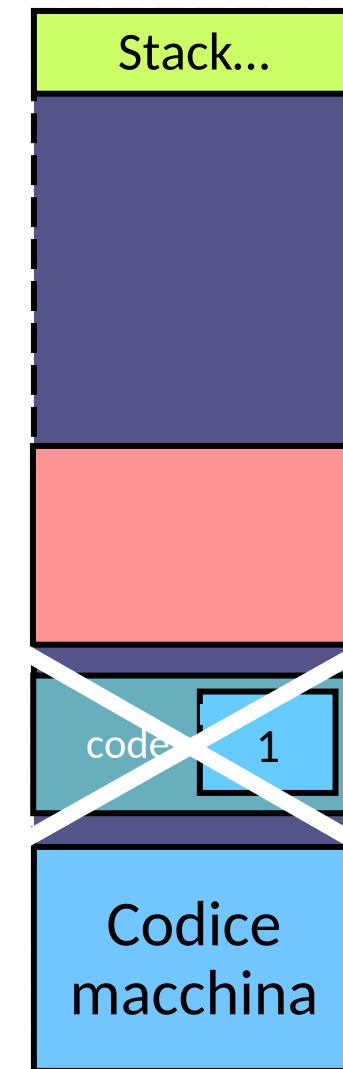
void main() {
 resultCode rc2(2);
 resultCode *pRC=
 new resultCode(3);
 //...
 delete pRC;
 pRC = NULL;
}
```





```
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;
}
```

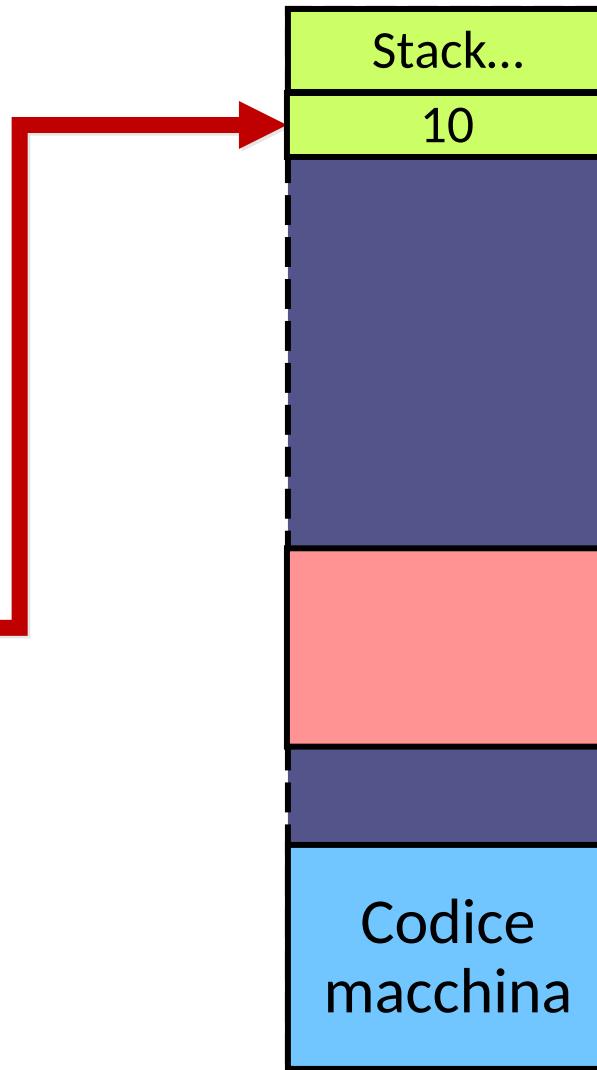
```
int main() {
 int i=10;
 f(i);
 return i;
}
```



# Passaggio per valore

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

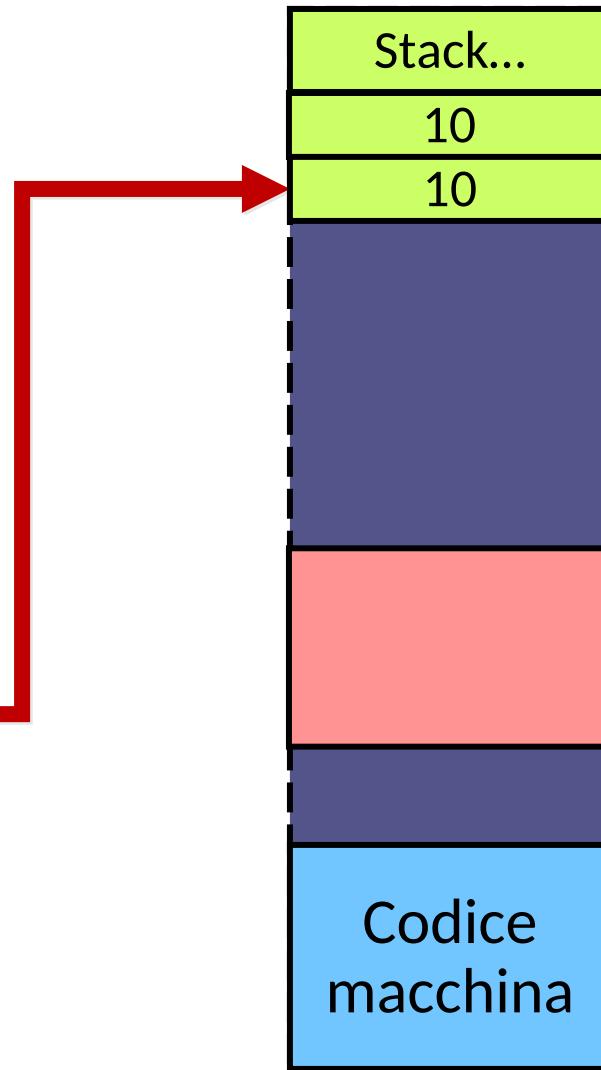
int main() {
 int i=10;
 f(i);
 return i;
}
```



# Passaggio per valore

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

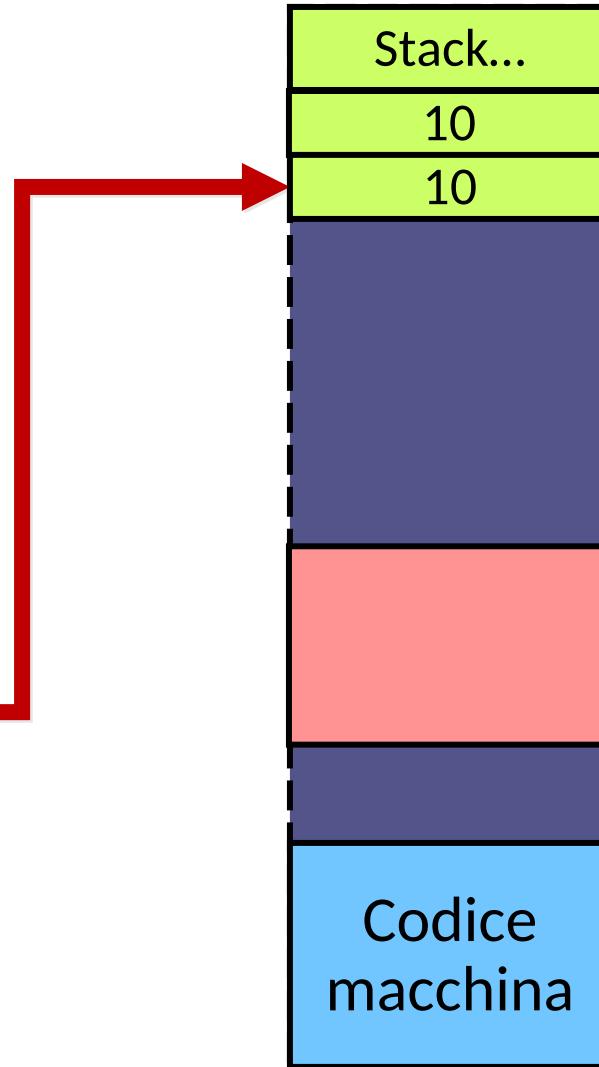
int main() {
 int i=10;
 f(i);
 return i;
}
```



# Passaggio per valore

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

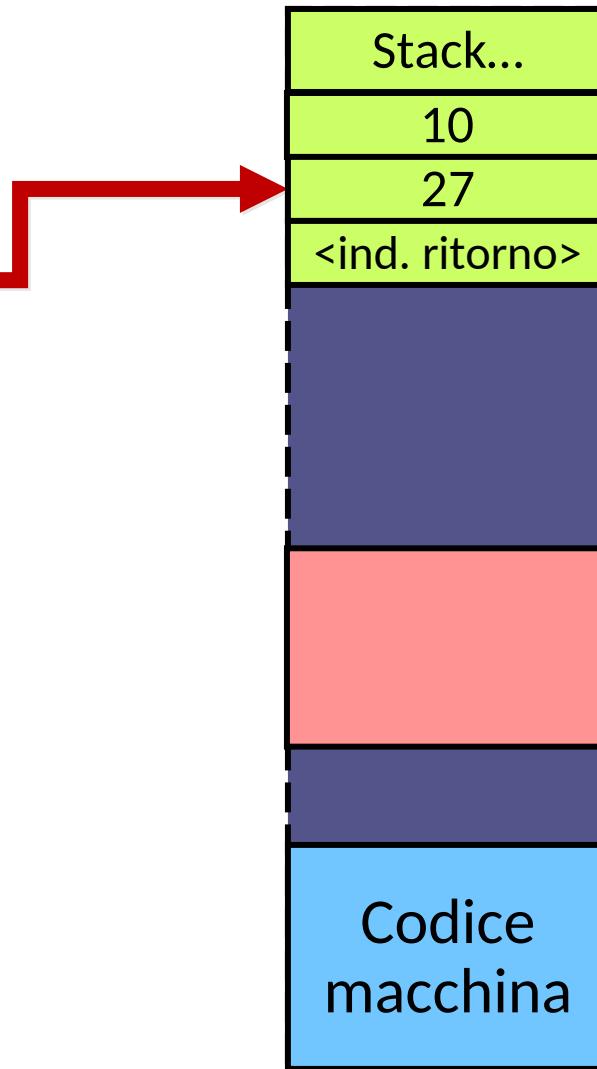
int main() {
 int i=10;
 f(i);
 return i;
}
```



# 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;
```

```
}
```

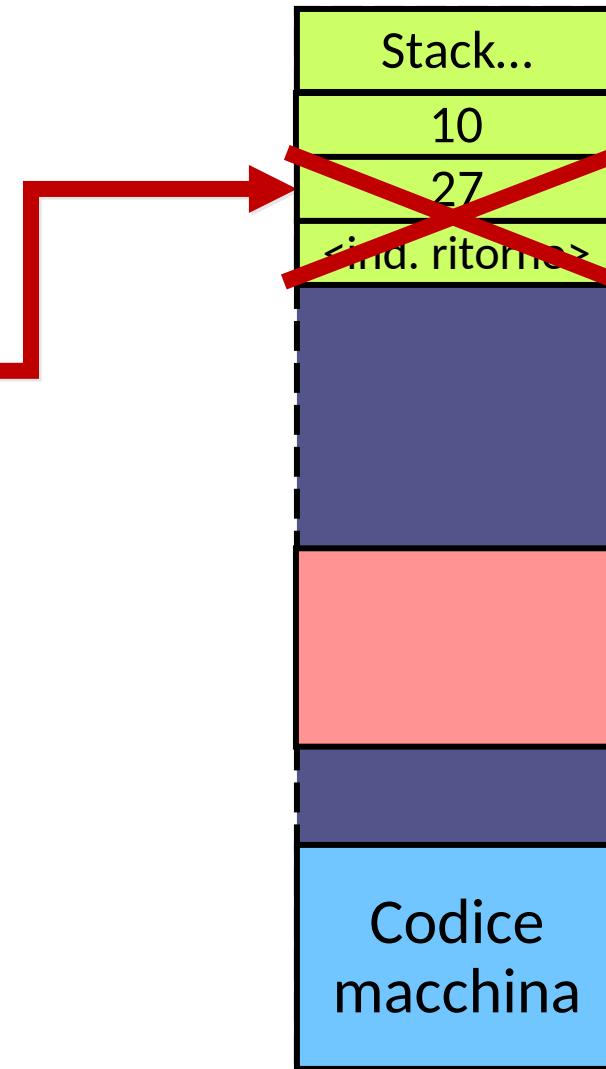
```
int main() {
```

```
 int i=10;
```

```
 f(i);
```

```
 return i;
```

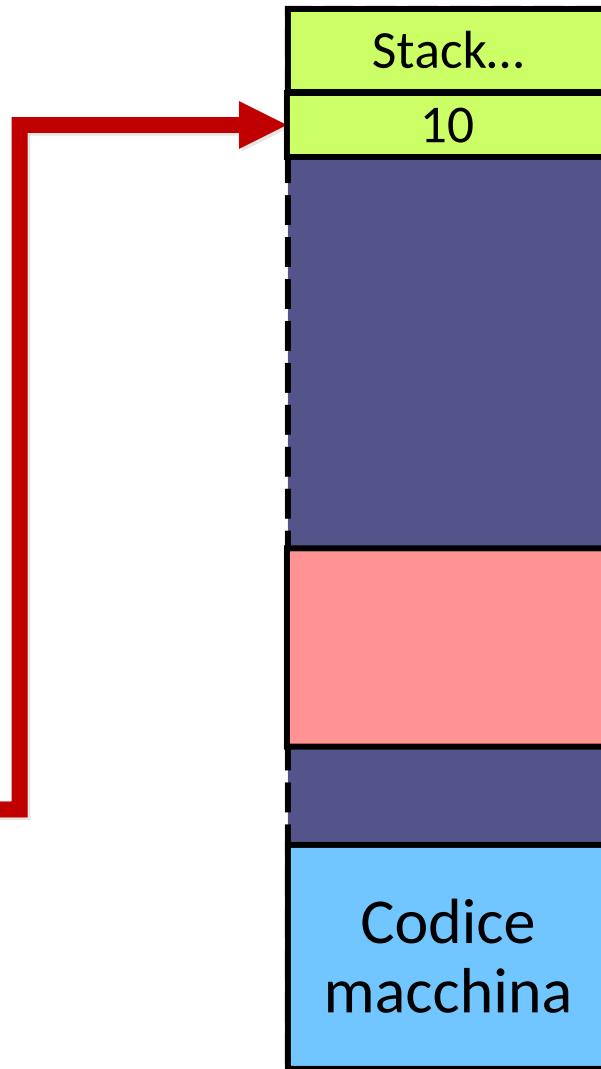
```
}
```



# 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;
}

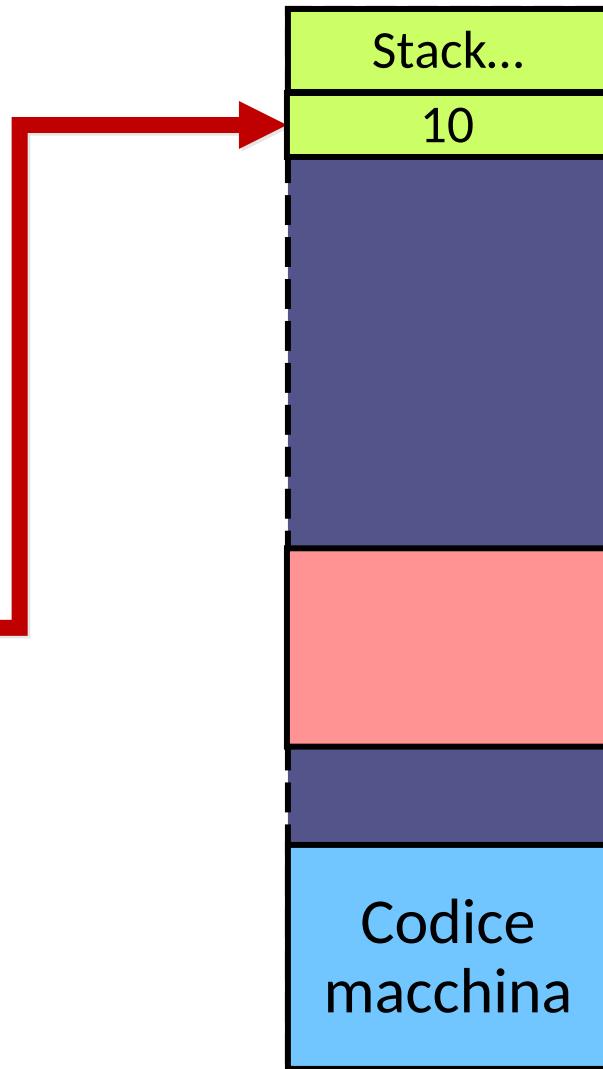
int main() {
 int i=10;
 f(&i);
 return i;
}
```



# Passaggio per indirizzo

```
void f(int* p) {
 if (p!=NULL)
 *p=27;
}

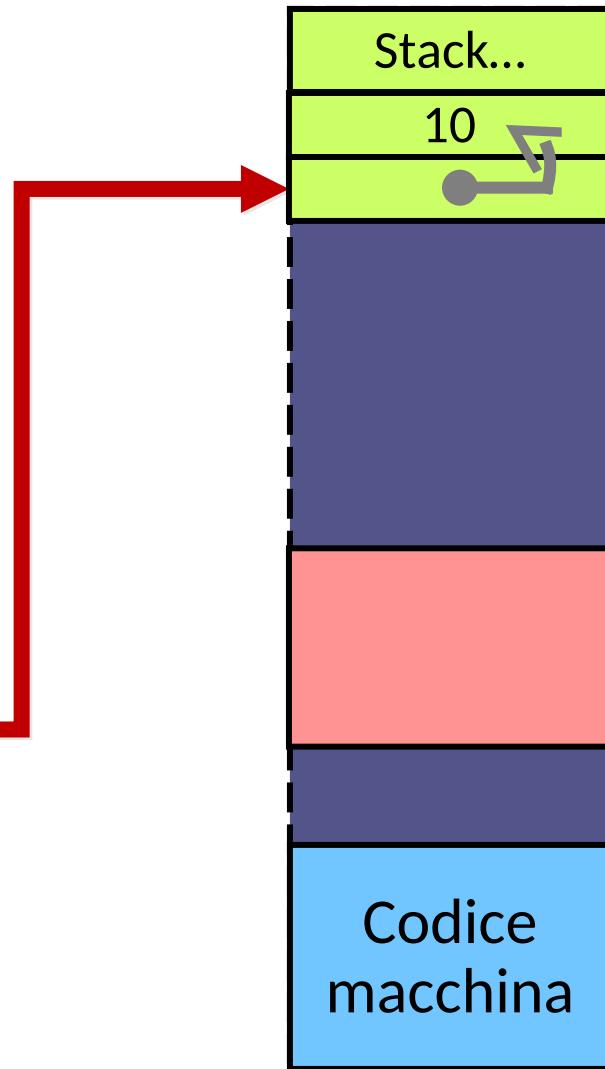
int main() {
 int i=10;
 f(&i);
 return i;
}
```



# Passaggio per indirizzo

```
void f(int* p) {
 if (p!=NULL)
 *p=27;
}

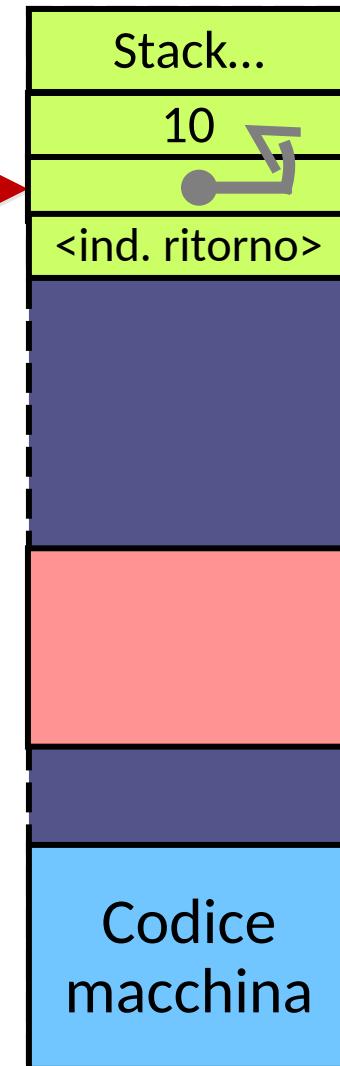
int main() {
 int i=10;
 f(&i);
 return i;
}
```



# Passaggio per indirizzo

```
void f(int* p) {
 if (p!=NULL)
 *p=27;
}

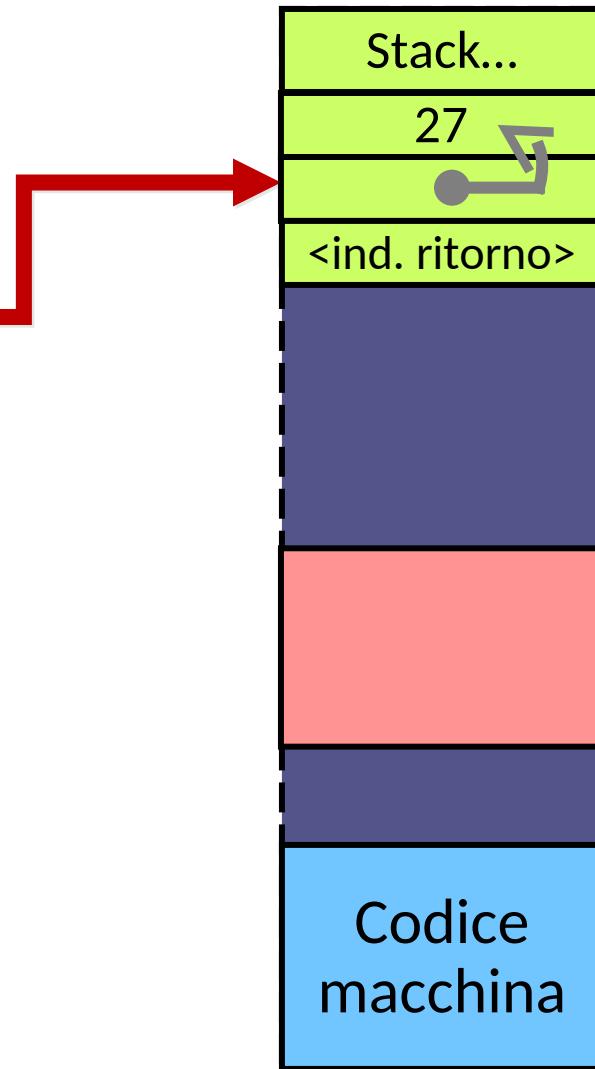
int main() {
 int i=10;
 f(&i);
 return i;
}
```



# Passaggio per indirizzo

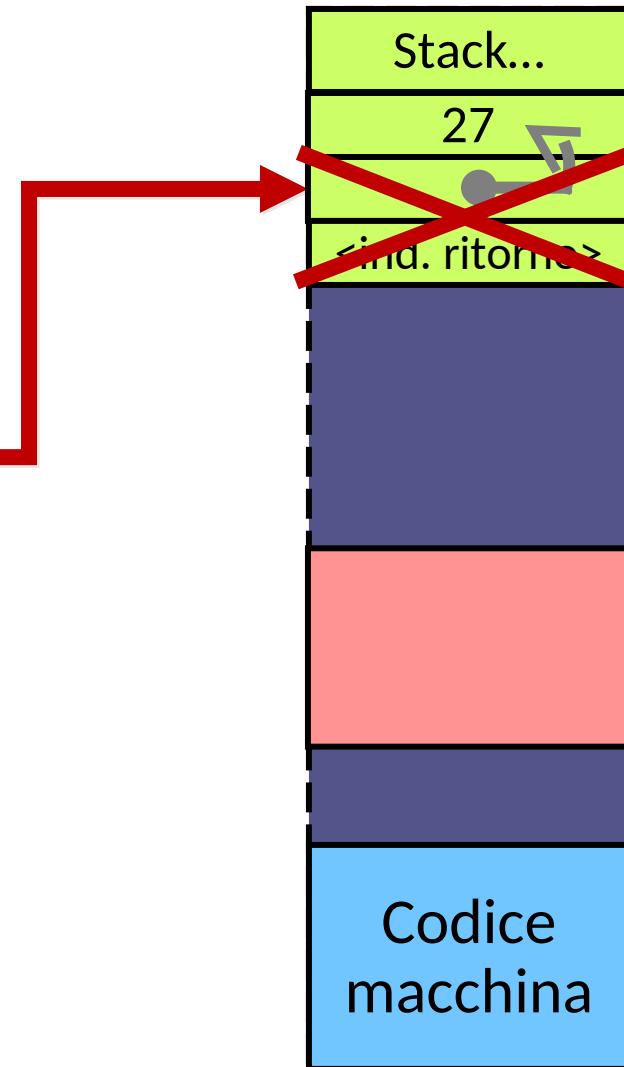
```
void f(int* p) {
 if (p!=NULL)
 *p=27;
}

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



# Passaggio per indirizzo

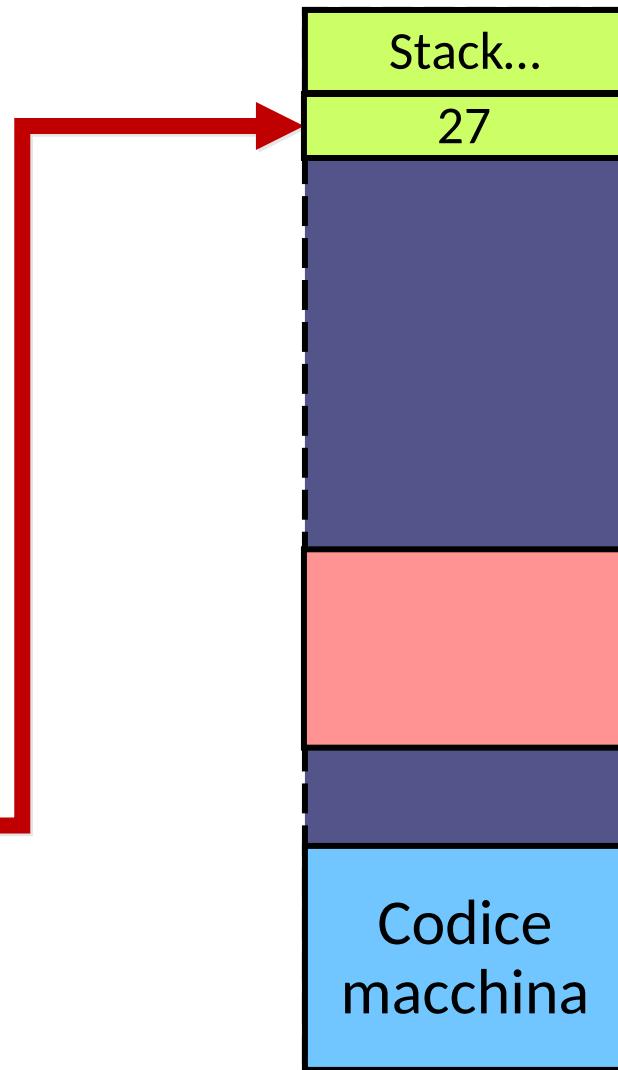
```
void f(int* p) {
 if (p!=NULL)
 *p=27;
}
int main() {
 int i=10;
 f(&i);
 return i;
}
```



# 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;
}

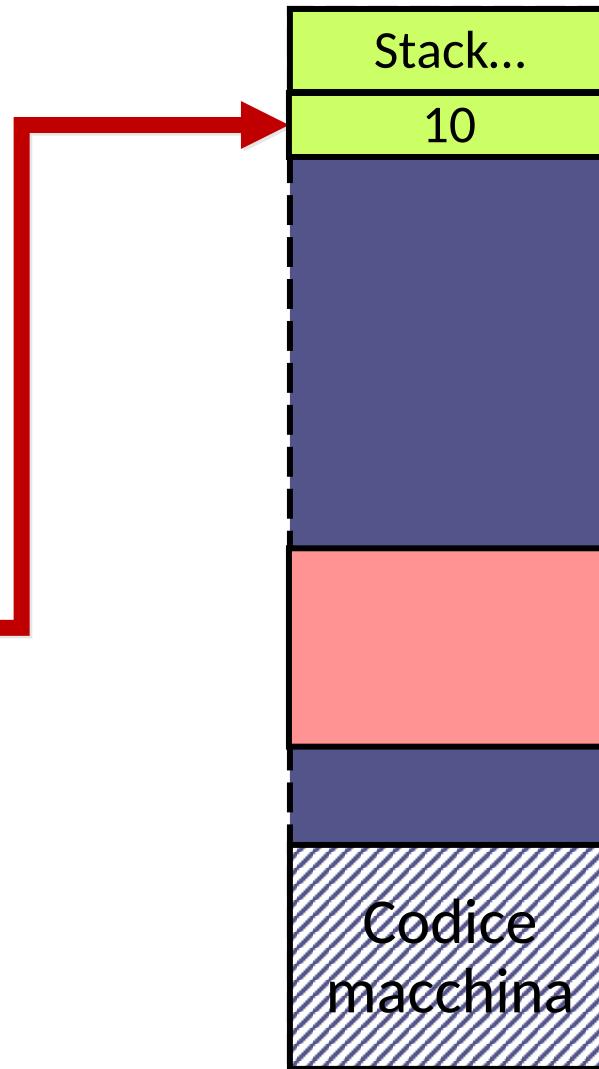
int main() {
 int i=10;
 f(i);
 return i;
};
```



# Passaggio per riferimento

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

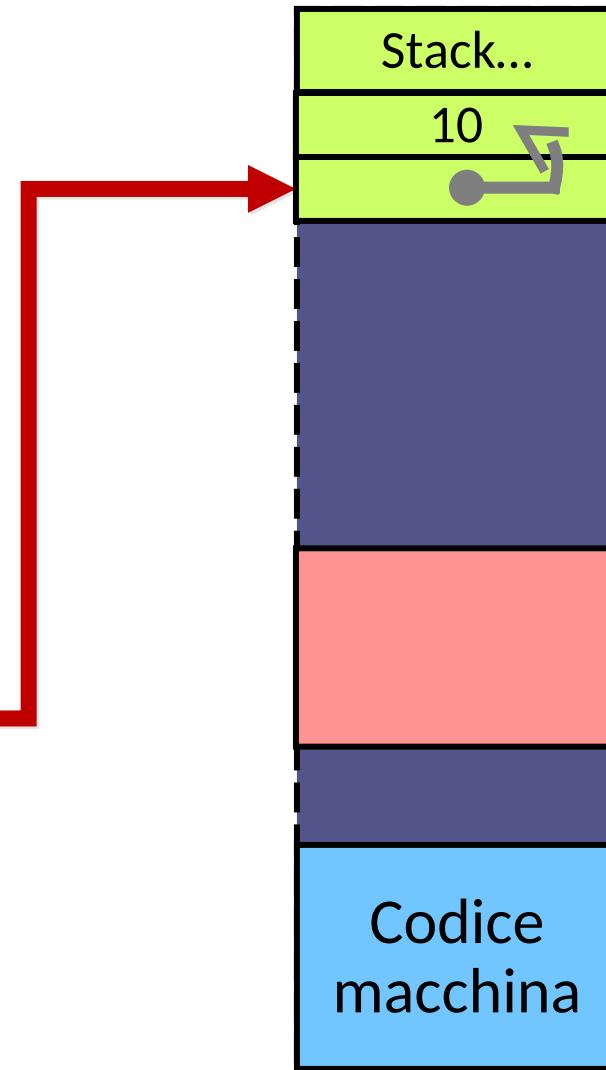
int main() {
 int i=10;
 f(i);
 return i;
};
```



# Passaggio per riferimento

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

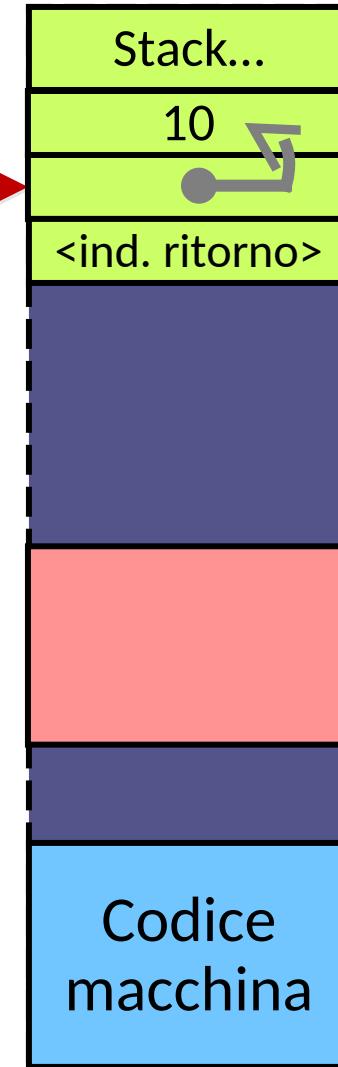
int main() {
 int i=10;
 f(i);
 return i;
};
```



# Passaggio per riferimento

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

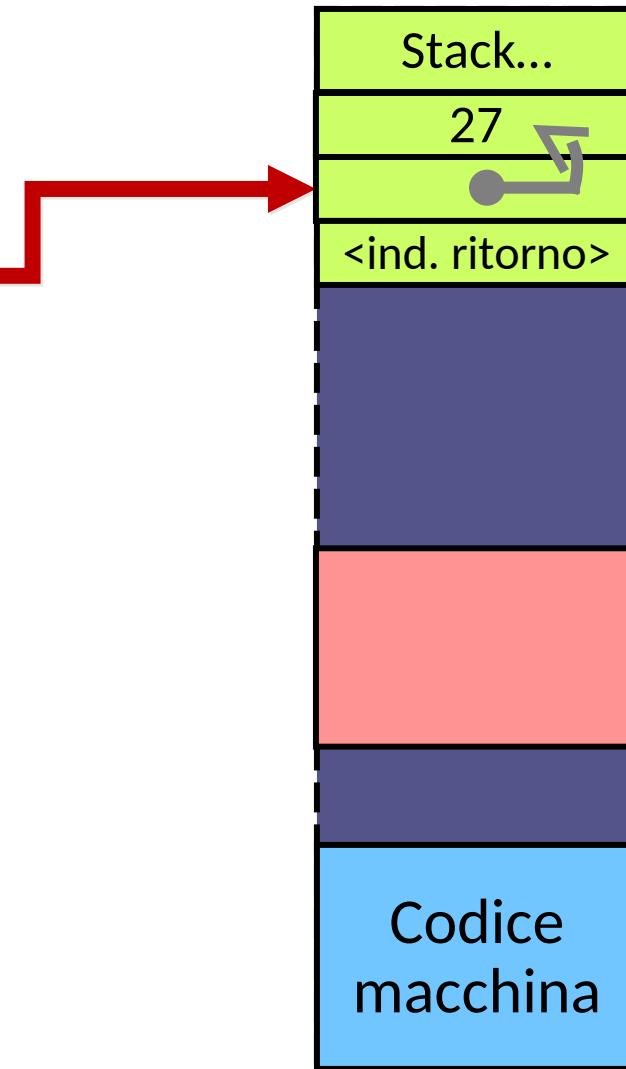
int main() {
 int i=10;
 f(i);
 return i;
};
```



# Passaggio per riferimento

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

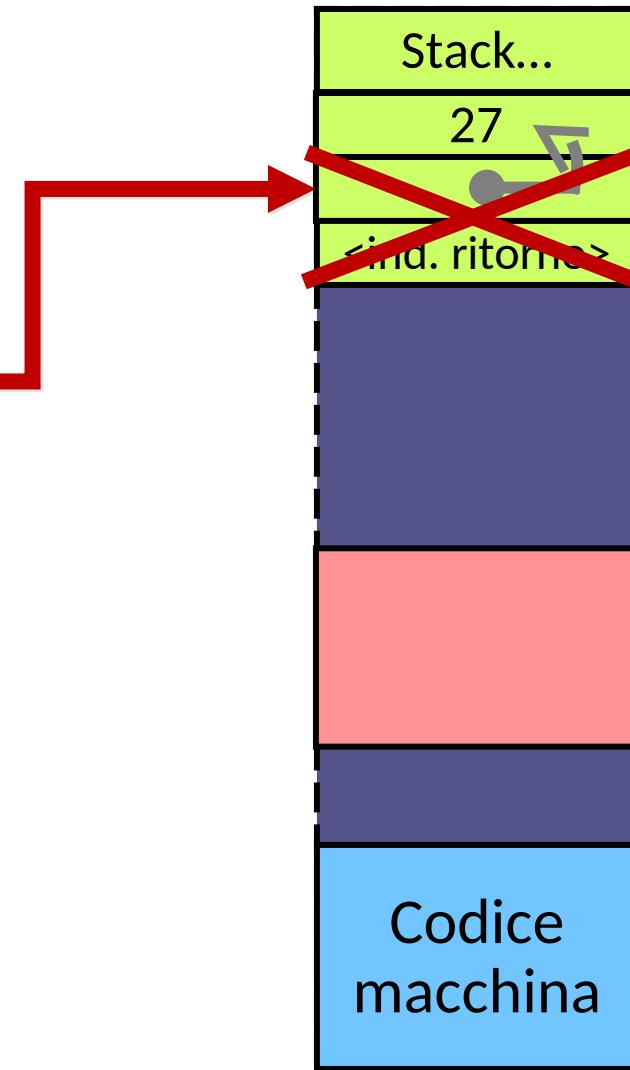
int main() {
 int i=10;
 f(i);
 return i;
};
```



# Passaggio per riferimento

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

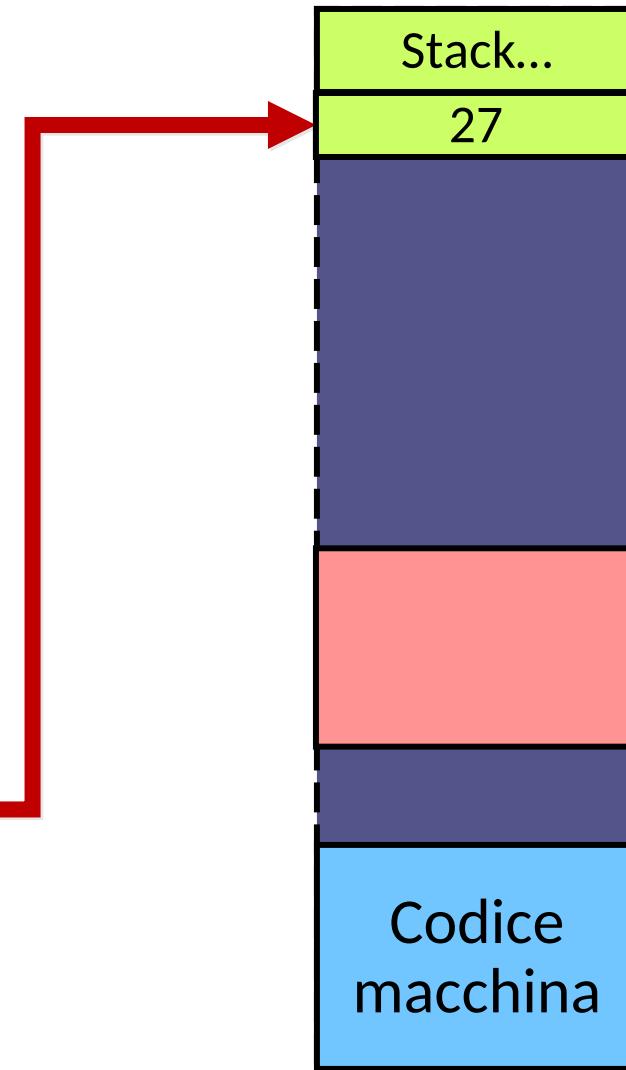
```
int main() {
 int i=10;
 f(i);
 return i;
};
```



# 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)





# Gestione delle eccezioni

Programmazione di Sistema  
A.A. 2017-18

# Argomenti

- Eccezioni
- Strategie di gestione



# Gestire le anomalie

- Non tutte le operazioni hanno sempre successo
  - Possono verificarsi errori e fallimenti di varia natura
  - Alcuni ascrivibili al comportamento dell'utente, altri al sistema
- Possibili fallimenti
  - Saturazione della memoria
  - Disco pieno
  - File assente
  - Accesso negato
  - Malfunzionamento hardware
  - Rete inaccessibile
  - ...



# Errori attesi

- Occorre verificare se siano o meno presenti
  - Gestendoli nel punto preciso in cui sono stati identificati
- Problemi
  - A volte il test viene omesso ( int scanf(char\*, ...) )
  - Altre volte, l'errore viene rilevato in una funzione che non sa come trattarlo: occorre propagarlo all'indietro
- Problemi legati alla propagazione
  - Una funzione potrebbe ritornare già un valore
  - Un costruttore non può ritornare nulla
  - Cosa capita se il chiamante di una funzione non propaga un errore?



# Errori non attesi

- Molto più difficili da gestire
  - Possono verificarsi pressoché ovunque
  - Il tentativo di gestirli rende praticamente illeggibile il codice
- È molto improbabile che sia possibile gestirli nel posto in cui si sono verificati
  - Se non si riesce ad allocare un oggetto interno ad un'implementazione mentre si invoca un metodo, cosa fare?
- Occorre un modo alternativo per poter strutturare il codice
  - Che permetta di fare fronte alle evenienze là dove si ha la possibilità di prendere una decisione opportuna



# Eccezioni

- Meccanismo che permette di trasferire il flusso di esecuzione ad un punto precedente, dove si ha la possibilità di gestirlo
  - Saltando tutto il codice intermedio
  - Evitando il rischio di dimenticarsi di propagare una indicazione di errore
- Si notifica al sistema la presenza di un'eccezione creando un dato qualsiasi
  - E passandolo come valore alla parola chiave “throw”
- Quando il sistema esegue “throw”
  - Abbandona il normale flusso di elaborazione
  - E inizia la ricerca di una contromisura



# Eccezioni

- Il tipo del dato passato a throw è arbitrario
  - Serve a descrivere quanto si è verificato
  - Il suo tipo viene utilizzato per scegliere quale contromisura adottare
- La libreria standard offre la classe std::exception
  - Può essere usata come base di una gerarchia di ereditarietà, per creare classi più specifiche
- Occorre documentare bene l'uso delle eccezioni
  - Evidenziandone la semantica



# Eccezioni

- Se un'eccezione si verifica in un blocco try...
  - ...o in un metodo chiamato dall'interno di un blocco try...
- ...si contrae lo stack fino al blocco try incluso
  - Tutte le variabili locali vengono distrutte ed eliminate



# Eccezioni

```
try {
 //le istruzioni eseguite qui possono
 //lanciare un'eccezione
}

catch (out_of_range& oor){
 //contromisura
 //per errore specifico
}

catch (exception& e) {
 //contromisura generica
}
```



# Eccezioni

- Le clausole “catch” effettuano un match basato sul tipo dell’eccezione
  - Se la classe dell’eccezione coincide o deriva da (is\_a) quella indicata, il codice corrispondente è eseguito e l’eccezione rientra
- I blocchi catch sono esaminati nell’ordine in cui sono scritti
  - Occorre ordinarli dall’eccezione più specifica alla più generica
  - Se non c’è alcuna corrispondenza, l’eccezione rimane attiva e il programma ritorna alla funzione chiamante...



# Eccezioni

- ...ed eventualmente al chiamante del chiamante...
  - Fino ad incontrare un blocco catch adatto...
  - ...o fino alla contrazione completa dello stack, con la terminazione del flusso di esecuzione



# La classe std::exception

- Incapsula una stringa, che viene inizializzata nel costruttore
  - Si accede al suo contenuto con la funzione membro what()
  - Serve per documentare l'errore, NON per creare del codice che reagisca all'eccezione

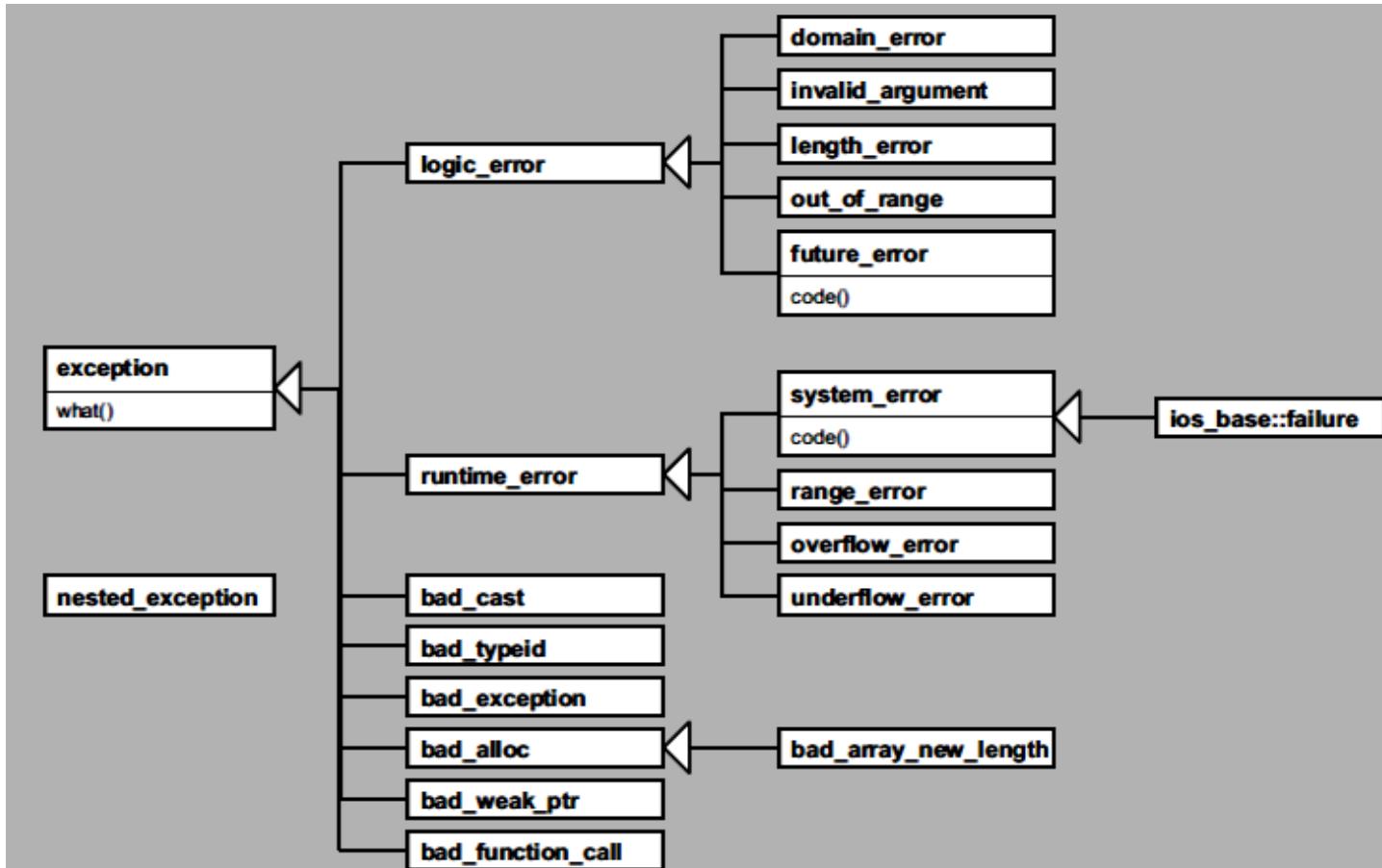


# Alcune classi derivate

- `std::logic_error` - incoerenza del codice
  - Principali sottoclassi:  
`domain_error`  
`invalid_argument`  
`length_error`  
`out_of_range`
- `std::runtime_error` - condizione inattesa
  - Sottoclassi:  
`overflow_error`,  
`range_error`  
`underflow_error`  
`system_error`



# La classe std::exception



# Dichiarare le eccezioni

- Le classi std::exception e suoi derivati sono dichiarate in diversi file
  - `#include <exception>`
  - `std::exception` e `std::bad_exception`
- `#include <stdexcept>`
  - Per la maggior parte delle classi relative agli errori logici e in fase di esecuzione
- `#include <system_error>`
  - Per gli errori di sistema (C++11)



# Dichiarare le eccezioni

- **#include <new>**
  - Per le eccezioni relative alla mancanza di memoria
- **#include <ios>**
  - Per gli errori di I/O
- **#include <future>**
  - Per gli errori relativi all'esecuzione asincrona (C++11)
- **#include <typeinfo>**
  - Per le eccezioni legate ai cast dinamici o alla RTTI



# Strategie di gestione

- Il blocco catch ha lo scopo di rimediare all'eccezione che si è verificata
  - In base al tipo di evento, occorre adottare una contromisura adeguata
- Strategie di gestione
  - Terminare, in modo ordinato, il programma
  - Ritentare l'esecuzione, evitando di entrare in un loop infinito
  - Registrare un messaggio nel log e rilanciare l'eccezione



# Terminare il programma

- Si procede a salvare lo stato del programma e rilasciare eventuali risorse globali
  - Azione estrema, di solito lasciata al livello più esterno (ma non solo)

```
try {
 //azioni a rischio
}

catch(...)
{
 // eventuali azioni di salvataggio

 //codice di errore
 exit(-1);
}
```

# Terminare il programma

- La sintassi “...” indica un’eccezione qualsiasi
- Permette l’ingresso nel blocco catch qualunque sia il tipo di dato lanciato (che però non è accessibile)



# Ritentare l'esecuzione

- Strategia adatta quando il fallimento è dovuto a cause temporanee
  - Congestione di rete
  - Mancanza di risorse (disco, memoria) se l'applicazione può rilasciarne

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



# Ritentare l'esecuzione

- L'istruzione “throw;” al termine del blocco catch rilancia l'eccezione catturata



# Registrare un messaggio

- Si usa un meccanismo opportunamente predisposto per la registrazione degli errori
  - Come il flusso “`std::cerr`”
  - Il metodo `what()` di `exception` permette di accedere ad una descrizione dell’errore
  - L’istruzione `throw` fa sì che quando l’eccezione venga

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



# Cosa non fare

- Scrivere un blocco catch che non esegue nessuna strategia di riallineamento
  - E lascia che il programma, la cui esecuzione è parzialmente fallita, prosegua
- Stampare un messaggio di errore e poi continuare, in genere, non è una soluzione

```
try {
 //...
} catch (std::exception& e) {
 std::cerr<<"Errore:"<<e.what()<<"\n";
}
```



# Contrazione dello stack

- Il fatto che lo stack venga contratto in fase di lancio di un'eccezione è alla base di un pattern di programmazione tipico del C++
  - RAI – Resource Acquisition Is Initialization
- Aiuta a garantire che le risorse acquisite alla costruzione siano liberate
  - Può essere usato per eseguire azioni anche in presenza di eccezioni
  - Facendo attenzione a non sollevarne delle altre
- È l'unica alternativa al blocco “finally” di Java, che in C++ non esiste



```
class ActionTimer {
 long start;
 std::string msg;
public:
 ActionTimer(std::string tag):
 msg(tag){
 start=GetCurrentTime();
 }
 ~ActionTimer() {
 long t=GetCurrentTime()-start;
 std::cout<<msg<<":"<<t<<"\n";
 }
};

{
 ActionTimer at("b1");
 for(int i=0; i<8; i++){
 ActionTimer at2("b2");
 //do something...
 throw std::exception;
 }
}
```



# I costi delle eccezioni

- Se nessuna eccezione viene lanciata, non ci sono sostanziali penalità
  - La definizione di un blocco try si limita ad inserire alcune informazioni nello stack
- Se viene lanciata un'eccezione il costo è abbastanza elevato
  - Almeno un ordine di grandezza maggiore dell'invocazione di una funzione ...
  - ... a cui si aggiunge il costo dell'esecuzione dei vari distruttori
- Non è un meccanismo conveniente per gestire logiche locali
  - Un costrutto “if” è molto meno oneroso



# Spunti di riflessione

- Si scriva una classe dotata di costruttore e distruttore che incapsuli un valore
- Si verifichi con il debugger il funzionamento del codice seguente:

```
class MyClass; //Da definirsi

int main(int argc, char** argv){
 MyClass c1(1);
 try {
 MyClass c2(2);
 throw "errore";
 } catch (...) {
 throw;
 }
}
```





# Composizione di oggetti

Programmazione di Sistema

A.A. 2017-18

# Argomenti

- Oggetti composti
- Duplicazione degli oggetti
- Copia e movimento



# Oggetti composti

- Un oggetto può contenere altri oggetti
  - Come parte integrante della propria struttura dati
  - Oppure facendo riferimento ad essi tramite puntatori
- In entrambi i casi, è compito del costruttore inizializzare opportunamente tali oggetti
  - La sintassi è differente
  - La disposizione in memoria anche!



# Oggetti composti

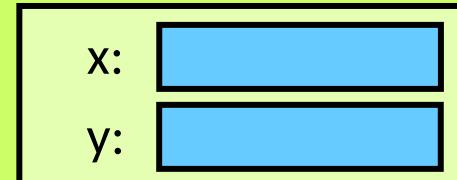
```
class CPoint {
public:
 CPoint(int x, int y);
private:
 int x, y;
};

class CRect {
public:
 CRect(int x1, int y1,
 int x2, int y2);
private:
 CPoint p1,p2;
};
```

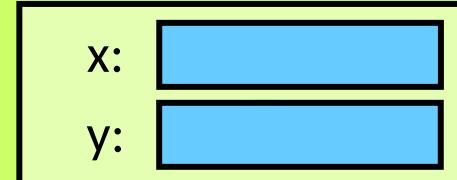
geom1.h



CPoint



p1:



p2:

CRect

# Oggetti composti

```
#include "geom1.h"
```

```
CPoint::CPoint(int x,
 int y) {

 this->x = x;

 this->y = y;
}
```

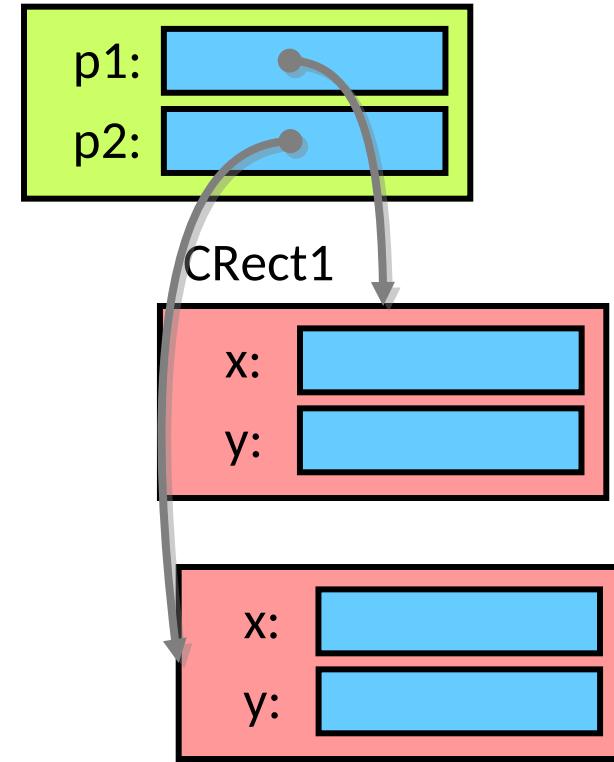
```
CRect::CRect(int x1, int y1,
 int x2, int y2) :
 p1(x1,y1),
 p2(x2,y2) {
}
```



# Oggetti composti

```
class CRect1 {
public:
 CRect1(int x1,
 int y1,
 int x2,
 int y2);
 ~CRect1();
private:
 CPoint *p1;
 CPoint *p2;
};
```

geom2.h



# Oggetti composti

```
CRect1::CRect1(int x1, int y1,
 int x2, int y2) {
 p1= new CPoint(x1,y1);
 p2= new CPoint(x2,y2);
}

CRect1:: ~CRect1(){
 delete p1; //p1=NULL;
 delete p2; //p2=NULL;
}
```



# Copia ed assegnazione

- Talora occorre copiare il contenuto di un oggetto in un altro oggetto
  - Esplicitamente, quando si esegue un'assegnazione
  - Implicitamente, quando si passa un oggetto per valore o si ritorna un oggetto
- Se l'oggetto destinazione esisteva prima della copia si parla di assegnazione
- Se, invece, viene creato al momento della copia, si parla di costruzione di copia



# Copia e assegnazione

```
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
```

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

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



# Copia e assegnazione

- Di base, il compilatore ricopia il contenuto di tutte le variabili istanza dell'oggetto sorgente nell'oggetto destinazione
  - Se si tratta di puntatori, questo comportamento può essere problematico



# Esempio

```
class CBuffer {
 int size;
 char* ptr;
public:
 CBuffer(int size);
 ~CBuffer();
};

CBuffer b1(20), b2(10);

b1 = b2;
```

```
CBuffer::
CBuffer(int size) {
 this->size=size;
 ptr=new char[size];
}

Cbuffer::
~CBuffer() {
 delete[] ptr;
}
```

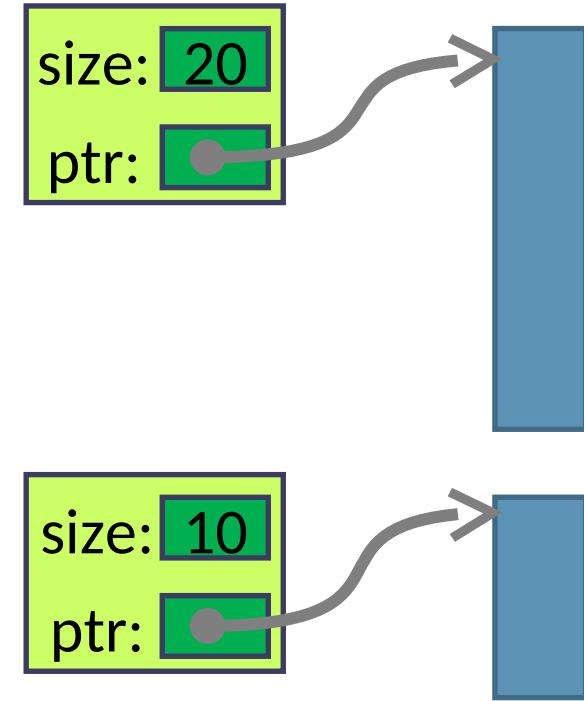


# Esempio

```
class CBuffer {
 int size;
 char* ptr;
public:
 CBuffer(int size);
 ~CBuffer();
};

CBuffer b1(20), b2(10);

b1 = b2;
```

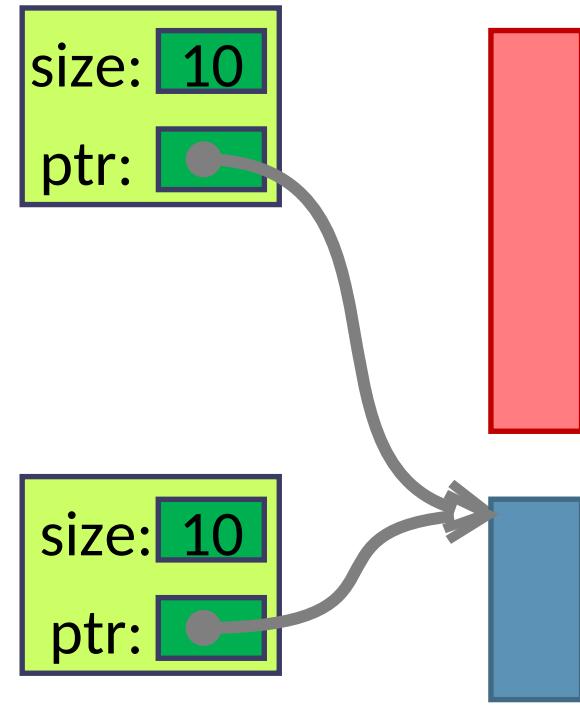


# Esempio

```
class CBuffer {
 int size;
 char* ptr;
public:
 CBuffer(int size);
 ~CBuffer();
};

CBuffer b1(20), b2(10);

b1 = b2;
```



# Copiare i puntatori

- Ogni volta che si deve copiare un puntatore sorge un'ambiguità
  - Basta duplicare il puntatore facendo riferimento allo stesso indirizzo?
  - Oppure occorre duplicare il blocco puntato?
  - Se l'oggetto destinazione esisteva già, che fine fa il vecchio puntatore?
  - Il compilatore, di base, sceglie la prima strategia
  - Il programmatore può modificarla, definendo le proprie procedure



# Costruttore di copia

- Indica al compilatore come comportarsi quando deve inizializzare un nuovo oggetto a partire da un originale

```
class CBuffer {
 CBuffer(const CBuffer& source);
};
```

# Costruttore di copia

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



# Operatore di assegnazione

- Indica al compilatore come comportarsi quando si assegna un nuovo valore ad un oggetto già esistente

◦ Che ha risorse da liberare  
CBuffer& operator=(**const** CBuffer&  
source );

# Operatore di assegnazione

- Distrugge il contenuto dell'oggetto destinazione
  - Prima di inizializzarlo con la copia dell'oggetto sorgente
- Restituisce un riferimento all'oggetto destinazione



# Operatore di assegnazione

```
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;
}
```



# Operatore di assegnazione

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

Se questa riga viene omessa...

...e si assegna un  
oggetto a se stesso, i dati contenuti  
vengono distrutti

# Operatore di assegnazione

```
CBuffer& operator=(const CBuffer&
 source) {

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

...in caso di eccezione, il distruttore rilascerebbe due volte la memoria



# Operatore di assegnazione

Questa è la fonte dei guai...  
...Se l'oggetto è grosso, e non c'è la memoria  
richiesta, lancia un'eccezione

```
this->ptr=new char[size];
```

```
 memcpy(this->ptr, source.ptr, size);
}
return *this;
}
```



# Equivalenza semantica

- Costruttore di copia ed operatore di assegnazione devono essere semanticamente equivalenti

```
CBuffer b1(5);
Cbuffer b2(7); //b2 viene creato
b2=b1; //e poi sovrascritto
CBuffer b3(b1); //b3 e b2 devono avere
//lo stesso contenuto
```

# Generazione automatica

- Di base, il compilatore inserisce automaticamente sia il costruttore di copia che l'operatore di assegnazione
  - Copiando o assegnando in modo ricorsivo tutti i membri della classe
- Si può impedire la duplicazione di un oggetto dichiarando privati sia il costruttore di copia che l'operatore di assegnazione
  - Se ne impedisce l'utilizzo da parte di codice esterno



# Generazione automatica

```
class CBuffer {
 //...
private:
 CBuffer(const CBuffer&);
 CBuffer& operator=(const CBuffer&);
};

CBuffer b1(10), b2(5);
b1=b2; //Errore di compilazione
CBuffer b3(b1); //idem
```



# Assegnazione e risorse

- L'operatore di assegnazione deve rilasciare tutte le risorse possedute
  - Per non creare leakage
  - Si sovrappone al distruttore
  - Occorre fare attenzione in caso di manutenzione!

# La regola dei tre

- Se una classe dispone di una qualunque di queste funzioni membro, occorre implementare le altre due
  - Costruttore di copia
  - Operatore di assegnazione
  - Distruttore
- In mancanza di ciò, il compilatore fornirà la propria implementazione
  - La quale, per lo più, non potrà essere corretta



# Copia e movimento

- Copiare un oggetto in un altro può essere un'operazione dispendiosa
  - In alcuni casi, lo si vuole evitare esplicitamente
  - Garantendo il controllo centralizzato delle risorse contenute nell'oggetto



# Movimento

- Accanto all'operazione di copia, la specifica C++ 2011 introduce il concetto di movimento
  - «Svuotare» un oggetto che sta per essere distrutto del suo contenuto e «travararlo» in un altro oggetto
- Candidati al movimento
  - Le variabili locali al termine del blocco in cui sono state definite
  - I risultati delle espressioni temporanee
  - Gli oggetti anonimi costruiti a partire dal tipo

```
std::string("ciao");
```
  - Tutto ciò che non ha un nome e può comparire solo a destra di "=" nelle assegnazioni (RVALUE)



# Costruttore di movimento

- Nelle classi in cui è utile, si può definire il costruttore di movimento
  - `TypeName(TypeName&& source);`
  - L'operatore `&&` indica un candidato al movimento del tipo che lo precede
  - RValue reference
- Il suo compito è «smontatare» l'originale e trasferire il suo contenuto nell'oggetto destinazione
  - L'originale verrà modificato, ma poiché sta per essere distrutto, questo non ha importanza
  - A patto che i "resti" dell'oggetto smontato possano essere distrutti senza creare danni



# Costruttore di movimento

```
class CBuffer {
 int size;
 char* ptr;
public:
 CBuffer(CBuffer&& source) {
 this->size=source.size;
 this->ptr=source.ptr;
 source.ptr=NULL;
 }
};
```



Risparmia l'operazione,  
onerosa, di allocare un  
nuovo blocco e inizializzarlo  
con il contenuto della stringa  
di partenza.  
**MODIFICA L'ORIGINALE!**



# Costruttore di movimento

- A differenza di quello di copia, il costruttore di movimento non è generato automaticamente dal compilatore
  - È compito del programmatore scegliere se e dove implementarlo
- Ogni volta in cui occorre copiare un valore, il compilatore valuta se effettuare una copia o un movimento del dato
  - A patto che sia disponibile il costruttore di movimento relativo



# Semantica del movimento

- Se si applica il movimento, il compilatore genera le seguenti pseudo-chiamate
  - Obj\_MoveConstructor(dst,src);
  - Obj\_Destructor(src);
- Occorre fare in modo che la chiamata al distruttore non elimini le risorse spostate

# Utilità del movimento

- Riduce, quando possibile, il costo del passaggio per valore
  - Sia nei parametri che nei dati che vengono restituiti da una funzione

```
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
}
```



# Operatore di movimento

- In alcuni casi (ad es., `std::unique_ptr`) la copia non è possibile
  - Il movimento, invece, sì
  - Rende possibile creare una funzione che ritorna un oggetto di tipo `unique_ptr`, senza violarne le regole
  - Se un oggetto contiene solo valori elementari o tipi valore, il movimento equivale alla copia



# Operatore di movimento

- Se i tipi valore contenuti, rappresentano una risorsa esterna, il vantaggio può essere grande
  - Descrittori di file, socket, connessioni a base dati, ...
  - L'alternativa sarebbe acquisire una seconda copia della risorsa e rilasciare la risorsa originale subito dopo

# Usi nella libreria standard

- Le classi della libreria standard sono state riviste per supportare la semantica del movimento
  - Aumentandone di molto l'efficienza e rendendole utili in applicazioni a basso livello



# Assegnazione per movimento

- Analogamente a quanto avviene per l'operatore di assegnazione semplice...
  - ...occorre dapprima liberare le risorse esistenti e poi trasferire il contenuto

```
dell'oggetto sorgente
CBuffer& operator=(CBuffer&& source) {
 if (this != &source) {
 delete[] this->ptr;
 this->size=source.size;
 this->ptr=source.ptr;
 source.ptr=NULL;
 }
 return *this;
}
```

# Il paradigma Copy&Swap (1)

- Il meccanismo di assegnazione (sia normale che per movimento) è, in generale, fonte di problemi
  - Unisce le operazioni di distruzione e di copia
  - In caso di eccezioni può generare mostri

```
class intArray {
 std::size_t mSize;
 int* mArray;
public:
 intArray(std::size_t size=0): //costruttore
 mSize(size), mArray(mSize? new int[mSize]: NULL) {}

 intArray(const intArray& that): //costruttore di copia
 mSize(that.mSize), mArray(mSize? new int[mSize]:NULL)
{
 std::copy(that.mArray, that.mArray+mSize, mArray);
 }
 ~intArray() { delete [] mArray; } //distruttore

 //segue...
```

# Il paradigma Copy&Swap

## (2)

E facile scrivere implementazioni errate dell'operatore di assegnazione

- Specialmente rispetto a possibili eccezioni
- Se si omette (1)
  - Si copierebbe un oggetto su se stesso, distruggendo l'array sorgente su cui si andrebbe poi a leggere
- Se si omette (2)
  - In caso di eccezione successiva, il distruttore potrebbe rilasciare due volte la memoria
- (3) è fonte di guai
  - Se l'oggetto è grosso, potrebbe non esserci la memoria richiesta
  - new[] lancia un'eccezione e l'esecuzione si interrompe lasciando un oggetto corrotto

```
intArray& operator=(
 const intArray& that) {

 if (this!=&that) { // (1)
 delete mArray;
 mArray=NULL; // (2)
 mSize=that.mSize;
 mArray = new int[mSize]; // (3)
 std::copy(that.mArray,
 that.mArray+mSize,
 mArray);
 }
 return *this;
}
```

- Occorre riscrivere la funzione in modo da evitare il rischio che le eccezioni possono introdurre

# Il paradigma Copy&Swap

## (3)

- Si introduce la funzione swap(...)
  - Definendola come friend della classe
  - Si occupa di scambiare il contenuto delle risorse tra due istanze
- Si appoggia sulla funzione std::swap
  - Per scambiare i riferimenti elementari contenuti sugli oggetti
- Si definisce l'operatore di assegnazione con un parametro di tipo valore
  - La logica dell'operatore di copia provvederà a fare un duplicato nel parametro that
  - In caso di eccezione, \*this non viene modificato (l'eccezione si verifica nel tentativo di creare la copia, prima di effettuare uno swap)
  - Si evita il test di identità tra sorgente e destinazione
- Swap permette anche di implementare il costruttore di movimento
  - Con le stesse garanzie

```
friend void swap(intArray& a,
 intArray& b) {

 std::swap(a.mSize, b.mSize);
 std::swap(a.mArray, b.mArray);
}

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);
}
```

# La funzione std::move(...)

- Funzione di supporto per trasformare un oggetto generico in un riferimento RValue, così da poterlo utilizzare nella costruzione o assegnazione per movimento
  - Definita nel file <utility>
  - Forza l'oggetto passato come parametro ad essere considerato come un riferimento RValue rendendolo disponibile per usi ulteriori (viene eseguito uno static\_cast<T&&>(t) )
  - Aiuta a rendere esplicita la conversione, quando si vuole forzare l'uso del movimento rispetto alla copia

```
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 =
""
```



# Spunti di riflessione

- Si implementi la classe MyString, che incapsula un'array di caratteri allocato dinamicamente
- Si creino i costruttori di copia e movimento e i corrispondenti operatori di assegnazione





# Ereditarietà e polimorfismo

Programmazione di Sistema  
A.A. 2017-18

# Argomenti

- Ereditarietà
- Polimorfismo
- Conversione tra tipi



# Ereditarietà

- A volte, classi diverse presentano comportamenti simili
  - In quanto modellano concetti semanticamente simili
- Sebbene si possa duplicare il codice condiviso, questa raramente è una buona idea
  - Il codice diventa difficile da leggere e la manutenzione praticamente impossibile
- Una classe può essere definita come specializzazione di una classe esistente
  - Ereditandone variabili istanza e funzioni membro
- La classe così definita viene detta sotto-classe
  - Quella da cui deriva viene detta super-classe



# Ereditarietà

- La sotto-classe specializza il comportamento della super-classe
  - Aggiungendo ulteriori variabili istanza e funzioni membro

```
class File {
 int fileDescriptor;

public:
 uint_8 read ();

 size_t readBlock(
 uint_8 *ptr,
 size_t offset,
 size_t count);

 int close();
};
```

```
class TextFile: public
File {

 CharCodec codec;

public:
 wchar_t readChar();

 size_t
 readCharBlock(
 wchar_t *ptr,
 size_t offset,
 size_t count);
};
```

# Ereditarietà multipla

- In C++, una classe può ereditare da una o più classi
  - Specializzando così il concetto che esse rappresentano
  - Non esiste una classe antenata radice dalla quale tutte le classi sono derivate
- L'ereditarietà può essere
  - Pubblica
  - Privata
  - Protetta
- Se occorre, i metodi della classe di base possono essere invocati usando l'operatore “`::`” preceduto dal nome della classe base



# Ereditarietà

```
class Base {
public:
 Base();
 void baz();
 ~Base() {}
};
class Der : public Base {
public:
 Der():Base(){
 Base::baz()
 }
 ~Der() {}
};
//...
{
 Der *obj = new Der();
 obj->baz();
 delete obj;
}
```

**Attenzione!** In C++ non esiste la parola chiave “super”, non funzionerebbe con ereditarietà multipla

# Polimorfismo

- Se classe A estende in modo pubblico la classe B, è lecito assegnare ad una variabile “v” di tipo B\* un puntatore ad un oggetto di tipo A
  - A <<is\_a>> B
- Se la classe A ridefinisce il metodo “m()”, cosa succede quando si invoca v->m() ?
  - Dipende da come è stato dichiarato il metodo nella classe base



# Polimorfismo

```
class B {
public:
 int m() { return 1; }
};

class A: public B {
public:
 int m() { return 2; }
};

int main(int argc, char** argv) {
 B* ptr = new A();
 return ptr->m(); // ???
}
```



# Polimorfismo

- Per default, il C++ utilizza l'implementazione definita nella classe a cui ritiene appartenga l'oggetto
- Nel caso precedente, `ptr` ha come tipo `B*`
  - Per cui il compilatore invocherà la definizione di `m()` contenuta nella classe `B`
  - Anche se, di fatto, l'oggetto cui `ptr` punta è di classe `A`
- Per modificare questo comportamento, occorre anteporre alla definizione del metodo  
**la parola chiave `virtual`**
  - Abilitando così il funzionamento polimorfico



# Metodi virtuali

- Solo le funzioni denominate “virtual” sono polimorfiche
- Le funzioni virtuali astratte sono dichiarate  
“ = 0; ”
- Una classe astratta contiene almeno una funzione virtuale astratta
  - Una classe puramente astratta contiene solo funzioni virtuali astratte
  - Equivalente ad una interfaccia Java



# Metodi virtuali

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

class Der : public Base {
public:
 Der() : Base() { ... }
 void foo(){
 Base::foo();
 }
 void bar() { ... }
 ~Der() {}
};
```

## Attenzione!

In Java,  
al contrario, i  
metodi non  
polimorfici vanno  
esplicitamente  
dichiarati “final”



# Distruttori virtuali

- Anche i distruttori non sono polimorfici per default
- È opportuno che tutte le classi con funzioni virtuali abbiano il distruttore dichiarato virtual
- Se una classe con metodi virtuali non ha un distruttore virtuale
  - è possibile che venga chiamato il distruttore errato



# Distruttori virtuali

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

class Der : public Base {
public:
 Der() : Base() { ... }
 void foo(){
 Base::foo();
 }
 void bar() { ... }
 ~Der() { ... }
};
```



# Polimorfismo

```
class Base {
 int v;
 virtual int f() = 0;
};
```

```
class Der1:Base {
 int f() {
 return 1;
 }
};
```

```
class Der2:Base {
 int f() {
 return 2;
 }
};
```

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

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

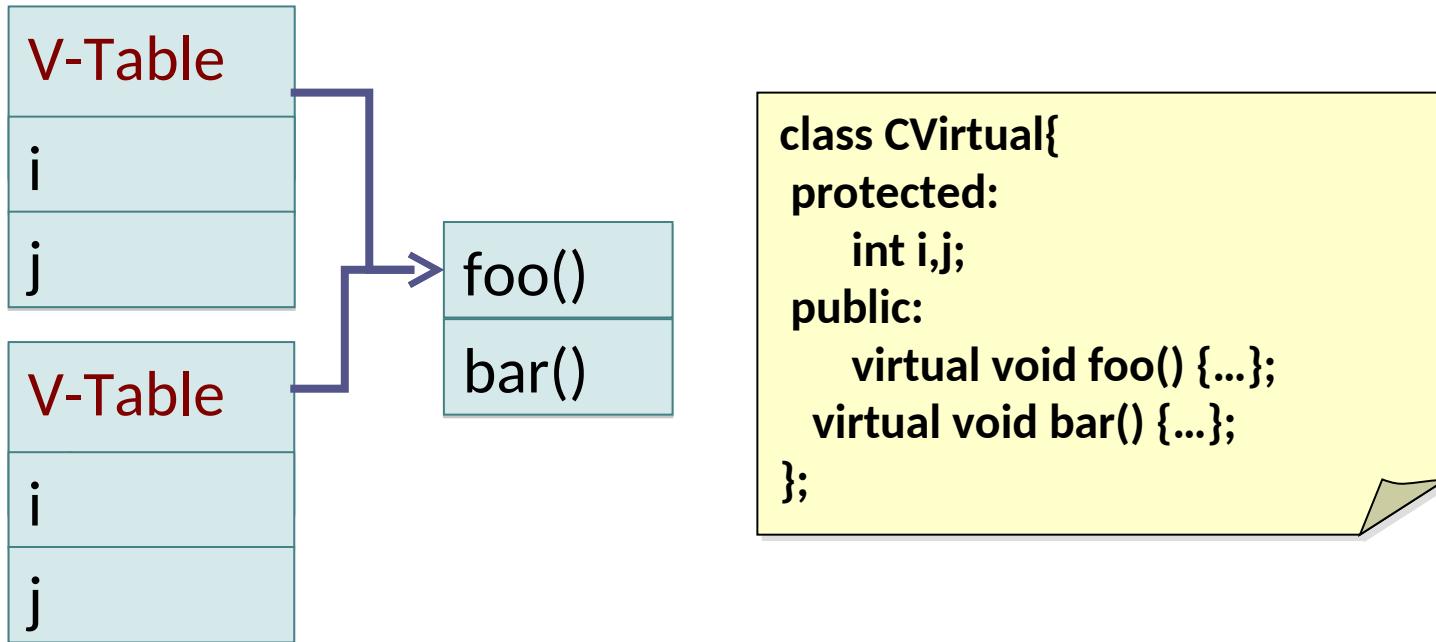
# Metodi virtuali e V-Table

- Dato che b1 e b2 hanno lo stesso tipo (Base\*), come fa il compilatore a decidere quale implementazione del metodo f() invocare?
- Per i metodi virtuali, si introduce un livello di indirezione attraverso un campo nascosto detto V-Table
- Ogni volta che viene creato un oggetto, nella memoria allocata si aggiunge un puntatore
  - Punta ad una tabella (statica) che contiene tante righe quanti sono i metodi virtuali dell'oggetto creato
  - Ciascuna riga viene riempita con il puntatore all'effettivo metodo virtuale



# Metodi virtuali e V-Table

- Se esistono più istanze di una data classe, queste condividono la stessa V-Table



# Metodi virtuali e V-Table

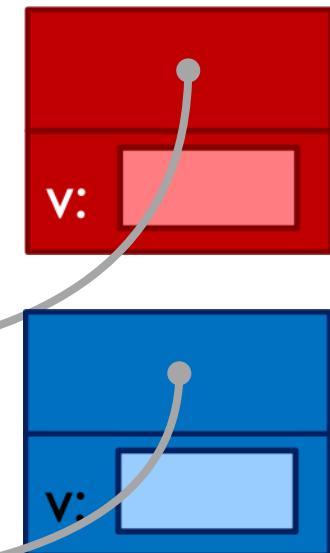
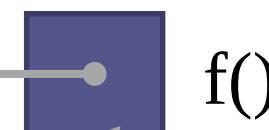
```
Base* b1= new Der1();
Base* b2= new Der2();

printf("%d\n", b1-
>f());
printf("%d\n", b2-
>f());
{ return
```

```
1; }
```

```
{ return
```

```
2; }
```



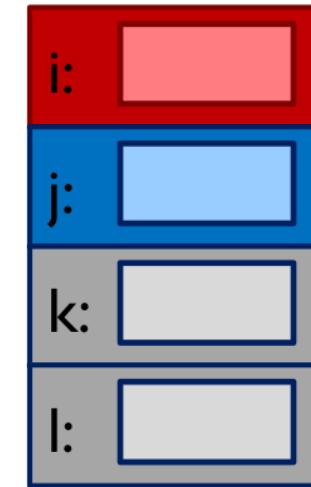
# Ereditarietà Multipla

- Una classe può ereditare da più di una classe
  - La sua interfaccia risulterà l'unione dei metodi contenuti nelle rispettive interfacce delle classi base uniti ai metodi propri della classe derivata
- Una classe può ereditare da più di una classe
  - Occorre evitare di derivare più volte dalla stessa classe



# Ereditarietà Multipla

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



# Operatori per Type Cast

- C++ supporta il type-cast in stile “C”
  - L'ereditarietà multipla e virtuale ne rendono pericoloso l'utilizzo
- C++ fornisce una serie di operatori per il type cast più efficienti e sicuri
  - `static_cast<T>`
  - `dynamic_cast<T>`
  - `const_cast<T>`
  - `reinterpret_cast<T>`



# **static\_cast<T>(p)**

```
class Base1;
class Base2;
Class Derivata:
 public Base1,
 public Base2 { };

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

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

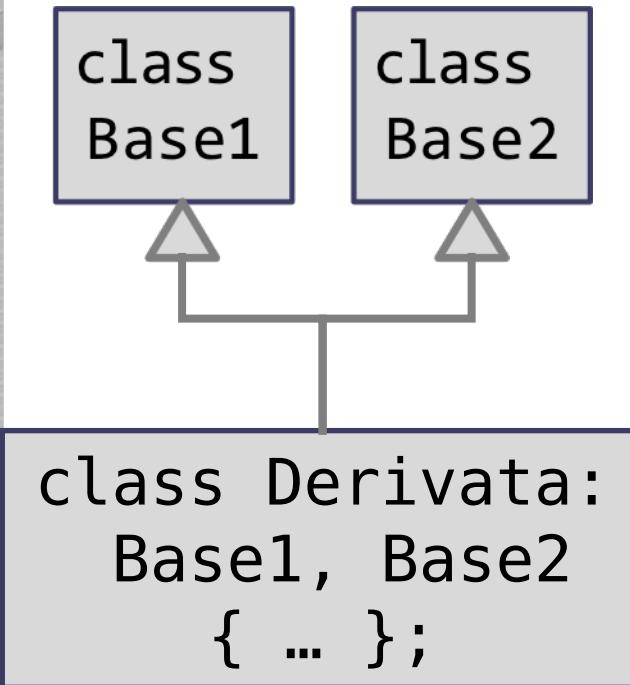


# `static_cast<T>(p)`

- Converte il valore di “p” rendendolo di tipo “T”
  - Se esiste un meccanismo di conversione disponibile
- Il meccanismo viene scelto in base al tipo di “p” come noto al compilatore
- Nel caso di puntatori, questo può essere diverso dal tipo effettivo di “p”
  - Non effettua controllo di compatibilità a run-time
- Se la conversione è illecita, il risultato non è predicibile
- Permette principalmente conversioni “in verticale” lungo l’asse ereditario
  - Sono possibili altre conversioni, se esistono esplicativi operatori



# **static\_cast<T>(p)**



```
Derivata *d=new Derivata();
```

```
Base1 *b1;
```

```
Base2 *b2;
```

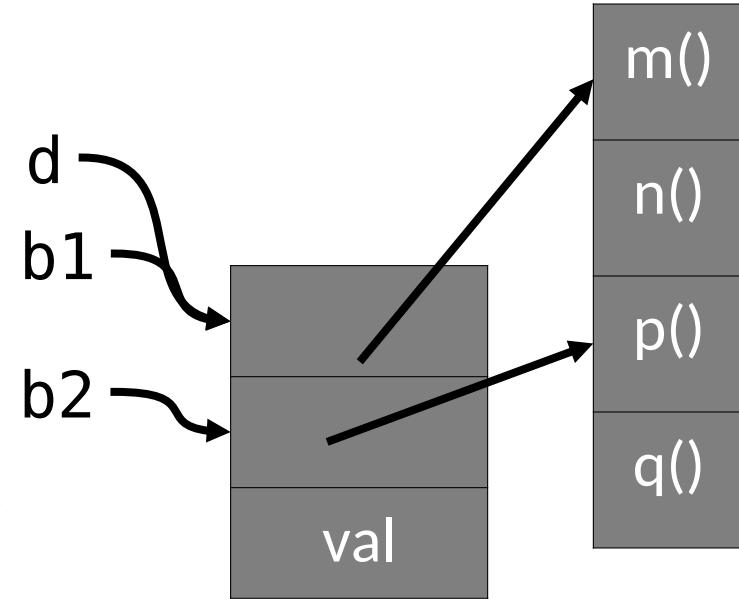
```
b1= static_cast<Base1 *>d;
```

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

# **static\_cast<T>(p)**

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

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

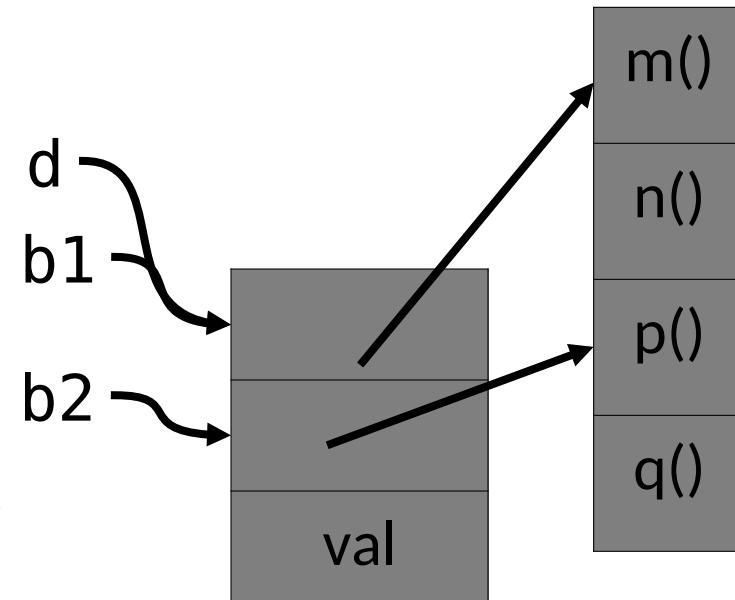


**Attenzione!** Anche se `b1==d` non  
è possibile scrivere  
`b2=static_cast<base2*>(b1)`

# **static\_cast<T>(p)**

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

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

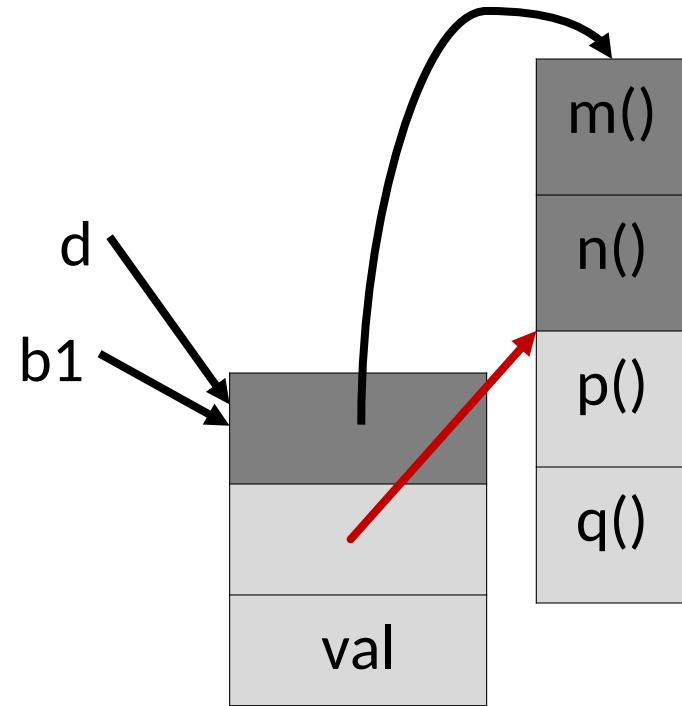


**Attenzione!** Anche se `b1==d` non  
è possibile scrivere  
`b2=static_cast<base2*>(b1)`

# **static\_cast<T>(p)**

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

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



Il compilatore non se ne accorge,  
ma in fase di esecuzione sorgono  
problemi

# dynamic\_cast<T>(p)

- Effettua controllo di compatibilità runtime assicurando cast sicuri tra tipi di classi
- Applicato a un puntatore, ritorna 0 se il cast non è valido
- Applicato a un riferimento genera un'eccezione in caso di incompatibilità
- Può effettuare il “downcast” da una classe virtuale di base ad una derivata



# dynamic\_cast<T>(p)

```
class Base1;
class Base2;
Class Derivata:
 public Base1,
 public Base2 { };

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

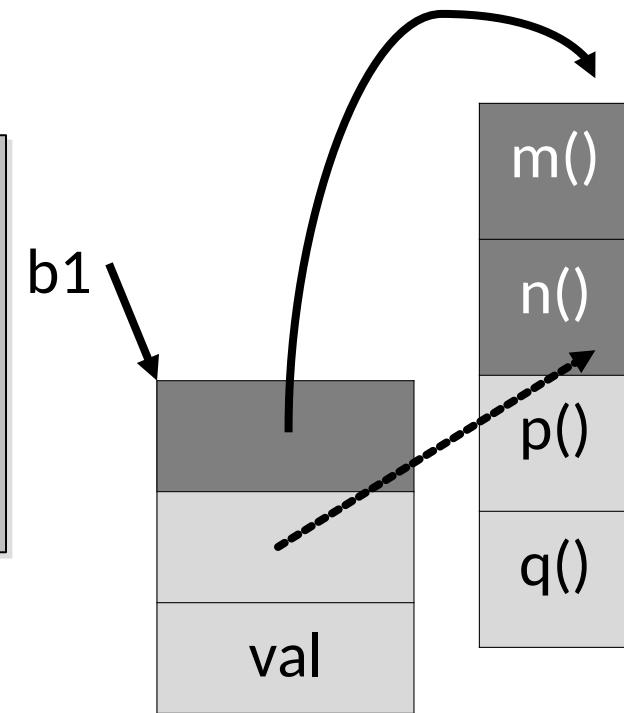
b1= dynamic_cast<Base1 *>(d);
b2= dynamic_cast<Base2 *>(d);
```



# dynamic\_cast<T>(p)

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

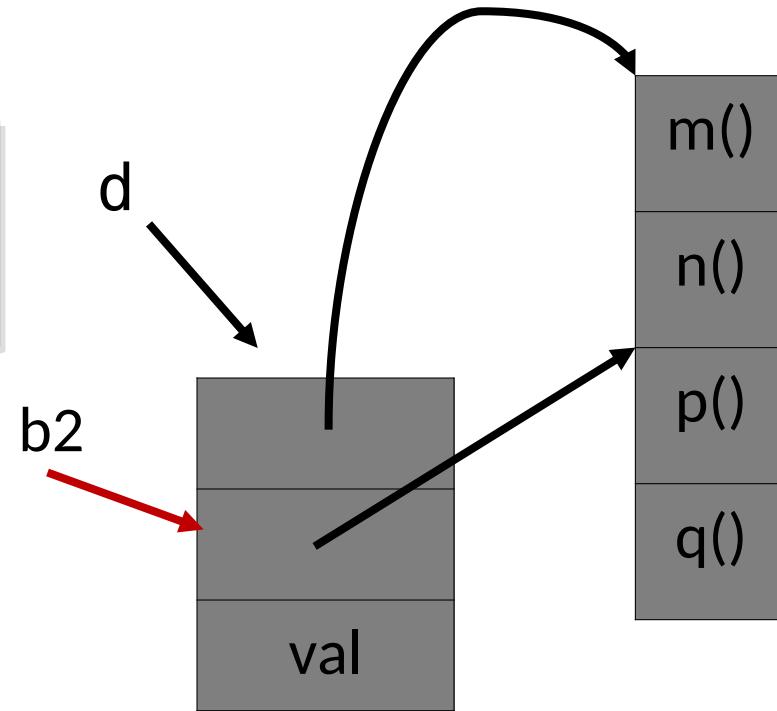
d = dynamic_cast< Derivata *>(b1);
```



Al contrario di static\_cast, d = NULL

# dynamic\_cast<T>(p)

```
Base2 *b2;
Derivata *d = new Derivata();
b2 = dynamic_cast<Base2*>(d);
```



In questo caso l'operazione non fallisce:  
b2 punta alle interfacce di d, derivate dalla classe di base Base2

# `reinterpret_cast<T>(p)`

- Interpreta la sequenza di bit di un valore di un tipo come valore di un altro tipo
- Autorizza il compilatore a violare il sistema dei tipi
- Utilizzato di solito per dati ritornati da chiamate al sistema operativo o ricevuti dall'hardware
- Analogo al C-style cast



# `const_cast<T>(p)`

- Elimina la caratteristica di costante dal suo argomento
- Può generare problemi se usato su variabili globali poste dal compilatore in memoria di sola lettura



# Spunti di riflessione

- Si scriva una gerarchia di classi che estendono una classe base, definendo due metodi virtuali e si verifichi il comportamento polimorfico





# Funzioni e operatori

Programmazione di Sistema  
A.A. 2017-18

# Argomenti

- Puntatori a funzione
- Oggetti funzionali
- Funzioni lambda
- Overloading degli operatori



# Puntatori a funzione

- In C++, come in C, è lecito salvare in una variabile il puntatore ad una funzione
  - Potrà essere utilizzato in seguito, permettendo di costruire programmi il cui comportamento cambia dinamicamente
- Per dichiarare una variabile di tipo puntatore a funzione si usa la sintassi
  - `<tipo_ritornato> (* var) (<argomenti>)`



# Puntatori a funzione

- Si assegna un valore attraverso l'operatore =
  - int f(int i, double d) {  
/\* corpo della funzione \*/  
};
  - int (\***var**)(int, double);
  - **var** = f;
  - **var** = &f; //identico al precedente
- Si invoca la funzione il cui puntatore è contenuto nella variabile con la sintassi
  - **var**(10, 3.1415926);
- Sia il tipo di ritorno che tutti i parametri formali della funzione devono corrispondere a quanto dichiarato nella definizione della variabile



# Puntatori a funzione

```
int f (int, int);
int f (int, double);
int g (int, int = 4);
double h (int);
int i (int);

int (*p) (int) = &g; // ERRORE: Manca il
parametro // di default
p = &h; // ERRORE: tipo di
ritorno //differente
p = &i; // OK
p = i; // OK

int (*p2) (int, double);
p2 = f; // OK: il compilatore
sceglie // "int f (int, double)"
```

# Oggetti funzionali

- In C++ esiste un ulteriore tipo invocabile:  
il «funtore» o «oggetto funzionale»
  - Istanza di una qualsiasi classe che abbia ridefinito la funzione membro **operator ()**

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

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



# Oggetti funzionali

- È possibile includere più definizioni di **operator ()**
  - Devono avere tipi differenti nell'elenco dei parametri
- Un oggetto funzionale può contenere variabili membro
  - Queste possono essere utilizzate all'interno delle funzioni operator() per tenere traccia di uno stato
- Il comportamento non è più quello di una funzione matematica (il cui output è sempre lo stesso a parità di input)
  - L'oggetto viene detto «funzionale»



# Oggetti funzionali

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

void main() {
 Accumulatore a;
 for (int i=0; i<10; i++)
 a(i);
 std::cout<<"Totale:"<<a.totale()<<std::endl;
 //stampa 45
}
```



# Mescolare i due approcci

- Utilizzando la programmazione generica, è possibile scrivere funzioni che accettano indifferentemente come parametri
  - puntatori a funzione
  - oggetti funzionali
- Questo permette di aumentare il livello di generalizzazione dei propri programmi
  - Approccio largamente utilizzato nella libreria standard
- L'utilizzo di oggetti funzionali è spesso molto utile
  - C++11 ha introdotto una notazione particolare per definirlo in modo compatto: le funzioni lambda



# Mescolare i due approcci

```
template <typename F>
void some_function(F& f) {
 //...
 f(); // f può essere un
funzionale o
 // un puntatore a funzione
 //...
}
```

# Funzioni minimali

- Spesso, l'utilizzo degli algoritmi presenti nella libreria standard richiede l'introduzione di funzioni usate una volta sola

```
void print(int i) {
 std::cout << i << " ";
}

int main() {
 std::vector<int> v;
 //...
 std::for_each(v.begin(), v.end(),
 print);
}
```



# Usare una funzione lambda

- Lo stesso comportamento può essere ottenuto usando una funzione lambda
  - Evitando di definire una funzione esplicita e doverle dare un nome che potrebbe collidere con altri nomi

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



# Restituire un valore

- Se il suo corpo contiene una sola istruzione return
  - Il tipo ritornato può essere dedotto automaticamente dal compilatore
  - Altrimenti occorre esplicitarlo nella definizione

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



# Catturare delle variabili

- Le parentesi quadre "[ ]" introducono la notazione lambda
  - Al loro interno è possibile elencare variabili locali il cui valore o il cui riferimento si vuole rendere disponibili nella funzione
- [x, y] - "x" e "y" sono catturate per valore
  - Viene effettuata una copia
  - La funzione  $\lambda$  potrà essere invocata anche quando tali variabili saranno uscite dallo scope
- [&x, &y] - "x" e "y" sono catturate per riferimento
  - Eventuali cambiamenti influenzano l'originale
  - Attenzione a riferimenti pendenti!
- [&] - cattura «tutto» per riferimento
- [x, &y] - cattura "x" per valore e "y" per riferimento



# Funzioni lambda: sintesi

- Sintassi
  - [ <captured\_vars> ] (<params>) -> <return\_type> { <function body> }
- Supportano diversi stili di programmazione
  - Algoritmi generici (funzione di confronto per sort)
  - Programmazione funzionale
  - Programmazione concorrente
- Migliorano la leggibilità di un programma
  - Permettendo di eliminare classi/funzioni piccole
  - ...a patto che si capisca cosa significano!



# Funzioni lambda:

- Vantaggi
  - Aumentano la leggibilità
    - Il corpo appare dove viene usato
  - Aumentano l'espressività
    - Rendono più chiare le intenzioni del programmatore
  - Aumentano la compattezza del codice
- Svvantaggi
  - Sintassi criptica
    - Per chi non conosce la notazione



# Operator overloading

- Tutti gli operatori in C++ (+,-,%,=, +=...) possono essere ridefiniti per l'uso con tipi di dati definiti dal programmatore
- Non è possibile definire nuovi operatori
- Non è possibile cambiare le precedenze
- Possono essere facilmente utilizzati in modo errato



# Operator overloading

```
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;
};
```



# Operator overloading

- La classe Frazione contiene due variabili intere private
  - numeratore (num) e denominatore (den)
- L'operatore “+” viene ridefinito in modo da sommare due istanze della classe Frazione
- L'operatore ritorna la copia di un oggetto di tipo Frazione



# Operator overloading

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

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

# Spunti di approfondimento

- Completare la classe Frazione aggiungendo le definizioni per gli operatori di sottrazione, moltiplicazione e divisione

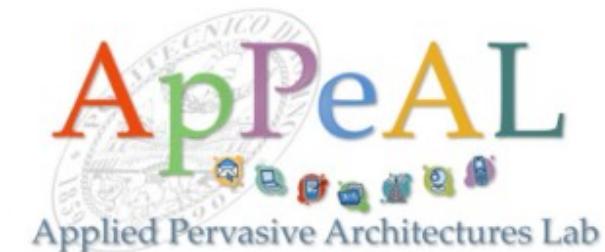




# Programmazione generica

Programmazione di Sistema  
A.A. 2017-18

Programmazione di Sistema



# Argomenti

- Programmazione generica
- Smart pointer



# Programmazione generica

- Un sistema di tipi stringente facilita la creazione di codice robusto
  - Identificando, in fase di compilazione, quei frammenti di codice che violano il sistema dei tipi
- In certe situazioni, per non violare il sistema dei tipi occorre replicare una grande quantità di codice generando problemi in fase di manutenzione
  - Occorre garantire che eventuali modifiche ad una versione del codice vengano propagate a tutte le altre
- Il linguaggio C++ offre un supporto alla generalizzazione
- dei comportamenti di un blocco di codice attraverso il concetto di “template”



# Template

- Frammenti (funzioni, classi) parametrici
  - Che vengono espansi in fase di compilazione, in base al loro uso effettivo
- Non richiedono controlli in fase di esecuzione
- Permettono di implementare una varietà di funzionalità diverse adattandole ai singoli tipi di dato
  - Ad esempio algoritmi per il valore minimo, massimo, medio, ...



# Usi dei template

- Algoritmi generici
- Collezioni di dati coerenti per il tipo encapsulato e relativi algoritmi
- Funzioni generalizzate
- La parte principale della libreria standard è basata su template



# Funzioni generiche

- Si può definire una funzione in modo che operi su un tipo di dato non ancora precisato
  - template <class T>  
const T& max(const T& t1, const T& t2) {  
 return (t1< t2 ? t2 : t1);  
}
- Opera sul tipo generico T, a patto che supporti
  - L'operatore “<” tra argomenti omogenei
  - Il costruttore di copia, per derivare una variabile temporanea a partire da un dato costante
- Ogni volta che si usa la funzione, il compilatore determina quale tipo effettivo assegnare a “T”, in base ai parametri passati



# Funzioni generiche

```
int i=max(10,20); // T → int
```

```
std::string s, s1 = ..., s2= ...;
s = max(s1, s2); // T →
std::string
```

```
max<double>(2, 3.1415928);
//forza la scelta di "T" a
double
```

# Classi generiche

- Lo stesso principio può essere adottato nella definizione delle classi
  - Quando queste vengono usate, si correda il tipo generico con l'indicazione della specializzazione richiesta

```
template <class T>
class Accum {
 T total;
public:
 Accum(T start): total(start) {}
 T operator+=(const T& t) {
 return total= total+t;
 }
 T value() { return total;}
};

Accum<std::string> sa("");
Accum<int> ia(0);
```



# Template C++

```
template <class T, int size>
class Stack {
public:
 Stack() { }
 ~Stack() { }

 void push(T val);
 T pop();

private:
 T data[size];
 int index;
};
```

```
Stack<int,100> s;
Stack<double,30> s1;
Stack<CProva,20> s2;
```

# Template C++

- I parametri del template possono essere identificatori di tipo di dato o valori costante
  - Evitano di ridefinire classi per tipi di dato o dimensioni differenti
- Ogni volta che si istanzia un template, indicando il valore dei parametri, il compilatore genera una nuova classe/funzione
  - Soluzione altrettanto versatile rispetto all'uso di gerarchie di ereditarietà e meno soggetta ad errori
  - Richiede pattern di programmazione appositi



# Specializzare un template

- Alcuni tipi potrebbero non essere utilizzabili all'interno di un dato template
  - Ad es., potrebbero non avere l'implementazione di un operatore usato nella definizione del template stesso
- Occorre modificare la classe
  - Fornendo le implementazioni e la semantica richieste
- Altrimenti, si può specializzare il template
  - Ovvero fornirne una definizione specifica per la classe non altrimenti usabile



# Specializzare un template

```
class Person {
 std::string name;
public:
 Person(std::string n): name(n) {}
 std::string name() { return name; }
};
template<> class Accum<Person> {
 int total;
public:
 Accum(int start = 0): total(start) {}
 int operator+=(const Person &) {return +
+total;}
 int value() { return total;}
};
```



# Template: punti di forza

- Aumentano notevolmente le possibilità espressive
  - Senza pregiudicare i tempi di esecuzione
  - Risparmiano tempo di sviluppo
  - Non mettono a rischio il sistema dei tipi
  - Introducono flessibilità in vista di futuri miglioramenti
  - Non solo a livello sintattico (lo fa già il compilatore), ma soprattutto a livello semantico



# Template: punti di attenzione

- Chi usa un template, deve verificarne le assunzioni sui tipi effettivamente usati
  - Eventuali casi particolari possono essere gestiti mediante la specializzazione
- Le violazioni sul tipo utilizzato conducono ad errori di compilazione difficili da interpretare
  - L'espansione del template conduce ad espressioni anche molto differenti dal codice originale
- Spesso, non occorre scrivere nuovi template
  - Occorre però sapere usare molto bene quelli esistenti



# Smart Pointer

- La ridefinizione degli operatori permette di dare una semantica alle operazioni '\*' e '->' anche a dati che non sono (solo) puntatori
- È possibile costruire oggetti che "sembrano" puntatori ma che hanno ulteriori caratteristiche
  - Garanzia di inizializzazione e rilascio
  - Conteggio dei riferimenti
  - Accesso controllato



# Smart Pointer

- Per aumentare il parallelo con i puntatori semplici, possono anche essere ridefinite le operazioni
  - Aritmetiche (`++`, `--`, ...)
  - Di confronto (`==`, `!=`, `!`, ...)
  - Di assegnazione/copia (`=`)



# Esempio

```
class int_ptr
{
 int* ptr;
 int_ptr(const int_ptr&);

 int_ptr& operator=(const int_ptr&);

public:
 explicit int_ptr(int* p) : ptr(p) { }

 ~int_ptr() {delete ptr;}

 int& operator*() {return *ptr;}

};
```



# Esempio

- Questa classe incapsula un puntatore ad un intero
  - Che si suppone essere allocato sullo heap
- Quando un oggetto di questo tipo viene distrutto la memoria viene rilasciata automaticamente
- Si potrebbero inserire controlli nel costruttore per verificare che il puntatore non sia nullo



# Generic smart\_ptr

```
template <class T>
class smart_ptr {
 T* ptr;
 smart_ptr(const smart_ptr<T>&);
 smart_ptr<T>& operator=(const
 smart_ptr<T>&);
public:
 explicit
 smart_ptr(T* p = 0) : ptr(p) {}
 ~smart_ptr() { delete ptr; }
 T& operator*() { return *ptr; }
 T* operator->() { return ptr; }
};
```



# Codici a confronto

```
void esempio1()
{
 MyClass* p =
 new MyClass();
 p->Eseguì();
 delete p;
}
```

```
void esempio2()
{
 smart_ptr<MyClass>
 p(new MyClass());
 p->Eseguì();
}
```

- smart\_ptr si occupa di chiamare l'operatore delete sul puntatore dell'oggetto encapsulato quando si esce dal blocco
  - Evitando perdite di memoria, anche in caso di eccezioni!



# Sicurezza per le eccezioni

- La funzione Esegui() potrebbe generare un'eccezione
  - Le righe di codice successive non verrebbero eseguite

```
void esempio1() {
 MyClass* p =new
 MyClass();
 p->Esegui();
 delete p;
}
```

# Sicurezza per le eccezioni

- L'uso di costrutti try/catch può fare esplodere le dimensioni del programma
  - L'uso dello smart pointer garantisce che la risorsa venga rilasciata automaticamente

```
void esempio3() {
 MyClass* p;
 try {
 p = new
 MyClass();
 p->Esegui();
 delete p;
}
catch (...) {
 delete p;
 throw;
}
}
```

# Copia e smart pointer

- A chi appartiene la memoria puntata da uno smart pointer?
  - Il problema si pone nel caso si voglia assegnare uno smart pointer ad un altro
- È possibile implementare strategie diverse
  - Passaggio di proprietà
  - Creazione di una copia
  - Condivisione con conteggio dei riferimenti
  - Condivisione in lettura e duplicazione in scrittura

# Ownership-Handling

- Se l'oggetto, referenziato dallo smart pointer, viene usato da un unico utilizzatore
  - può essere eliminato al termine delle operazioni dal distruttore dello smart pointer
- Se viene usato da più utilizzatori è necessario definire chi possa eliminare la risorsa
  - L'ultimo a conoscere l'oggetto
  - Un garbage collector che opera per conto del sistema



# Smart pointer nella libreria standard

- A partire dalla versione C++11, sono disponibili diversi template per la gestione automatica della memoria
  - `#include <memory>`
- `std::shared_ptr<BaseType>`
  - Implementa un meccanismo di conteggio dei riferimenti
- `std::weak_ptr<BaseType>`
  - Permette di osservare il contenuto di uno `shared_ptr` senza partecipare al conteggio dei riferimenti
- `std::unique_ptr<BaseType>`
  - Implementa il concetto di proprietà, impedendo la copia ma permettendo il trasferimento

# std::shared\_ptr

- Mantiene la proprietà condivisa ad un blocco di memoria referenziato da un puntatore nativo
  - Molti oggetti possono referenziare lo stesso blocco
  - Quando tutti sono stati distrutti o resettati, il blocco viene rilasciato
- Per default, il blocco referenziato viene rilasciato tramite l'operatore delete
  - In fase di costruzione di shared\_ptr, è possibile specificare un meccanismo di rilascio alternativo
- Un oggetto di questo tipo può anche non contenere alcun puntatore valido
  - Se è stato inizializzato o resettato al valore NULL



# **std::shared\_ptr**

- **shared\_ptr<T>( T\* nat\_ptr)**
  - Costruisce uno shared\_ptr che incapsula il puntatore nat\_ptr
- **~shared\_ptr<T>()**
  - Distrugge uno shared\_ptr
  - Libera la memoria, quando il contatore dei riferimenti è 0
- **operator=(const shared\_ptr<T>& other)**
  - Assegna il puntatore condiviso, incrementando il contatore
- **get(), operator\* (), operator-> ()**
  - Accesso al puntatore incapsulato



# **std::shared\_ptr**

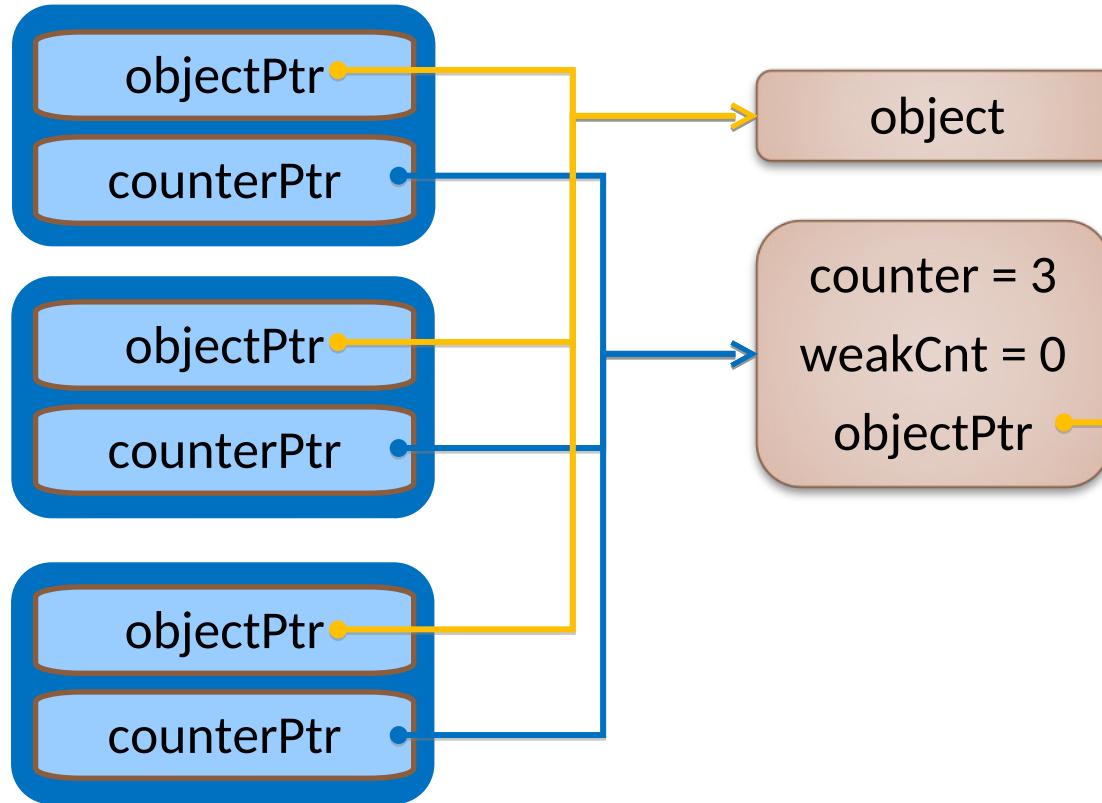
- **reset()**
  - Decrementa il contatore ed elimina il riferimento contenuto nello smart pointer
  - Se il contatore scende a 0, libera la memoria
- **reset(BaseType\* ptr)**
  - Come il precedente, ma sostituisce il puntatore contenuto con quello passato come parametro

# **std::shared\_ptr**

- `make_shared<BaseType>(params...)`
  - Funzione di supporto
  - Crea sullo heap un'istanza della classe `BaseType`
    - Usando i parametri passati
    - come parametri del
    - costruttore
  - Ne incapsula il puntatore in uno `shared_ptr`



# Implementazione tipica



# Implementazione tipica

- Ogni `shared_ptr` punta ad un blocco di controllo oltre che all'oggetto stesso
  - La dimensione dello smart pointer raddoppia
- Occorre memorizzare il blocco di controllo



# `std::auto_ptr<BaseType>`

- Il puntatore encapsulato negli oggetti di tipo `auto_ptr` appartiene a tali oggetti
  - La memoria cui fa riferimento deve appartenere allo heap ed essere stato allocato tramite `new`
- All'atto della distruzione di `auto_ptr`, viene invocato il metodo `delete`
  - Cosa capita se si duplica un oggetto di tipo `auto_ptr`, attraverso un'assegnazione o un costruttore di copia?
- La proprietà del puntatore viene trasferita al destinatario
  - L'oggetto sorgente viene modificato, ponendo a `NULL` il puntatore encapsulato
  - Altrimenti si rischierebbe di invocare `delete` due volte!



# Dipendenze cicliche

- Il conteggio dei riferimenti garantisce il rilascio della memoria in modo deterministico
  - Non appena un oggetto non ha più riferimenti viene liberato
- In alcuni casi, tuttavia, non funziona
  - Se si forma un ciclo di dipendenze ( $A \rightarrow B$ ,  $B \rightarrow A$ ) il contatore non può mai annullarsi, anche se gli oggetti A e B non sono più conosciuti da nessuno
- Occorre evitare la creazione di cicli ricorrendo a
  - `std::weak_ptr<BaseType>`



# `std::weak_ptr`

- Mantiene al proprio interno un riferimento “debole” ad un blocco custodito in uno `shared_ptr`
  - Modella il possesso temporaneo ad un blocco di memoria
- Si acquisisce temporaneamente l’accesso al dato tramite il metodo `lock()` che restituisce uno `shared_ptr`
  - Si verifica la validità del riferimento con il metodo `expired()`

# **std::weak\_ptr**

```
std::weak_ptr<int> gw;
void f(){
 if (auto spt = gw.lock())
 std::cout<<"gw:"<< *spt << "\n";
 else
 std::cout<< "gw e' scaduto\n";
}
int main() {
 { auto sp = std::make_shared<int>(42);
 gw = sp;
 f();
 } // sp viene distrutto, gw scade
 f();
}
```



# `std::unique_ptr`

- Incapsula il puntatore ad un oggetto o ad un array
  - Non può essere copiato né assegnato
  - Il puntatore può essere trasferito esplicitamente ad un altro `unique_ptr` con la funzione `std::move()`
- Quando viene distrutto, il blocco viene rilasciato

# **std::unique\_ptr - Usi tipici**

- Garantire la distruzione di un oggetto
  - Anche in caso di eccezioni
  - Passare la proprietà di un oggetto con ciclo di vita dinamico tra funzioni
- Gestire in modo sicuro oggetti polimorfici



# Spunti di riflessione

- Si implementi una lista circolare generica encapsulando i puntatori agli elementi in oggetti di tipo `shared_ptr<T>`



# Librerie C++

Anno Accademico 2017-18

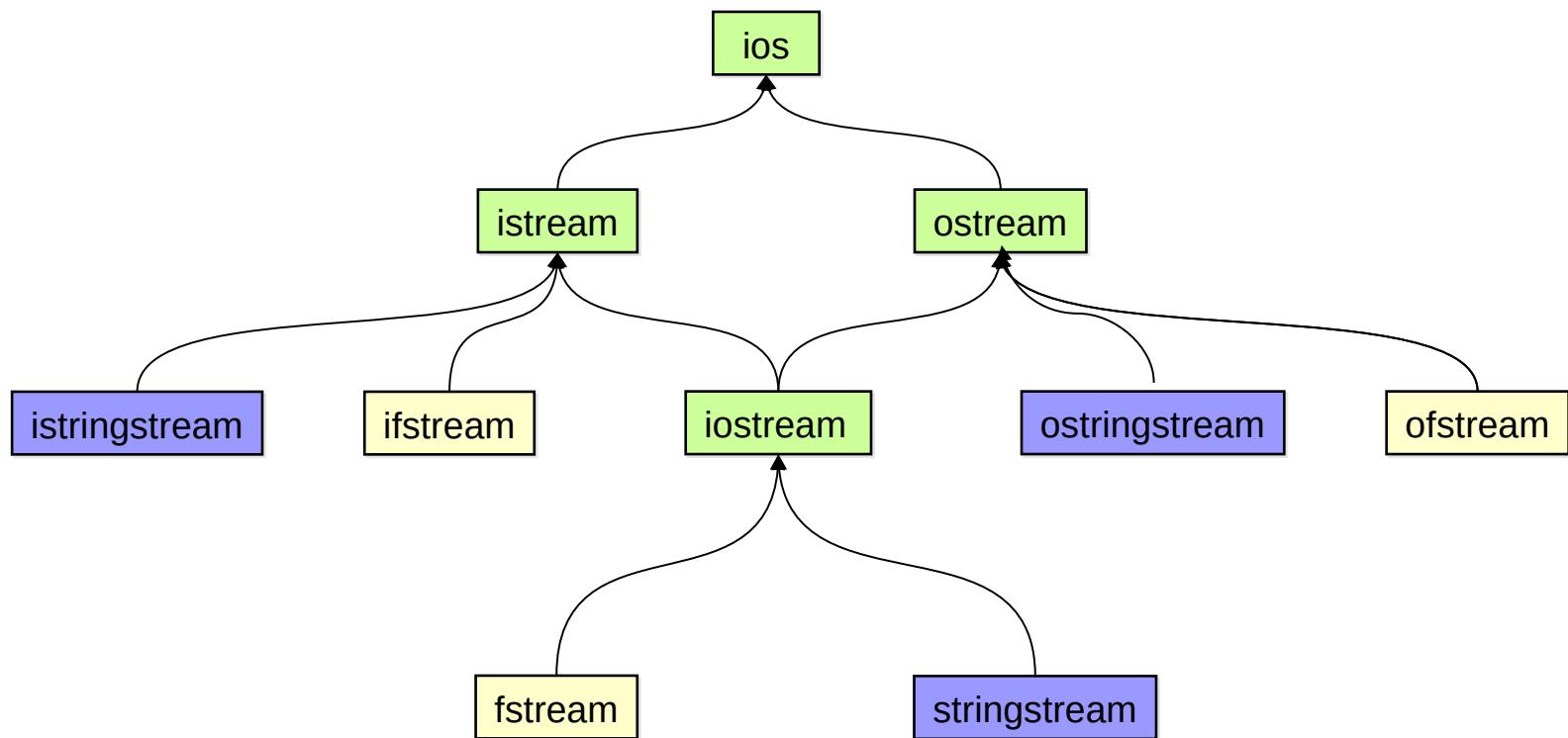


# Argomenti

- Input/output in C++
- Standard Template Library

# Input/Output in C++

- È possibile utilizzare le funzioni della libreria stdio.h
  - printf, scanf, ecc...
- Il C++ fornisce un'alternativa per la gestione dei flussi di input e di output



# Gerarchia di classi di I/O (1)

- **ios** è la classe base (virtuale) e contiene attributi e metodi comuni a tutti gli stream
  - stato dello stream: (errori recuperabili, errori irrecuperabili, fine del flusso, ...)
  - manipolazione del formato
  - non può essere istanziata direttamente
- **istream** e **ostream** eseguono rispettivamente le operazioni di input ed output su flussi generici
  - metodi di lettura e scrittura dello stream
  - operatori `>>` e `<<`
  - accesso indiretto allo stream tramite buffer (allocato internamente)
- **iostream** esegue operazioni di lettura e scrittura



# Gerarchia di classi di I/O (2)

- Operazioni su file
  - le classi base sono **ifstream** e **ofstream**
    - derivano (indirettamente) da ios
    - contengono metodi per la creazione dello stream (open, close)
  - le classi derivate definiscono stream per l'input e l'output
    - ifstream → accesso in sola lettura
    - ofstream → accesso in sola scrittura
    - fstream → accesso allo stream sia in lettura che in scrittura
- Esistono classi analoghe per le operazioni di lettura e scrittura su stringhe
  - istringstream, ostringstream, stringstream



# Output su video (1)

- Per le operazioni di output
  - si utilizza la variabile `std::cout`
  - l'operatore `<<`
  - le funzioni fornite dalla classe `ostream`

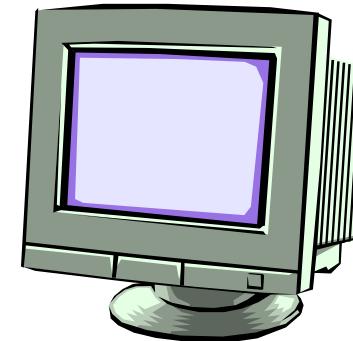


```
using namespace std;

char ch='a'; cout<<ch; // scrive un carattere
int i=10; cout<<i; // scrive un intero
float f=1.1; cout<<f; // scrive un float
char* s="ab"; cout<<s; // scrive una stringa
if (!cout) { ... } // Errore in scrittura
```

# Output su video (2)

- Operazioni fornite dalla classe **ostream**
  - scrittura di un carattere ch qualsiasi
    - `char ch='a'; cout.put(ch);`
  - scrittura di una blocco di byte lungo al più n
    - `cout.write(s,n);`



# I/O con formato

- Funzione **width**: ampiezza del campo
  - opera sull'istruzione di scrittura successiva e definisce il numero minimo di caratteri da impiegare
  - `cout.width(4); cout<<'a'; // stampa □ □ □ a`
- Funzione **precision**: precisione
  - è applicabile ai numeri reali e definisce il numero massimo di caratteri da impiegare
  - `cout.precision(4); cout<<1.4142 // stampa 1.414`

# Manipolatori

- Particolari oggetti, detti manipolatori, possono essere inviati in scrittura o in lettura ad un flusso
  - permettono di cambiare il formato di rappresentazione, inserire o estrarre particolari dati nel flusso
  - Definiti nel file <iomanip>

```
using namespace std;
int i = 10;
cout<<i<<" (0x"<<hex<<setfill('0')<<setw(4)<<i<<")"<<endl;
```

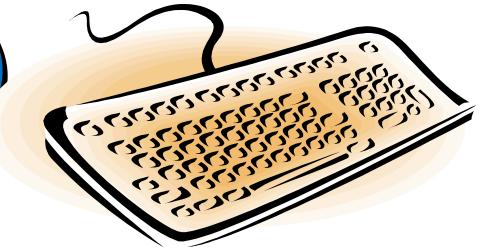
10 (0x000a) ←



# Manipolatori di formato

- **skipws**: ignora gli spazi
- **left**: allinea a sinistra
- **right**: allinea a destra
- **internal**: segno a sinistra del campo, valore a destra
- **dec**: intero base decimale
- **oct**: intero base ottale
- **hex**: intero base esadecimale
- **showpoint**: reale, usa punto decimale
- **flush**: svuota il buffer in uscita
- **endl**: invia a cout il carattere '\n'
- **ends**: invia a cout il carattere '\0'
- **ws**: produce l'eliminazione degli spazi in input
- ...

# Input da tastiera (1)



- Le operazioni primarie di I/O avvengono attraverso variabili standard
  - `std::cin` - flusso standard di ingresso
  - `std::cout` - flusso standard di uscita
  - `std::cerr` - flusso standard di errore (privo di buffer)
  - `std::clog` - flusso standard di errore (con buffer)
- Per le operazioni di input
  - si utilizza la variabile `std::cin`
  - l'operatore `>>`
  - le funzioni fornite dalla classe `istream`
- Utilizzando il costrutto  
`using namespace std;`  
si può omettere il prefisso `std::`

# Input da tastiera (2)

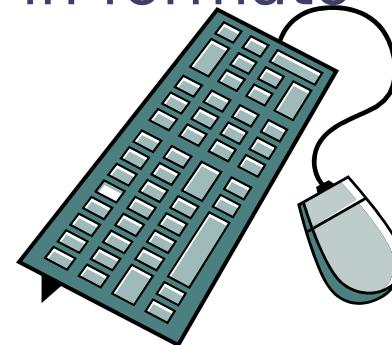
- Nella lettura con l'operatore >>
  - `cin>>ch;` ignora gli eventuali spazi iniziali
  - `cin>>s;` ignora gli spazi iniziali e si arresta al primo carattere di spaziatura, la stringa viene completata con \0
  - nel caso di errore o di EOF il valore restituito da (`!cin`) è 0

```
char ch; cin>>ch; // legge un carattere
int i; cin>>i; //legge un intero
float f; cin>>f; //legge un float
char s[100]; cin>>s; //legge una stringa (fino
 //ad uno spazio, tab, return,...)
if(!cin) cout << "Errore in lettura"<<endl;
```



# Input da tastiera (3)

- Funzioni fornite dalla classe **istream**
  - lettura di un carattere ch qualsiasi
    - `cin.get(ch);`
  - lettura di una stringa di caratteri lunga al più n-1
    - `cin.getline(s,n);`
  - lettura di un blocco di dati in formato binario
    - `cin.read(s,1000);`



# Gestione dello stato dello stream

- A differenza di quanto avviene in Java, in C++ le operazioni di IO non sollevano eccezioni
  - Il programmatore deve testare **esplicitamente** il risultato di ogni operazione effettuata
- Gli indicatori seguenti registrano la condizione che si è verificata a seguito dell'ultima operazione di I/O
  - **eof()** → true se è stato incontrato EOF
  - **fail()** → true se è avvenuto un errore di formato, ma che non ha causato la perdita di dati
  - **bad()** → true se c'è stato un errore che ha causato la perdita di dati
  - **good()** → true se non c'è stato alcun errore
- Il metodo **clear** ripristina lo stato del flusso

# Esempio (1)



```
#include <iostream>
#include <iomanip>
using namespace std;
int main ()
{
 int a, b, c = 8, d = 4;
 float k ;
 char name[30] ;
 cout << "Enter your name" << endl ;
 cin.getline (name, 30) ;
 cout << "Enter two integers and a float " << endl ;
 cin >> a >> b >> k ;
 if (! cin.good()) return -1; //error
```

# Esempio (2)

```
cout << "\nThank you, " << name << ", you entered"
 << endl << a << ", " << b << ", and ";
cout.width (4);
cout.precision (2);
cout << k << endl;

// Modo alternativo per controllare l'output
cout << "\nThank you, " << name << ", you entered"
 << endl << a << ", " << b << ", and " << setw (4)
 << setprecision (2) << k << endl;
return 0;
}
```



# Esempio - Output

Enter your name

*R. J. Freuler*

Enter two integers and a float

12 24 67.857

Thank you, R. J. Freuler, you entered  
12, 24, and 67.857

Thank you, R. J. Freuler, you entered  
12, 24, and 67.857

# Esempio - I/O da file



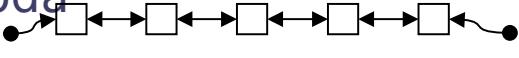
```
#include <fstream>
using namespace std;
int main () {
 int a, b, c ;
 ifstream fin ; //creazione dell'input stream
 fin.open ("my_input.dat") ; //apertura del file
 fin >> a >> b ; //lettura di due interi
 if (!fin) return -1; // equivalente a if (!fin.good())...
 c = a + b ;
 ofstream fout ; //creazione di un file di output
 fout.open ("my_output.dat"); // apertura del file
 fout << c << endl ; //scrittura del risultato
 fin.close () ; //chiusura input file
 fout.close () ; //chiusura output file
 return 0;
}
```

# Standard Template Library

- Il comitato di standardizzazione del C++ ha definito un insieme di funzionalità, espresse attraverso template, comuni alle diverse implementazioni
  - Standard Template Library
- La libreria STL è caratterizzata da
  - classi **contenitore**, i cui oggetti sono collezioni omogenee di altri oggetti
    - vector, deque, list, set, multiset, map, multimap, **string**
  - **algoritmi** generici
    - ricerca, fusione, ordinamento,...
    - è necessario che il tipo di dato su cui viene applicato abbia definite specifiche operazioni (ad esempio ==, <, >)
- Per accedere agli elementi di una classe contenitore si utilizzano gli **iteratori**
  - oggetti di tipo cursore che si comportano come puntatori

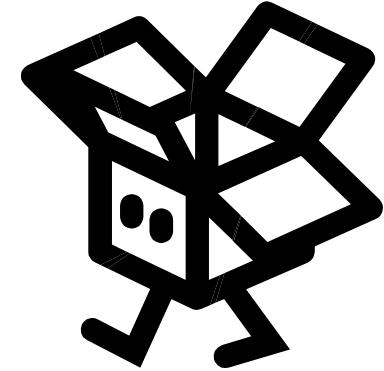
# Contenitori (1)



- I contenitori di tipo **sequenziale** organizzano linearmente una collezione di oggetti
  - array “tradizionale” A horizontal bar divided into 10 equal segments by vertical lines, representing a 1-dimensional array of 10 elements.
  - fornisce un accesso casuale a tempo costante ad una sequenza di lunghezza fissa
  - vector A horizontal bar divided into 10 equal segments, with the last segment being longer to represent a dynamically growing array.
  - fornisce un accesso casuale a tempo costante, con tempo di inserimento e cancellazione costante in coda
  - deque A horizontal bar divided into 10 equal segments, with arrows at both ends pointing towards the center, indicating insertion and deletion from both ends.
  - fornisce un accesso casuale a tempo costante, con tempo di inserimento e cancellazione costante in testa e in coda
  - list A sequence of six square nodes connected by arrows. The first node has a solid arrow pointing to the second, and each subsequent node has a solid arrow pointing to the next. The last node has a curved arrow pointing back to the first node, forming a circular link.
  - fornisce un accesso casuale con tempo lineare e tempo di inserimento e cancellazione costante in qualsiasi posizione

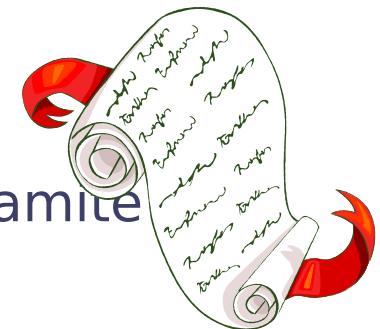
# Contenitori (2)

- I contenitori di tipo **associativo** forniscono l'accesso agli oggetti della collezione tramite chiavi
  - set
    - la chiave deve essere unica
  - multiset
    - la chiave può essere duplicata
  - map
    - basata su coppie chiave, valore
    - la chiave deve essere unica
  - multimap
    - la chiave può essere duplicata



# Liste (1)

- Sono costituite da sequenze di dati omogenei
  - Incapsulano tutte le operazioni di gestione
  - forniscono una sintassi ed una semantica simile a quella dei tipi elementari
- Un oggetto di tipo list encapsula una lista doppiamente collegata di oggetti
  - dal punto di vista dell'utente è una sequenza di oggetti che su cui è possibile
    - inserire nuovi oggetti
    - cancellare gli oggetti
    - esaminarne il contenuto
  - si accede agli elementi di una lista tramite un iteratore



# Liste (2)

- Le liste contengono dati omogenei e generici
  - ogni lista può contenere un solo tipo di oggetti
    - viene specificato alla creazione della lista
    - deve supportare copia e assegnazione
  - il compilatore impedisce che vengano inseriti dati di un tipo inappropriate
- Per utilizzare le liste è necessario includere il file `<list>`
  - definisce i template che sono usati per dichiarare e creare le liste (che appartengono allo spazio dei nomi “std”)

```
#include <list>
using namespace std; //evita di dover scrivere
std::list<...>
```
- Si dichiara una lista istanziandone il template
  - `list<int> w;`
  - `list<string> y;`

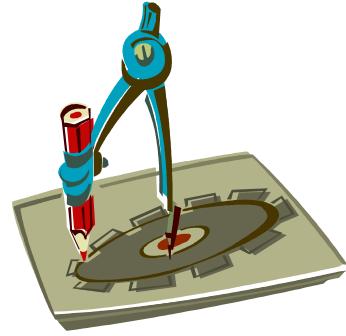


# Creazione di liste (3)



- Lista vuota
  - `list <int> c0;`
- Lista con n elementi con il valore di default
  - `list <int> c1( 3 ); // 0 0 0`
- Lista con n elementi del valore specificato
  - `list <int> c2( 5, 2 ); // 2 2 2 2 2`
- Utilizzando l'allocatore di un'altra lista
  - `list <int> c3( 3, 1, c2.get_allocator( ) ); // 1 1 1`
- Copiando un'altra lista
  - `list <int> c4(c2); // 2 2 2 2 2`

# Iteratori



- Oggetti che indicano una posizione all'interno di un contenitore
  - Si accede all'oggetto dereferenziando l'iteratore (come fosse un puntatore)
- Se si incrementa un iteratore (`operator++()`), si avanza all'elemento successivo
  - Viceversa, si torna all'elemento precedente se lo si decrementa (`operator--()`)
- Due iteratori possono essere confrontati per egualianza (`operator==(())`) o diversità (`operator!=(())`)

# Puntatori come iteratori

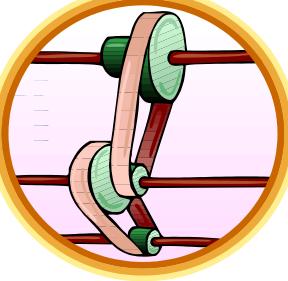
- L'iteratore più semplice è un puntatore ad un blocco di oggetti
- Si delimita il blocco con due puntatori
  - L'inizio (incluso)
  - La fine (esclusa)
- In un contenitore vuoto, inizio e fine coincidono
- Dereferenziare il puntatore finale è un errore
  - Si punterebbe oltre il limite dell'array

```
class C;
C array[10];

C *iter = array;
C *end = array+10;

for(; iter != end; iter+
+) {
 C elem =
 *iter;
 //...
}
```

# Generalizzare gli iteratori



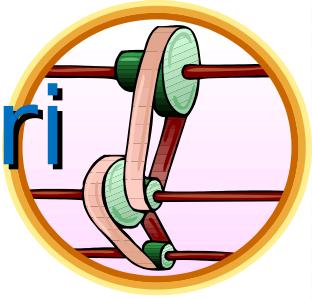
- Il codice precedente può essere generalizzato
- Invece che operare su puntatori, può operare su oggetti di altro tipo purché possano essere
  - Confrontati (operator==, operator!=)
  - Incrementati (operator++)
  - Dereferenziati (operator\*, operator->)

```
class C;
list<C> l(10);

list<C>::iterator iter =
 l.begin();
list<C>::iterator end =
 l.end();

for(; iter != end; iter++) {
 C elem = *iter;
 //...
}
```

# Generalizzare gli iteratori



- Il codice dell'algoritmo non è cambiato
  - È cambiato il tipo di dato usato per scorrere il contenitore
- Il compilatore si occupa di selezionare le corrette operazioni (definite nella classe dell'iteratore) per eseguire confronti, incrementi e accesso al contenuto
  - Ogni contenitore offre un proprio tipo di iteratore in grado di implementare la necessaria semantica

# Liste - Operazioni (1)

- **push\_back** aggiunge un elemento alla fine della lista
- **push\_front** aggiunge un elemento all'inizio della lista
- **back** restituisce il riferimento all'ultimo elemento della lista
- **front** restituisce il riferimento al primo elemento della lista

```
list <int> c1;
c1.push_back(10); c1.push_back(11);
int& i = c1.back();
int& j = c1.front();
```



# Liste - Operazioni (2)

- **erase** rimuove un elemento o una serie di elementi a partire dalla posizione specificata  
`c1.push_back( 10 );  
c1.push_back( 20 );  
c1.erase( c1.begin( ) );`
- **insert** inserisce uno o più elementi nella lista nella posizione specificata

```
list <int>::iterator Iter;
c1.push_back(10);
c1.push_back(20);
Iter = c1.begin();
Iter++;
c1.insert(Iter, 100);
```

# Liste - Operazioni (3)

- **clear** elimina tutti gli elementi della lista  
`c1.clear( );`
- **empty** verifica se la lista è vuota  
`bool b = c1.empty( );`
- **size** restituisce il numero di elementi nella lista  
`list <int>::size_type l = c1.size( );`
- **max\_size** restituisce il numero massimo di elementi che la lista può contenere  
`list <int>::size_type m = c1.max_size( );`
- **pop\_back** cancella l'ultimo elemento della lista  
`c1.pop_back( );`
- **pop\_front** cancella il primo elemento della lista  
`c1.pop_front( )`

# Liste - Operazioni (3)

- `remove_if` cancella tutti gli elementi per cui è verificata la condizione specificata

```
class is_odd :
 public std::unary_function<int, bool> {
 public: bool operator() (int val) {
 return (val % 2) == 1; }
 };

int main() {
 list <int> c1;
 c1.push_back(3); c1.push_back(4); c1.push_back(5);
 c1.push_back(6); c1.push_back(7); c1.push_back(8);
 c1.remove_if(is_odd()); // c1 = 4 6 8
}
```

# Liste - Operazioni (4)

- **merge** elimina gli elementi della lista passata come argomento e li inserisce nella lista destinazione, riordinandola in ordine crescente (o nell'ordine specificato)

```
list <int> c1, c2, c3;
list <int>::iterator c1_iter, c2_iter, c3_iter;
c1.push_back(3); c1.push_back(6);
c2.push_back(2); c2.push_back(4);
c3.push_back(5); c3.push_back(1);
c2.merge(c1); // 2 3 4 6
c2.merge(c3, greater<int>()); // 6 5 4 3 2 1
```

# Liste - Operazioni (5)

- **remove** cancella l'elemento passato come parametro

```
c1.push_back(5);
c1.remove(5);
```

- **resize** specifica la nuova dimensione della lista

```
c1.push_back(10); c1.push_back(20);
c1.push_back(30);
c1.resize(6,40); // c1 =
10,20,30,40,40,40
```



# Liste - Operazioni (5)

- **reverse** inverte l'ordine degli elementi  
`c1.reverse( );`
- **sort** ordina gli elementi in ordine ascendente o secondo l'ordinamento specificato  
`c1.push_back( 20 );`  
`c1.push_back( 10 );`  
`c1.push_back( 30 );`  
`c1.sort(); // c1 = 10 20 30`  
`c1.sort( greater<int>() ); // c1 = 30 20 10`



# Liste - Operazioni (6)

- **splice** cancella gli elementi dalla lista passata come argomento e li inserisce nella lista destinazione

```
list <int> c1, c2;
c1.push_back(10); c1.push_back(11);
c2.push_back(12); c2.push_back(20);
c2.push_back(21);

c2.splice(c2.begin(), c1);
// c2 = 10 11 12 20 21
```



# Liste - Operazioni (7)

- **unique** rimuove gli elementi adiacenti duplicati

```
c1.push_back(-10);
c1.push_back(10);
c1.push_back(10);
c1.push_back(20);
c1.push_back(-10);
c1.push_back(20);
c1.sort();
//c1 = -10-10 10 10 20 20
c1.unique();
// c1 = -10 10 20
```



# Liste - Operazioni (7)

- **swap** scambia gli elementi di due liste

```
c1.push_back(1);
c1.push_back(2);
c1.push_back(3);
c2.push_back(10);
c2.push_back(20);
c3.push_back(100);
c1.swap(c2); // c1 = 10 20
swap(c1,c3); // c1 = 100
```

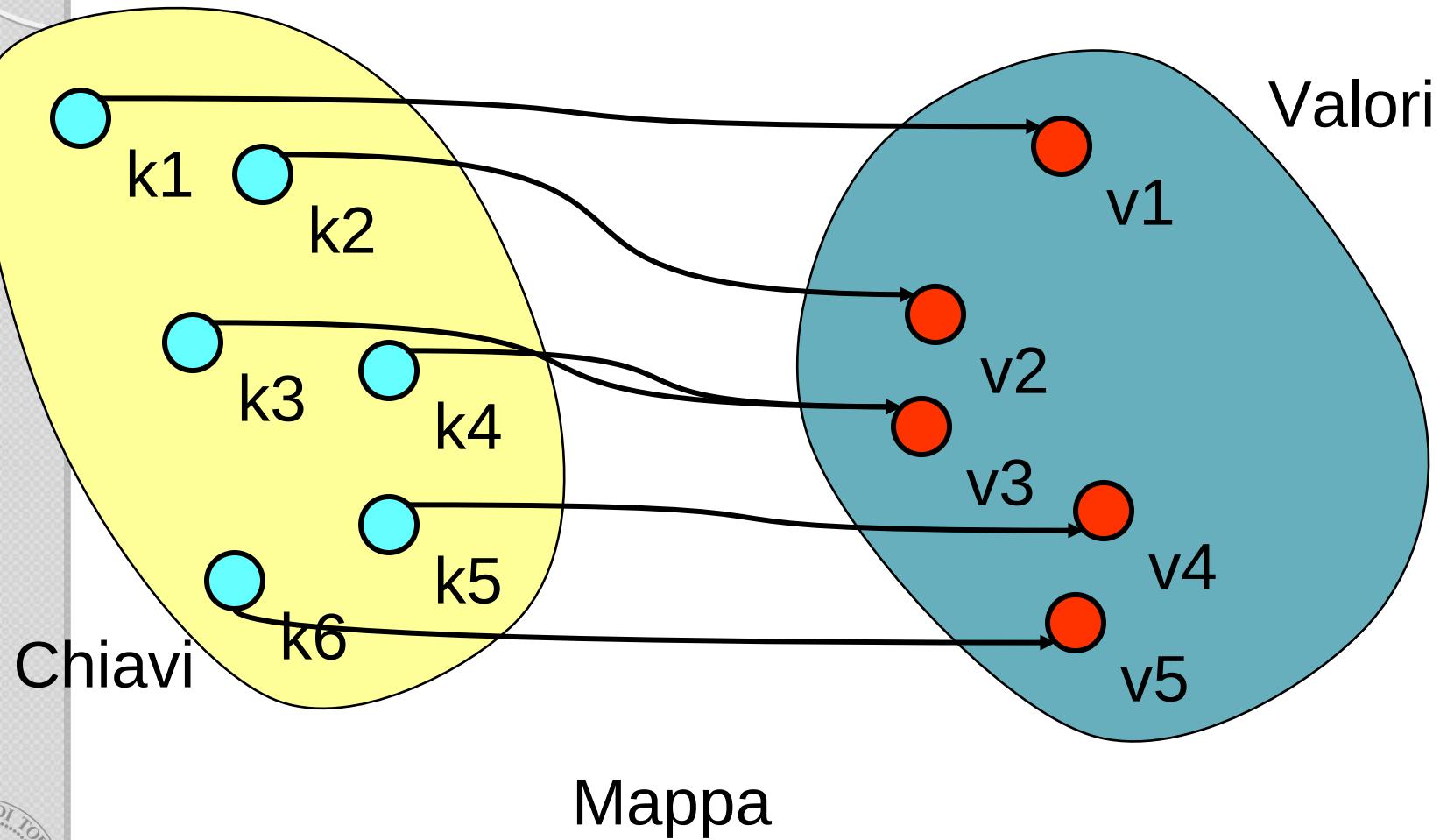


# Mappe (1)

- Modellano il concetto di relazione tra due insiemi di oggetti, detti rispettivamente chiavi e valori
  - Una mappa associa, ad ogni chiave, uno ed un solo valore
  - Le chiavi devono essere oggetti immutabili
- Per utilizzare le mappe è necessario includere il file <map>
  - `#include <map>`



# Mappe (2)

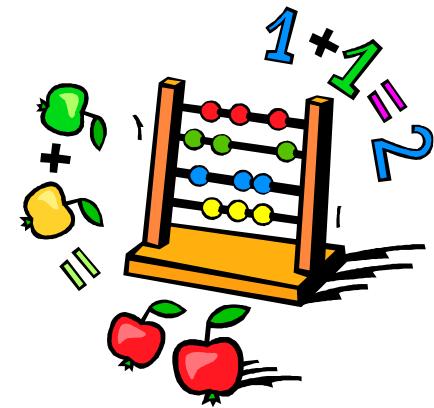


# Mappe (3)



- Una mappa è un contenitore:
  - di tipo *sorted associative*
    - ▀ mantiene gli oggetti ordinati in base alle loro chiavi
  - di tipo contenitore di tipo *pair associative*
    - ▀ pair <const Key, Data>
  - di tipo *unique associative*
    - ▀ due elementi non possono avere la stessa chiave
- Per accedere agli elementi di una mappa, si utilizzano gli iteratori
  - l'inserimento di un elemento non invalida gli iteratori esistenti che puntano a elementi esistenti
  - la cancellazione di un elemento non invalida gli iteratori, tranne quelli che puntano all'elemento cancellato

# pair <t1,t2>



- Coppia di elementi eterogenei
- t1 e t2 devono essere modelli di tipo Assignable
  - un tipo di dato è Assignable se
    - è possibile costruire una copia delle sue istanze
    - è possibile assegnare un nuovo valore ad una variabile che contiene un'istanza precedente
- Per accedere al primo elemento della coppia
  - pair.first;
- Per accedere al secondo elemento della coppia
  - pair.second;

# Operatori (1)

- **insert** – inserisce uno o più elementi in una mappa

```
map <int, int> m1;
typedef pair <int, int> Int_Pair;
m1.insert (Int_Pair (0, 0));
m1.insert (Int_Pair (1, 1));
m1.insert (Int_Pair (2, 4));
```

# Operatori (2)

- **begin** – restituisce l'iteratore alla posizione del primo elemento della mappa

```
map <int, int> m1;
map <int, int> :: iterator m1_Iter;
typedef pair <int, int> Int_Pair;
m1.insert (Int_Pair (0, 0));
m1.insert (Int_Pair (1, 1));
m1.insert (Int_Pair (2, 4));
m1_Iter = m1.begin ();
```



# Operatori (3)

- **end** – restituisce un iteratore alla posizione successiva all'ultimo elemento della mappa

```
map <int, int> m1;
map <int, int> :: iterator m1_Iter;
typedef pair <int, int> Int_Pair;
m1.insert (Int_Pair (1, 10));
m1.insert (Int_Pair (2, 20));
m1.insert (Int_Pair (3, 30));
m1_cIter = m1.end();
```



# Operatori (4)

- **empty** - restituisce true se la mappa è vuota
- **clear** - cancella tutti gli elementi della mappa
- **count** - restituisce il numero di elementi la cui chiave è quella passata come parametro

```
map<int, int> m;
map<int, int>::size_type i;
typedef pair<int, int> Int_Pair;
m1.insert(Int_Pair(3, 4));
i = m.count(3); // i = 1
```

# Operatori (5)

- **equal\_range** - restituisce una coppia di iteratori che identificano l'intervallo di elementi la cui chiave è pari al valore specificato

```
typedef map <int, int, less<int> > IntMap;
typedef pair <int, int> Int_Pair;
m1.insert (Int_Pair (1, 10));
m1.insert (Int_Pair (2, 20));
m1.insert (Int_Pair (3, 30));
pair <IntMap::const_iterator, IntMap::const_iterator> p1,
 p2;
p1 = m1.equal_range(2);
p1.first -> second; // vale 20
p1.second -> second; // vale 30
```

# Operatori (6)

- **erase** - rimuove uno o piu' elementi dalla posizione specificata  
`Iter1 = ++m1.begin();  
m1.erase(Iter1);`
- **max\_size** - restituisce la dimensione massima della mappa
- **size** - restituisce il numero di elementi della mappa



# Operatori (7)

- **find** – restituisce un iteratore che punta all'elemento la cui chiave è quella passata come parametro

```
m1.insert (Int_Pair (1, 10));
m1.insert (Int_Pair (2, 20));
m1.insert (Int_Pair (3, 30));
map <int, int> :: const_iterator m1Iter =
 m1.find(2);
```



# Operatori (8)

- **operator[]** – inserisce o modifica il valore di un elemento della mappa  
`m1[ 1 ] = 10;`
- **get\_allocator** – restituisce una copia dell'allocatore utilizzato per costruire la mappa  
`map <int, int>::allocator_type m1_Alloc;  
map <int, int, allocator<int> > m1;  
m1_Alloc = m1.get_allocator( );`



# Operatori (9)

- **lower\_bound** – restituisce un iteratore al primo elemento il cui valore della chiave è maggiore o uguale di quello specificato
- **upper\_bound** – restituisce un iteratore al primo elemento il cui valore della chiave è maggiore di quello specificato
- **swap** – scambia gli elementi di due mappe

# Esempio (1)

```
struct Itstr {
 bool operator()(const char* s1, const char* s2) const
 {
 return strcmp(s1, s2) < 0;
 } };
int main() {
 map<const char*, int, Itstr> months;
 months["january"] = 31;
 months["february"] = 28;
 months["march"] = 31;
 months["april"] = 30;
 months["may"] = 31;
 months["june"] = 30;
 months["july"] = 31;
 months["august"] = 31;
//... continua
```



# Esempio (2)

```
months["september"] = 30;
months["october"] = 31;
months["november"] = 30;
months["december"] = 31;

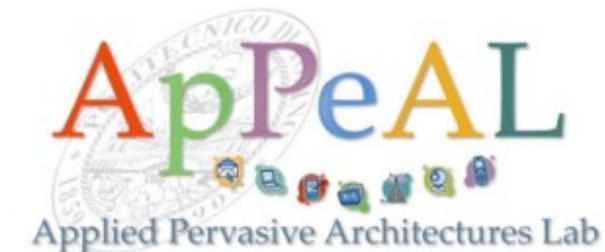
cout << "june -> " << months["june"] << endl;

map<const char*, int, Itstr>::iterator cur = months.find("june");
map<const char*, int, Itstr>::iterator prev = cur;
map<const char*, int, Itstr>::iterator next = cur;
++next;
--prev;
cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
```



# Librerie

Programmazione di Sistema  
A.A. 2017-18



# Argomenti

- Uso di librerie
- Librerie statiche
- Librerie dinamiche o condivise

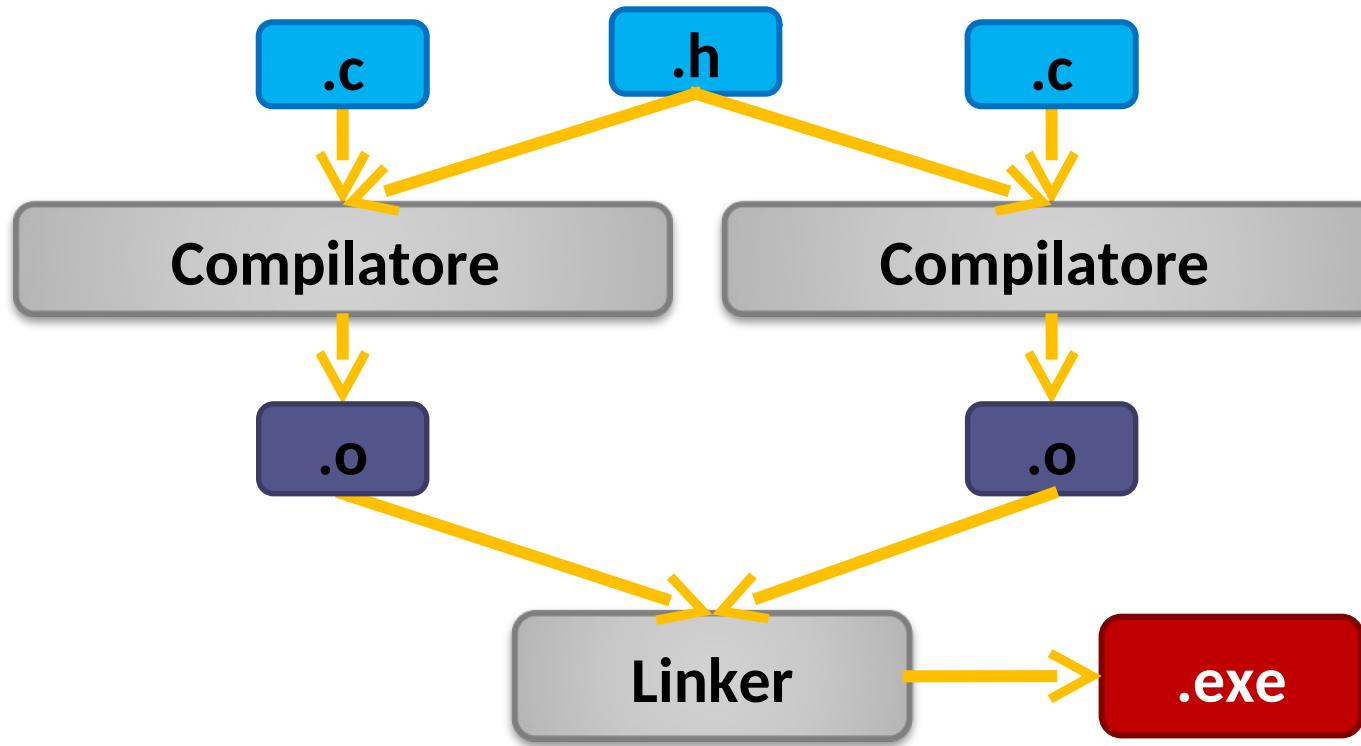


# Il processo di compilazione

- Il programma è suddiviso in un insieme di file sorgenti
  - Ciascuno è compilato separatamente
  - I file header permettono di dichiarare simboli definiti esternamente
- Il compilatore produce moduli oggetto
  - Contengono codice macchina con riferimenti pendenti alle funzioni/variabili esterne
- Il linker unisce più moduli oggetto in un eseguibile finale
  - Risolvendo i riferimenti ai simboli esterni ed inserendo gli opportuni indirizzi ad essi



# Il processo di collegamento



# Librerie

- Insieme di moduli oggetto archiviati in un unico file
  - Facilitano la gestione modulare dei programmi
  - Riducono i tempi di compilazione
  - Favoriscono l'incapsulamento
- Permettono di radunare funzioni e strutture dati pertinenti ad uno stesso dominio
  - Funzioni matematiche
  - Manipolazione di stringhe
  - Gestione della concorrenza



# Contenuto di una libreria

- Funzioni e variabili esportate
  - Accessibili ai programmi che la utilizzano
- Funzioni e variabili private
  - Accessibili solo alle altre funzioni della stessa libreria
- Costanti ed altre risorse
- Le funzioni e le variabili esportate sono dichiarate in un file header
  - Viene incluso dai moduli sorgente che le utilizzano
  - I simboli sono preceduti dalla parola chiave «extern»

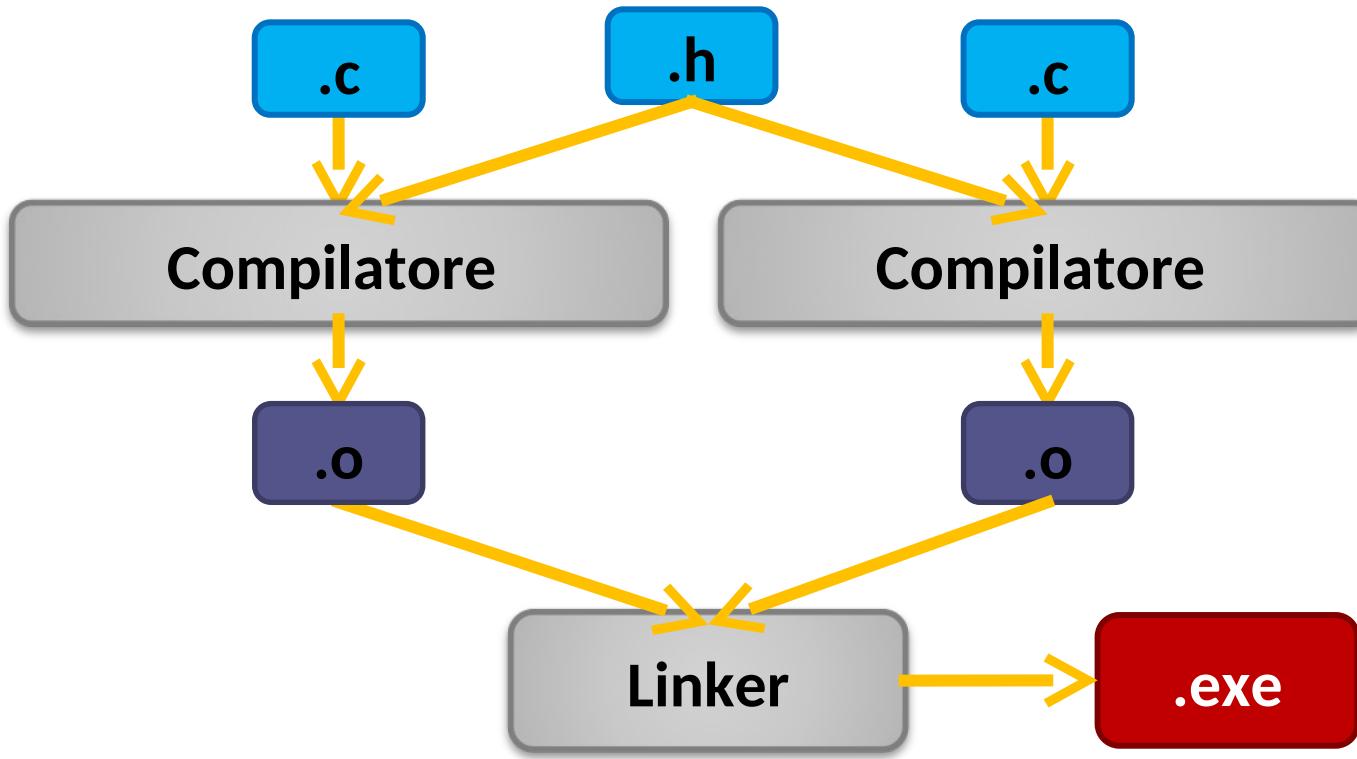


# Caricamento delle librerie

- L'uso di una libreria richiede due fasi
  - Identificazione dei moduli necessari e loro caricamento in memoria
  - Aggiornamento degli indirizzi per puntare correttamente ai moduli caricati
- Le due operazioni possono essere fatte in fasi differenti
  - Durante il collegamento (linker)
  - Durante il caricamento del programma (loader)
  - Durante l'esecuzione (programma stesso)



# Il processo di collegamento



# Tassonomia delle librerie

- Librerie statiche
- Librerie dinamiche o condivise
  - Collegate dinamicamente
  - Caricate dinamicamente

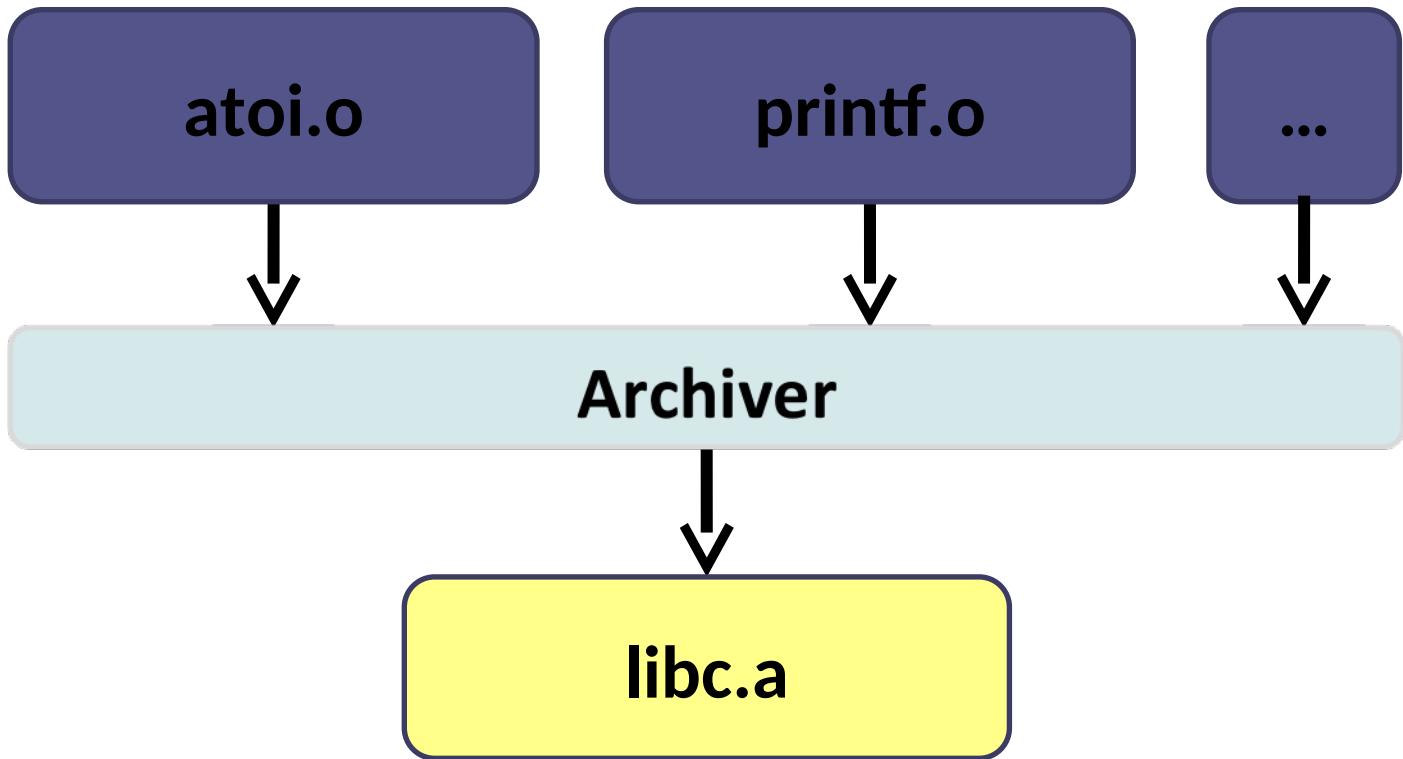


# Librerie a collegamento statico

- Contengono funzionalità collegate staticamente al codice binario cliente in fase di compilazione
- Una libreria statica è un file archivio
  - Contiene un insieme di file object, creati dal compilatore a partire dal codice sorgente
  - L'impacchettamento viene effettuato dall'archiver



# Archiver

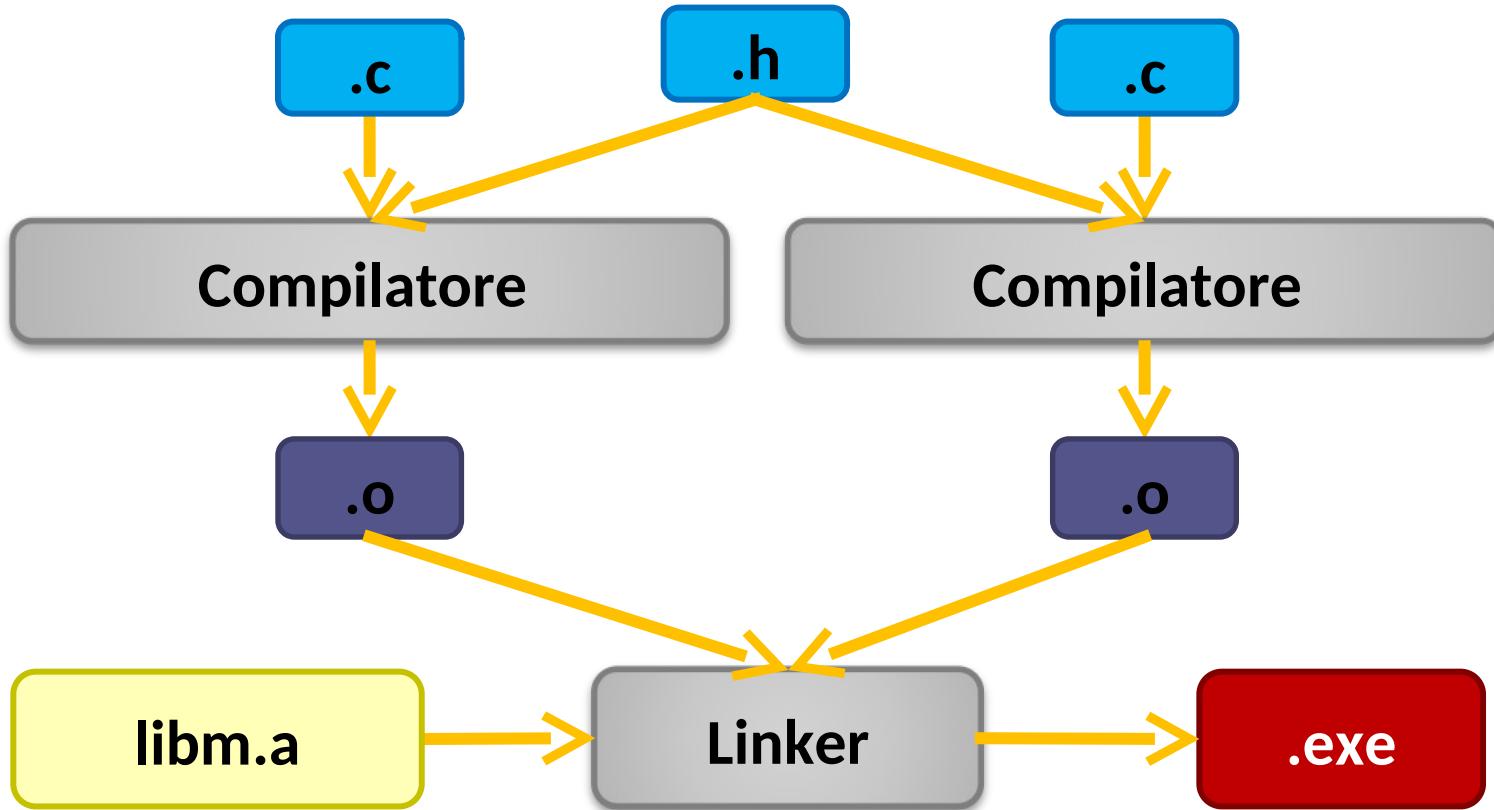


# Collegamento di una libreria statica

- Il linker identifica in quali moduli della libreria si trovano le funzioni di richiamate nel programma.
- Carica nell'eseguibile solo quelli necessari
- Terminata la fase di collegamento, i moduli ed i simboli della libreria statica risultano annessi nel codice binario originario



# Collegamento statico



# Collegamento statico

- Vantaggi
  - Il codice delle librerie è certamente presente nell'eseguibile.
  - Non ci sono dubbi sulla versione della libreria adottata
- Svantaggi
  - Gli stessi contenuti sono presenti in processi differenti
    - Il sistema non se può accorgere e utilizza pagine fisiche distinte
    - Bassa efficienza
  - L'uso delle librerie statiche comporta una riduzione della modularità del codice
    - Ogni applicazione che ne fa uso deve essere ricompilata ad ogni modifica



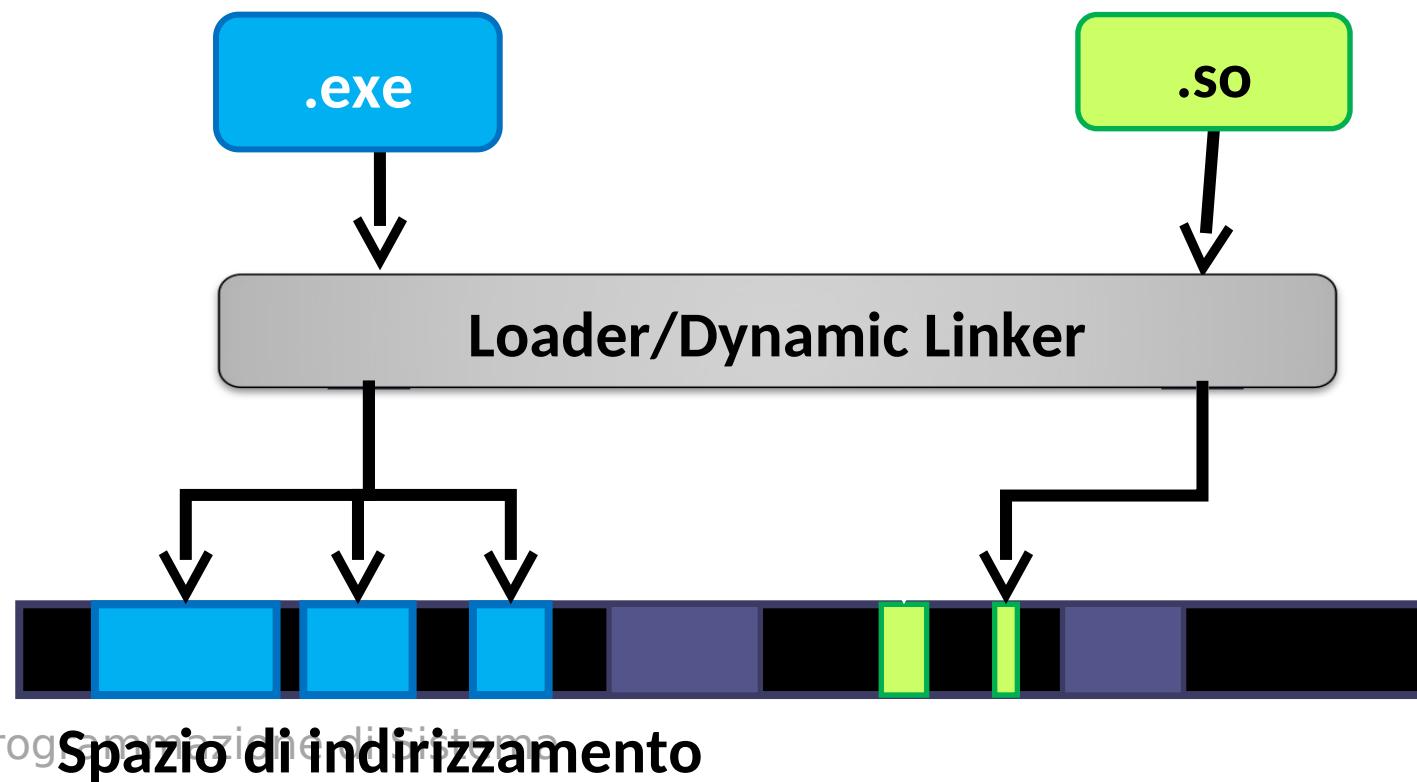
# Librerie statiche

- Linux
  - In Linux, GCC offre come archiver il tool ar
  - Per convenzione le librerie statiche cominciano con il prefisso lib e hanno un'estensione .a
- Windows
  - Hanno un'estensione .lib
  - Possono essere create a partire dai moduli oggetto con il programma “lib”
    - I Visual Studio offre un procedimento guidato per la loro creazione



# Librerie a collegamento dinamico

- Il file eseguibile non contiene i moduli della libreria
  - Vengono caricati successivamente, nello spazio di indirizzamento del processo



# Collegamento dinamico

- All'atto della creazione del processo, il loader mappa nello spazio di indirizzamento tutte le librerie condivise
  - Risolvendo i simboli corrispondenti
- In Linux il dynamic linker è il programma ld.so
  - Anch'esso è una libreria ELF condivisa, priva di ulteriori riferimenti a librerie dinamiche
  - All'avvio di un applicativo, viene mappato in memoria



# Dynamic Linker

- Mappa le librerie nello spazio di memoria del processo
- Aggiorna la tabella dei simboli (variabili e funzioni) per ogni libreria caricata



# Collegamento dinamico – Windows

- Il dynamic linker in Windows fa parte del kernel stesso
  - Funzionamento simile al caricamento dinamico di un ELF in Linux



# Caricamento dinamico

- È possibile controllare esplicitamente il caricamento delle librerie condivise
  - L'applicazione può avviarsi in assenza di qualsiasi libreria e farne richiesta quando necessario



# Caricamento dinamico – Linux

- Linux espone le Dynamic Loading API
  - `dlopen()`
    - ▀ rende un file object accessibile al programma
  - `dlSYM()`
    - ▀ recupera l'indirizzo di un simbolo (variabile o funzione) di un file object aperto
  - `dlerror()`
    - ▀ Ritorna l'ultimo errore occorso
  - `dlclose()`
    - ▀ Chiude il file object aperto



# Esempio

```
#include <stdio.h>
#include <dlfcn.h>

void invoke (char *lib, char *m, float
arg) {
 void* dl_handle = dlopen(lib,
RTLD_LAZY);
 if (!dl_handle) return;
 float (*func)(float) = dlsym(dl_handle,
m);
 if (func==NULL) return;
 printf("Result: %f\n", (*func)(arg));
 dlclose(dl_handle);
}

int main(int argc, char *argv[]){
 invoke ("libm.so", "cosf", 3.14156f);
}
```



# Librerie a caricamento dinamico - Windows

- Moduli binari in formato Portable Executable con estensione ".dll"
  - Possono contenere funzioni, variabili globali, costanti, risorse
  - Una funzione di ingresso (DlIMain)
- La funzione LoadLibrary(...) carica la libreria indicata e la mappa nello spazio di indirizzamento del processo
  - Restituisce la handle del modulo caricato



# Funzione di ingresso - Windows

- La DLL può specificare un punto di ingresso opzionale
  - Chiamato quando un processo o un thread mappano e/o rilasciano la DLL nel proprio spazio di indirizzamento
- DllMain(HINSTANCE h, DWORD r, PVOID unused)
  - h, handle, è l'indirizzo a partire dal quale è stata mappata la DLL
  - r, indica il tipo di evento (DLL\_PROCESS\_ATTACH, DLL\_PROCESS\_DETACH, ... )



# Condivisione dati DLL

- Normalmente, le variabili di una stessa DLL non sono condivise tra processi diversi
  - Ogni processo ne contiene una copia indipendente
  - Codice e risorse sono condivisi in sola lettura
- Le variabili globali esportate da una DLL sono memorizzate all'interno processo che la utilizza
  - Se più processi utilizzano la stessa DLL, esistono altrettante copie delle variabili globali
- È possibile creare zone di memoria condivise tra tutti i processi che usano una data DLL
  - Collocando le variabili globali in un apposito segmento della DLL



# Esportare simboli DLL

- Per poter esportare ed importare funzioni e strutture dati
  - Usare direttive chiave (dllexport, dllimport)
  - Creare un file module definition (.def)



# Direttive chiave

- \_\_declspec(dllexport)
  - Per importare i simboli pubblici della DLL
- \_\_declspec(dllexport)
  - Per esportare i simboli pubblici dalla DLL e renderli accessibili
- **DllExport/DllImport**
  - Tipicamente si usano all'interno di blocchi specifici
    - All'interno di un define statement per gli export
    - All'interno di un ifdef statement per gli import



# Esempio

```
// SampleDLL.h
#ifndef EXPORTING_DLL
extern __declspec(dllexport) void
HelloWorld() ;
#else
extern __declspec(dllimport) void
HelloWorld() ;
#endif
```

```
// SampleDLL.c
#define EXPORTING_DLL
#include "sampleDLL.h"

BOOL APIENTRY DllMain(...)

void HelloWorld() {printf("Hello world");}
```



# Esempio

```
// Static_Dll_usage.c
#include "sampleDLL.h"
void someMethod() {
 HelloWorld();
}
```

```
// Dynamic_Dll_usage.c
HINSTANCE hDLL
=LoadLibrary(L"sampleDLL.dll");
if (hDLL != NULL)
{
 DLLPROC Hw = (DLLPROC)
GetProcAddress(hDLL,L"HelloWorld");
 if (Hw != NULL) (*Hw)();
 FreeLibrary(hDLL);
}
```



# Direttive chiave

- In fase di compilazione il compilatore può modificare il nome della funzione esportata per tenere conto del numero e tipo dei suoi parametri (Name mangling)
  - Usare direttiva extern per impedire tale comportamento



# Module definition

- È possibile usare un file .def in cui dichiarare le funzioni DLL da esportare

```
// SampleDLL.def
//
LIBRARY "sampleDLL"

EXPORTS
 HelloWorld
```

# Compilazione di DLL

- Visual Studio genera, oltre al .dll, anche un file .lib
  - Contiene uno stub delle API offerte
  - Deve essere linkato dai progetti che intendono utilizzare in modo statico la DLL



# Uso dll - caso statico

- Il sorgente del programma che usa la DLL include il file di intestazione.
  - Si utilizzano i simboli esportati dalla DLL
- Il programma viene collegato staticamente con il file .lib associato
  - Esso carica e rilascia il .dll
  - Mappa automaticamente l'accesso a funzioni e variabili



# Uso dll - caso dinamico

- Il programma principale non fa riferimenti diretti ai simboli definiti dalla DLL
- La DLL viene caricata esplicitamente tramite LoadLibrary()
- Si accede alle funzioni tramite GetProcAddress(...)
- Si rilascia la libreria tramite FreeLibrary() o tramite FreeLibraryAndExitThread()



# Spunti di riflessione

- Si realizzi un programma che carica dinamicamente una libreria e ne invoca un metodo a scelta





# Programmazione concorrente

Programmazione di Sistema  
A.A. 2017-18

# Argomenti

- Programmazione concorrente
- Modello di esecuzione dei thread
- Sincronizzazione
- Thread nativi in Windows
- Thread nativi in Linux



# Programmazione concorrente

- Un programma concorrente dispone di due o più flussi di esecuzione
  - Nello stesso spazio di indirizzamento
  - Per perseguire un obiettivo comune
- All'atto della creazione, un processo dispone di un unico flusso di esecuzione
  - Thread principale
  - Esso può richiedere al S.O. la creazione di altri thread
- Il S.O. e/o le librerie di supporto allocano le risorse fisiche necessarie
  - Lo scheduler ripartisce, nel tempo, l'utilizzo dei core disponibili tra i diversi thread in modo non deterministico



# Vantaggi

- Sovrapposizione tra computazione e operazioni di I/O
  - È possibile sfruttare i tempi di attesa delle operazioni di I/O per eseguire altre parti dell'algoritmo
- Riduzione del sovraccarico dovuto alla comunicazione tra processi
  - Lo spazio di memoria è condiviso, quindi non occorre copiare i risultati parziali né serializzarne i contenuti
- Utilizzo delle CPU multicore
  - Vero parallelismo
  - Più flussi di esecuzione possono svolgersi contemporaneamente, riducendo il tempo totale di elaborazione



# Svantaggi

- Aumento significativo della complessità del programma
  - Nuove fonti e tipologie di errore
  - Non determinismo dell'esecuzione



# Complessità

- La memoria non può più essere pensata come un “deposito statico”
  - I dati scritti possono cambiare in conseguenza dell’attività di altri thread
- I thread devono coordinare l’accesso alla memoria
  - Tramite opportuni costrutti di sincronizzazione



# Errori

- L'uso superficiale dei costrutti di sincronizzazione porta a blocchi passivi o attivi del programma ...
- Si possono causare malfunzionamenti casuali
  - Dovuti al comportamento non deterministico e asincrono dell'esecuzione concorrente
  - Estremamente difficili da riprodurre e da eliminare
- Gli errori possono manifestarsi cambiando la piattaforma di esecuzione
  - Oppure soltanto dopo numerose esecuzioni

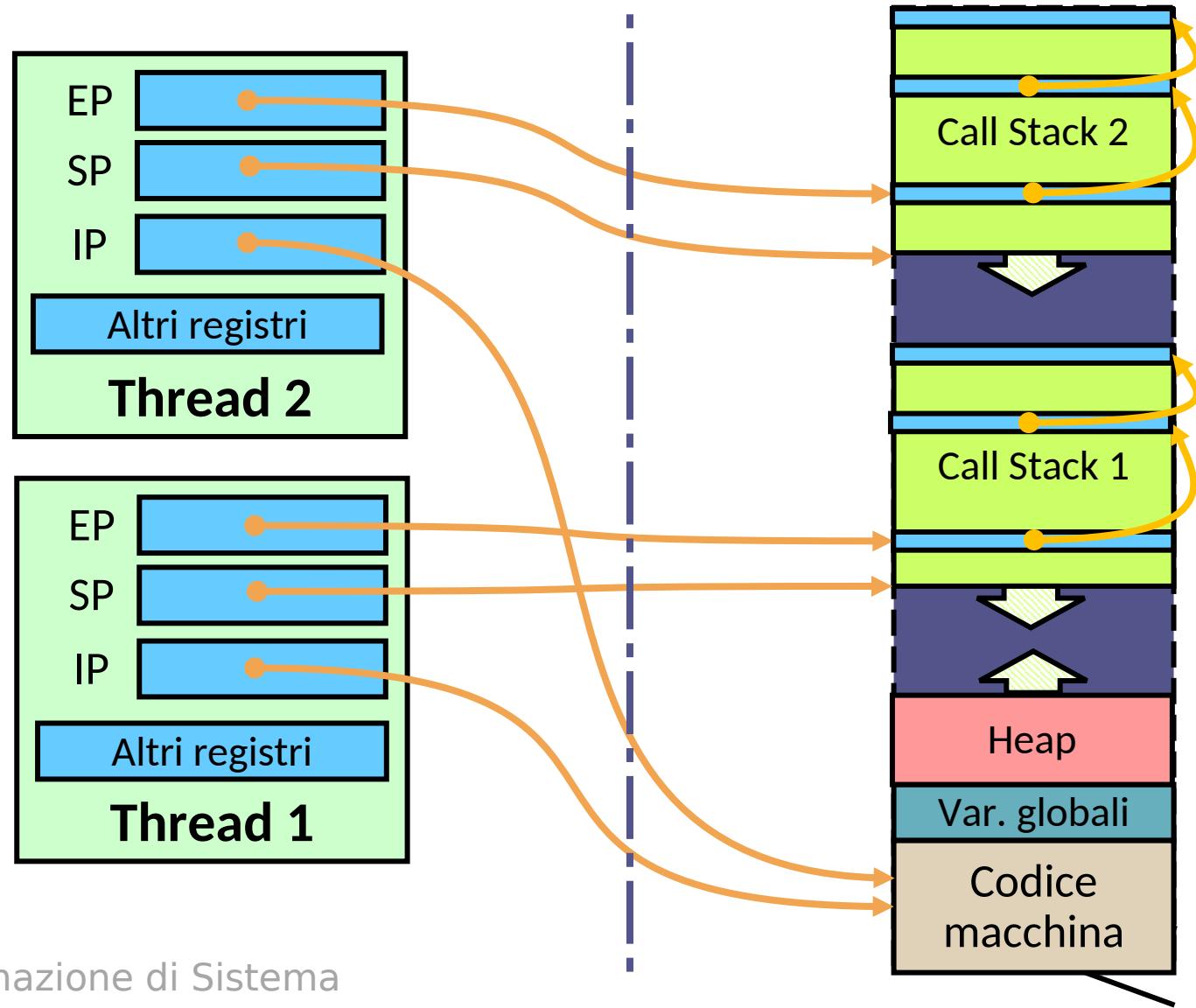


# Thread e memoria

- Ogni thread dispone di
  - Un proprio stack delle chiamate
  - Un proprio puntatore all'ultimo contesto per la gestione delle eccezioni
  - Lo stato del proprio “processore virtuale”
- I thread dello stesso processo condividono
  - Le variabili globali
  - L'area in cui è memorizzato il codice
  - L'area delle costanti
  - Lo heap



# Thread e memoria



# Esecuzione concorrente

- L'esecuzione si alterna
  - Ogni thread accede a variabili locali elementari, memorizzate nello stack (il cui puntatore può essere dischiuso ad altri thread), a dati condivisi sullo heap ed alle variabili globali
- Se due o più attività cooperano per raggiungere un obiettivo comune
  - Occorre regolare lo svolgimento di un thread anche in base a quanto sta succedendo negli altri
  - Occorre essere in grado di comunicare delle informazioni e sapere quando esse sono valide/disponibili



# Esempio

```
int data[10];

void thread1() {
 for (int i=0; i<10; i++)
 data[i]= calcola_nuovo_valore();
}

void thread2() {
 for (int i=0; i<10; i++)
 usa_valore(data[i]);
 //Quando è pronto questo dato?
}
```



# Esecuzione e non determinismo

- L'esecuzione di un singolo thread procede secondo le normali regole sequenziali
  - Per cui è possibile dire cosa avvenga «prima» e cosa «dopo»
- Se più thread sono in esecuzione, non è possibile fare assunzioni sulle velocità relative di avanzamento
- L'esecuzione ripetuta dello stesso programma (con gli stessi ingressi) può portare a risultati differenti
  - A seguito delle interazioni (complesse) tra il S.O. e i dispositivi HW



# Esempio

```
#include <thread>
#include <iostream>
#include <string>

void run(std::string msg) {
 for (int j=0; j<10; j++) {
 std::string s= msg + std::to_string(j) + "\n";
 std::cout << s;
 }
}

int main() {
 std::thread t1(run, "aaaa");
 std::thread t2(run," bbbb");
 t1.join();
 t2.join();
}
```

aaaa0  
bbbb0  
aaaa1  
bbbb1  
aaaa2  
bbbb2  
aaaa3  
bbbb3  
aaaa4  
bbbb4  
aaaa5  
aaaa6  
aaaa7  
bbbb5  
aaaa8  
aaaa9  
bbbb6  
bbbb7  
bbbb8  
bbbb9

# Osservazioni

- L'output del programma precedente è solo uno dei possibili risultati
  - Se lo stesso programma viene eseguito più volte, si ottengono risultati differenti
- È assolutamente possibile (e a volte succede) che tutte le righe di uno dei thread precedano quelle dell'altro
- L'unica certezza è che le righe che cominciano con "aaaa" sono tra loro ordinate in modo crescente
  - Così come le righe che cominciano con "bbbb"
- Il non determinismo dà origine a comportamenti del tutto inattesi in un contesto di elaborazione sequenziale



# Esempio

```
#include <iostream>
#include <thread>

int a=0;

void run() {
 while (a>=0) {
 int before=a;
 a++;
 int after=a;
 if (after-before!=1)
 std::cout<< before<<" -> "<<
 after<<" ("
 << after-before<<")\n";
 }
}
```



# Esempio

```
//creo due thread e ne attendo la
terminazione

int main() {
 std::thread t1(run);
 std::thread t2(run);

 t1.join();
 t2.join();
}
```



# Esecuzione

119944578 -> 119944552 (-26)  
123397102 -> 123397584 (482)  
128314912 -> 128314956 (44)  
395835151 -> 395835236 (85)  
396049424 -> 396098482 (49058)  
412859791 -> 412859826 (35)  
419214490 -> 419214537 (47)  
419406880 -> 419406877 (-3)  
433982464 -> 433982472 (8)  
436005364 -> 436215900 (210536)  
441453011 -> 441454010 (999)  
446802106 -> 446802106 (0)

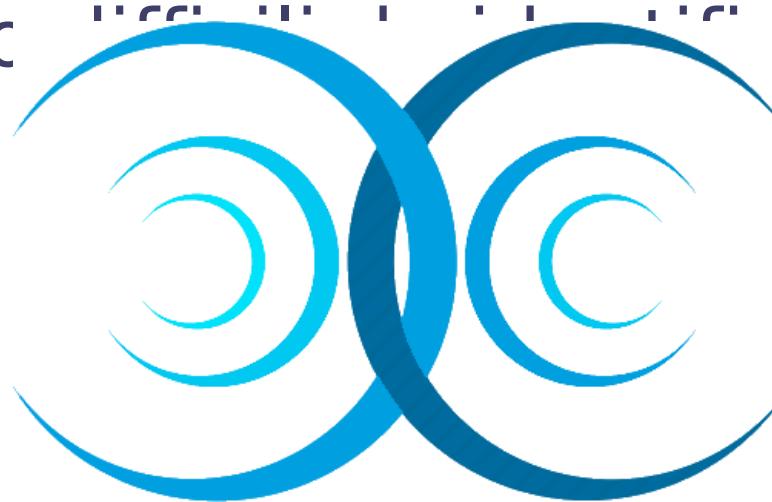
# Domande

- Esaminando l'uscita del programma precedente si vedono molti casi in cui la differenza tra after e before è superiore a 1
  - In alcuni casi tale valore è anche molto grande: perché?
- Talora capita che la differenza sia nulla o negativa
  - Come è possibile, se entrambi i flussi incrementano sempre la variabile a?



# Interferenza

- Si verifica quando più thread fanno accesso ad un medesimo dato, modificandolo
  - La sua presenza dà origine a **malfunzionamenti casuali**, molti di cui difficili da individuare

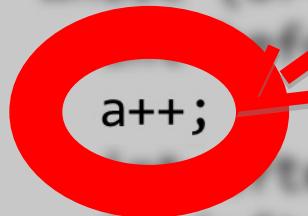


# Interferenza

```
#include <iostream>
#include <thread>

int a=0;

void run() {
 while (a<8) {
 a=a+1;
 if (a>5)
 std::cout<< before(<< " -> " << after(<< "("
 << after-before(<<")\n";
 }
}
```



Sembra un'azione  
innocente, ma nasconde  
due operazioni in cascata:  
**int temp = a;**  
**a = temp+1;**

# Sincronizzazione

- È necessario evitare che altri thread accedano ad una risorsa condivisa mentre una modifica è in corso
  - Meccanismo che regoli l'accesso alle zone pericolose
- Ogni S.O. offre strumenti leggermente diversi, sia in termini di strutture dati che di API
  - La libreria C++ 2011 introduce un meccanismo standard per la sincronizzazione, indipendente dalla piattaforma soggiacente



# Sincronizzazione

- Windows
  - CriticalSection
  - ConditionVariable
  - Oggetti kernel
    - Mutex, Event, Semaphore, Pipe, Mailslot, ...
- Linux
  - Oggetti della libreria PThreads
    - pthread\_mutex,pthread\_cond
  - Oggetti kernel
    - Semafori, pipe, segnali, futex



# Correttezza

- Occorre fare in modo che non capiti mai che un thread “operi” su un dato, alterandone il contenuto
  - Mentre un altro sta già operando sullo stesso oggetto
- In particolare, non devono essere visibili stati “transitori” dell’oggetto
  - Dovuti al meccanismo di aggiornamento
- Gli oggetti condivisi mutabili devono godere di questa proprietà
  - Questi mantengono al proprio interno degli “invarianti”
  - Perché gli oggetti immutabili non sono soggetti ad interferenza?



# Correttezza

- Bisogna impedire che gli invarianti siano violati
  - Si effettuano le mutazioni (cambi di stato) con metodi che garantiscono la validità degli invarianti prima e dopo l'esecuzione e che bloccano l'accesso concorrente
- Si accede allo stato attraverso altri metodi
  - Che controllano che non ci sia una mutazione in corso
  - E che impediscono che essa inizi mentre si sta facendo accesso allo stato condiviso
- È compito del programmatore riconoscere quando e dove utilizzare la sincronizzazione
  - Un uso sbagliato porta a risultati disastrosi



# Accesso condiviso: le cause del problema

- Atomicità
  - Quali operazioni di memoria hanno effetti indivisibili?
- Visibilità
  - La scrittura di una variabile può essere osservata da una lettura eseguita da un altro thread?
- Ordinamento
  - Sotto quali condizioni, sequenze di operazioni effettuate da un thread sono visibili nello stesso ordine da parte di altri thread?



# Accesso non sincronizzato ad un dato

- Se due thread fanno accesso allo **stessa struttura dati**, rispettivamente in lettura e scrittura
  - Non c'è nessuna garanzia su quale delle due operazioni sia eseguita per prima



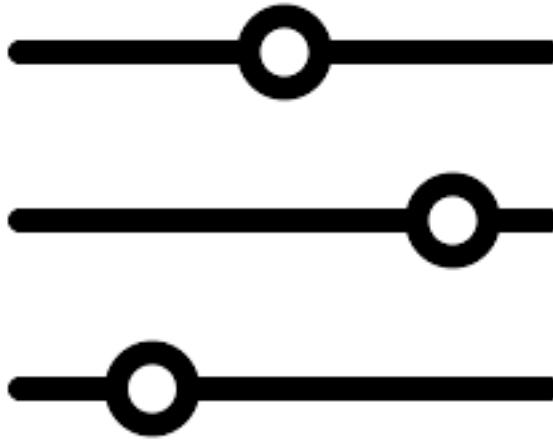
# Accesso ad un dato mentre un modifica è in corso

- Se un thread legge un dato che un altro thread sta modificando
  - Il valore letto può essere **diverso** sia dal valore iniziale che da quello finale



# Ri-ordinamento

- Se, quando osservato dall'esterno, il comportamento di un singolo thread appare indistinguibile a seguito di una modifica
  - Sia il **compilatore** che la **CPU** possono invertire l'ordine delle esecuzione delle singole istruzioni



# Accesso non sincronizzato

```
int val; //variabili globale
void f(int v) { ... }

void thread1() {
 // invoco f con |val|
 if (val >=0)
 f(val) // non è detto che a
questo
 // punto val sia ancora
>=0
} else {
 f(-val); //idem
}
}
```



# Accesso non sincronizzato

```
std::vector<int> v; //globale

void threadFunc() {
 //...
 if (!v.empty()) {
 std::cout<<v.front()<<std::endl;
 // anche qui, v.front()
 // potrebbe non esistere più
 }
}
```

# Accesso non sincronizzato

- In generale, non è lecito operare né il lettura né in scrittura su una qualsiasi struttura dati mentre è in corso un'altra scrittura
- Fanno eccezione
  - L'accesso a elementi distinti di uno stesso contenitore
  - L'uso dei flussi di caratteri std::cin, std::cout, std::cerr



# Lettura durante una modifica

- Se, nello stesso istante, due thread differenti leggono e scrivono una stessa cella di memoria il risultato è impredicibile
  - Potrebbe essere restituito il valore precedente alla scrittura
  - Potrebbe essere restituito il valore scritto
  - Potrebbe essere restituito un qualsiasi altro valore

# I problemi introdotti

- Mancata sincronizzazione
  - Malfunzionamenti casuali anche gravi
- Errata sincronizzazione
  - Blocco
- Eccessiva sincronizzazione
  - Scarse prestazioni

# Meccanismi di sincronizzazione

- La complessità di gestione della sincronizzazione spinge all'utilizzo di tecniche note ed affidabili (pattern)
  - È sempre necessaria molta cautela nella realizzazione di programmi multi-thread



# Uso dei thread

- Le API dei S.O. permettono la gestione del ciclo di vita dei thread
  - Creazione e terminazione di thread
  - Meccanismi di sincronizzazione
  - Aree private di memoria
- I dettagli relativi a ciascuna piattaforma differiscono alquanto
  - Rendendo complessa la portabilità delle applicazioni
- La versione 2011 del linguaggio C++ ha introdotto una standardizzazione nella creazione e gestione dei thread



# Thread in windows

- Creazione di un thread

```
HANDLE WINAPI CreateThread(
 LPSECURITY_ATTRIBUTES
 lpThreadAttributes,
 SIZE_T
 dwStackSize,
 LPTHREAD_START_ROUTINE
 lpStartAddress,
 LPVOID
 lpParameter,
 DWORD
 dwCreationFlags,
 LPDWORD
 lpThreadId
) ;
```



# Parametri

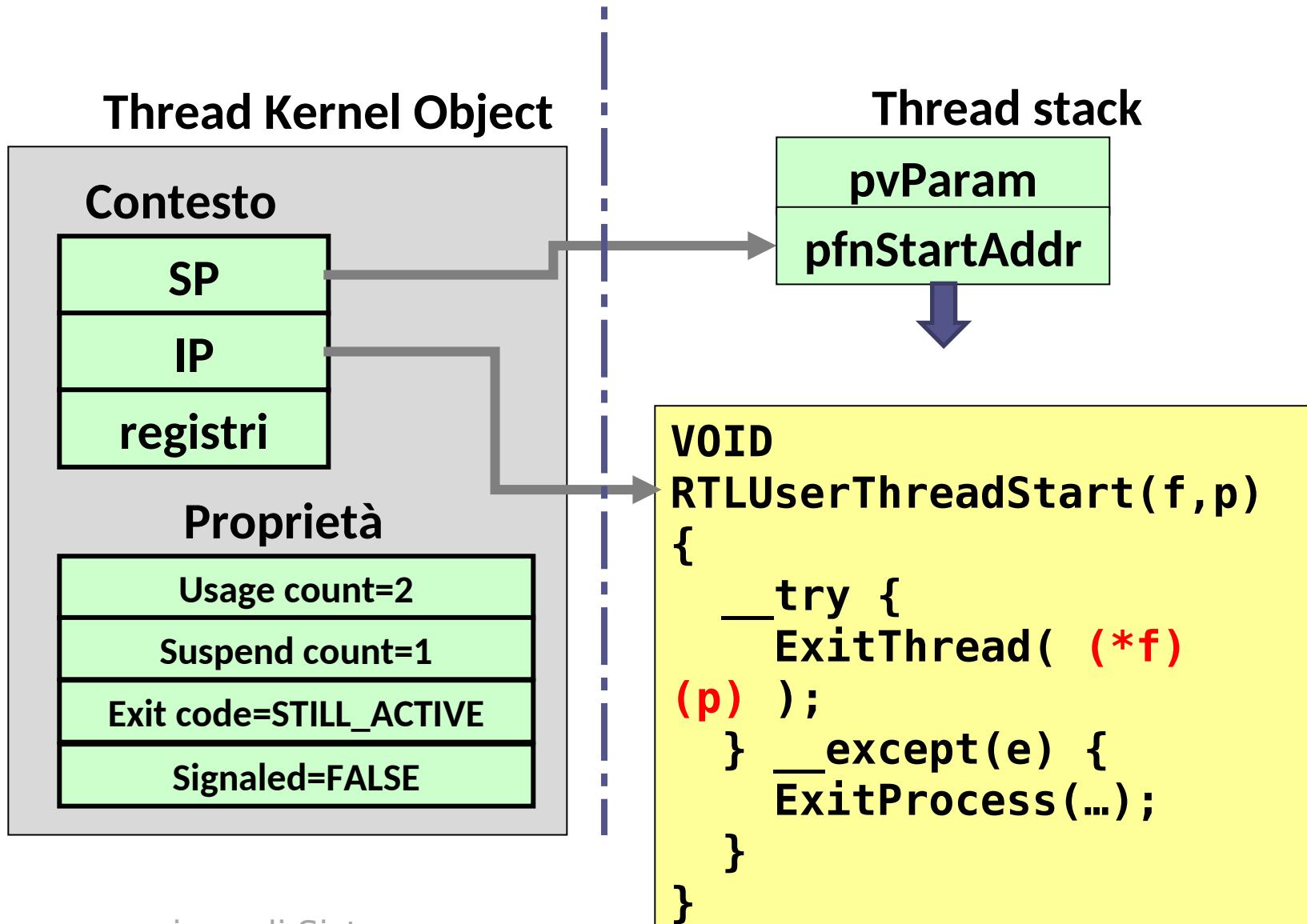
- **lpThreadAttributes**
  - Puntatore al contesto di sicurezza in cui il thread verrà eseguito
- **stackSize**
  - Dimensione dello stack
- **lpStartAddress**
  - Indirizzo della funzione principale del thread
- **lpParameter**
  - Puntatore ad un eventuale parametro passato alla funzione principale
- **dwCreationFlag**
  - Indica se il thread dovrà essere attivato subito (0) o sospeso (**CREATE\_SUSPENDED**)
- **lpThreadId**
  - Indirizzo di una variabile in cui sarà memorizzato l'id del thread



# CreateThread

- La funzione ritorna un riferimento opaco (handle)
  - Utilizzato per fare riferimento al thread nelle altre funzioni del S.O.
  - NULL indica un fallimento





# Thread e librerie C/C++

- La libreria standard del C utilizza variabili globali il cui accesso non è sincronizzato
  - Se usate (direttamente o indirettamente) da un thread secondario, il risultato non è deterministico
- Con i compilatori Microsoft, sono disponibili le funzioni wrapper
  - `_beginthreadex(...)`
  - `_exitthreadex(...)`
  - Definite in `<process.h>`
- Garantiscono che vengano create e rilasciate versioni locali al thread delle variabili globali della libreria standard



# Identificare un thread

- HANDLE GetCurrentThread()
  - Restituisce una pseudo-handle del thread corrente utilizzabile solo al suo interno
  - Per essere passata ad altri richiede una duplicazione esplicita (DuplicateHandle)



# Terminare un thread

- Un thread termina quando
  - La funzione associata al thread ritorna
  - Il thread invoca ExitThread al proprio interno
  - Un altro thread del processo invoca TerminateThread
- Un thread termina quando
  - Vengonoificate le funzioni ExitProcess(...) o TerminateProcess(...) specificando il processo cui il thread appartiene
  - Attenzione al codice di startup!



# Terminare un thread

- In C++, è opportuno evitare di chiamare ExitThread e ExitProcess
  - I distruttori degli oggetti allocati nello stack del thread e quelli delle variabili globali non verrebbero eseguiti
- Fare in modo che un thread causi la terminazione ordinata di un altro thread può essere complesso

# Terminare un thread

- Per garantire la corretta distruzione di tutti gli oggetti nello stack del thread, occorre basarsi su una variabile condivisa
  - Che deve essere ispezionata periodicamente dal thread che si vuole terminare
- Quando questa assume un dato valore, il thread fa in modo di ritornare dalla propria routine principale
  - Occorre che sia dichiarata volatile e che l'accesso avvenga in modo sincronizzato



# Attendere la terminazione di un thread

- Quando un thread termina, l'oggetto kernel corrispondente passa nello stato "segnalato"
  - Si può attendere tale condizione senza consumare CPU attraverso le funzioni di attesa



# Funzioni di attesa

```
DWORD WINAPI WaitForSingleObject(
 HANDLE hHandle,
 DWORD dwMilliseconds
) ;
```

```
DWORD WINAPI WaitForMultipleObjects(
 DWORD nCount,
 const HANDLE *lpHandles,
 BOOL bWaitAll,
 DWORD dwMilliseconds
) ;
```



# Handle e oggetti kernel

- Le handle dei thread fanno riferimento ad oggetti posseduti dal S.O.
  - Contengono un contatore di utilizzo
  - Quando diventa zero, il kernel può distruggere l'oggetto
- Quando un thread non ha più bisogno di usare una handle, deve segnalarlo tramite `CloseHandle()`
  - Decrementa il contatore dell'oggetto e rende invalida la handle



# Handle e oggetti kernel

- Nel caso dei thread, il contatore inizialmente vale 2
  - L'oggetto corrispondente è accessibile al thread creatore ed al thread creato
- Se non vengono rilasciati, gli oggetti kernel possono saturare la memoria di sistema
  - Problematico nel caso di processi che durano a lungo



# Thread in Linux

- Linux offre un insieme di chiamate a sistema per la gestione dei thread
  - Interfaccia a basso livello, complessa da utilizzare
  - Usate dalla libreria PThread per offrire un accesso più semplice



# La libreria PThread

- Libreria per la gestione di thread per i sistemi UNIX
  - Specificata dallo standard IEEE POSIX 1003.1c
  - Definita nel file pthread.h
- Offre un'interfaccia basata sul linguaggio C
  - Strutture dati e funzioni per la gestione del ciclo di vita di un thread
  - Per la compilazione ed il collegamento, usare il flag  
"-pthread"



# Tipi e funzioni

- `pthread_t`
  - Identifica in modo univoco un thread in un processo
- `pthread_t pthread_self()`
  - Identifica il thread corrente
- `int pthread_equal(t1,t2)`
  - Confronta due identificatori



# Creazione di thread

```
int pthread_create(
 pthread_t* tidp,
 pthread_attr_t* attr,
 void* (* function)(void*),
 void* arg
);

// restituisce 0 in caso di successo
```



# Attendere la terminazione

```
int pthread_join(
 pthread_t tid,
 void** retvalp
);

// restituisce 0 in caso di successo
// retvalp punta ad un blocco in cui
// viene salvato il valore ritornato
```



# Spunti di riflessione

- Si scriva un programma Linux che crei un thread secondario e che ne attenda la terminazione
- Si riimplementi il programma in Windows

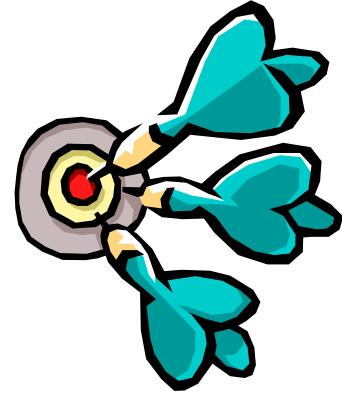




# Programmazione concorrente in C++11 - Parte I

Programmazione di Sistema  
A.A. 2017-18

# Argomenti



- Esecuzione asincrona
- Attesa dei risultati
- Sincronizzazione

# Programmazione concorrente in C++11

- Il concetto di thread è parte integrale dello standard del linguaggio
  - Facilita la scrittura di programmi portabili
- Introduce un livello di astrazione più elevato che facilita il programmatore
  - Permettendogli di concentrarsi sugli aspetti specifici del proprio algoritmo invece che sui dettagli del S.O.



# Programmazione concorrente in C++11

- Sono disponibili due approcci
  - Uno di alto livello, basato su std::async e std::future
  - Uno di basso livello che richiede l'uso esplicito di thread e costrutti di sincronizzazione
- Entrambi richiedono un compilatore aggiornato
  - VisualStudio a partire dalla versione 2012
  - GCC a partire dalla versione 4.7.x (con parte delle funzionalità già presenti in 4.6.y)



# Esecuzione asincrona e risultati futuri

- Spesso, un compito complesso può essere decomposto in una serie di compiti più semplici
  - I cui risultati potranno essere poi combinati per fornire la funzionalità complessiva
- Due sotto-compiti sono tra loro indipendenti se
  - La computazione di uno non dipende da quella di un altro
  - Non fanno accesso, in scrittura, a dati condivisi
- Due sotto-compiti indipendenti possono essere eseguiti in parallelo
  - Combinando poi i risultati quando sono disponibili



# Esecuzione asincrona e risultati futuri

- La funzione std::async e la classe std::future rendono molto semplice questo tipo di decomposizione
  - Entrambe sono definite nel file <future>



# std::async()

- Prende come parametro un oggetto chiamabile
  - Puntatore a funzione o oggetto funzionale che ritornano un dato di tipo generico T ed eventuali parametri da passare
  - Restituisce un oggetto di tipo std::future<T>
- Quando questa funzione viene eseguita, se possibile, invoca in un thread separato l'oggetto chiamabile con i relativi parametri
  - In ogni caso, ritorna immediatamente, senza attenderne il completamento della funzione chiamata



# Accedere al risultato

- Si accede al risultato dell'esecuzione invocando il metodo `get()` sull'oggetto `future` ritornato
  - Se l'esecuzione è andata a buon fine, restituisce il risultato
  - Se il thread secondario è terminato con un'eccezione, la rilancia nel thread corrente
  - Se l'esecuzione è ancora in corso, si blocca in attesa che finisca
  - Se l'esecuzione non è ancora iniziata, ne forza l'avvio nel thread corrente



# Esempio

```
#include <future>
#include <string>
std::string f1(std::string p1, double p2) { ... }
std::string f2(int p) { ... }

int main() {
 // calcola f1("...", 3.14) + f2(18)
 std::future<std::string> future1 =
 std::async(f1, "...", 3.14);
 std::string res2 = f2(18);
 std::string res1 = future1.get();
 std::string result = res1+res2;
}
```

Mentre f1() viene calcolato, esegue f2 nel thread principale



# std::async

- La funzione accetta eventuali parametri da passare all'oggetto chiamabile
  - std::async(f1, "...", 3.14);
  - L'oggetto chiamabile può essere preceduto da una politica di lancio
- std::launch::async
  - Attiva un thread secondario
  - Lancia un'eccezione di tipo system\_error se il multithreading non è supportato o non ci sono risorse per creare un nuovo thread
- std::launch::deferred
  - Valutazione pigra: l'oggetto chiamabile sarà valutato solo se e quando qualcuno chiamerà get() o wait() sul future relativo
  - Non verrà generato alcun thread aggiuntivo
- Se la politica viene omessa
  - Il sistema prova dapprima ad attivare un thread secondario
  - Se non può farlo, segna l'attività come deferred



# Attenzione!

- La creazione di un thread secondario ha un costo significativo
  - Se le operazioni da eseguire sono poche (dell'ordine di un migliaio di cicli macchina) conviene eseguirle direttamente

# Esempio

```
template <typename Iter>
int parallel_sum(Iter beg, Iter end)
{
 typename Iter::difference_type len = end-beg;
 if(len < 1000) // pochi elementi,
 // conviene valutazione sincrona
 return std::accumulate(beg, end, 0);

 Iter mid = beg + len/2;
 auto handle = std::async(std::launch::async,
 parallel_sum<Iter>, mid,
end);
 int sum = parallel_sum(beg, mid);
 return sum + handle.get();
}

int main()
{
 std::vector<int> v(10000, 1);
 std::cout << "Somma:" << parallel_sum(v.begin(),
v.end()) << '\n';
}
```



# std::future<T>

- Fornisce un meccanismo per accedere in modo sicuro e ordinato al risultato di un'operazione asincrona
  - Il tipo T rappresenta il tipo del risultato ritornato
- Mantiene internamente uno stato condiviso con il blocco di codice responsabile della produzione del risultato
- Quando si invoca il metodo get() lo stato condiviso viene rimosso
  - L'oggetto future entra in uno stato invalido
  - get() può essere chiamato UNA SOLA VOLTA



# `std::future<T>`

- Per forzare l'avvio del task e attenderne la terminazione, senza prelevare il risultato, si utilizza il metodo `wait()`
  - Può essere chiamato più volte: se il task è già terminato, ritorna immediatamente
- È possibile attendere per un po' di tempo il completamento del compito, attraverso i metodi `wait_for(...)` e `wait_until(...)`



# `std::future<T>`

- `Wait_for` e `wait_until` NON forzano l'avvio
  - Se il task è stato lanciato con la modalità deferred
- Richiedono un parametro, rispettivamente di tipo
  - `std::chrono::duration`
  - `std::chrono::time_point`
- Indicando una durata nulla, permettono di scoprire se il task sia o meno già terminato



# std::future<T>

- Restituiscono
  - std::future\_status::deferred se la funzione non è ancora partita
  - std::future\_status::ready se il risultato era già pronto o lo è diventato nel tempo di attesa
  - std::future\_status::timeout, se il tempo è scaduto, senza che il risultato sia diventato pronto



# Attenzione!

- Quando un oggetto future viene distrutto, il distruttore ne attende la fine
  - Se la computazione è ancora attiva, questo può comportare attese significative
- Le attività lanciate in questo modo, non sono cancellabili, se non avendo cura di condividere, con la funzione chiamata, una variabile che possa essere usata come criterio di terminazione
  - Attenzione al riordinamento introdotto dal compilatore ed alle barriere di memoria



# Esempio

```
int quickComputation(); //approssima il risultato con un'euristica
int accurateComputation(); // trova la soluzione esatta,
 // richiede abbastanza tempo
std::future<int> f; // dichiarato all'esterno perché il suo ciclo di
 // vita
 // potrebbe estendersi più a lungo della
funzione
int bestResultInTime()
{
 // definisce il tempo disponibile
 auto tp = std::chrono::system_clock::now() +
 std::chrono::minutes(1);
 // inizia entrambi i procedimenti
 f = std::async (std::launch::async, accurateComputation);
 int guess = quickComputation();
 // trova il risultato accurato, se disponibile in tempo
 std::future_status s = f.wait_until(tp);
 // ritorna il miglior risultato disponibile
 if (s == std::future_status::ready) {
 return f.get();
 } else {
 return guess; // accurateComputation() continua...
 }
}
```



# **std::shared\_future<T>**

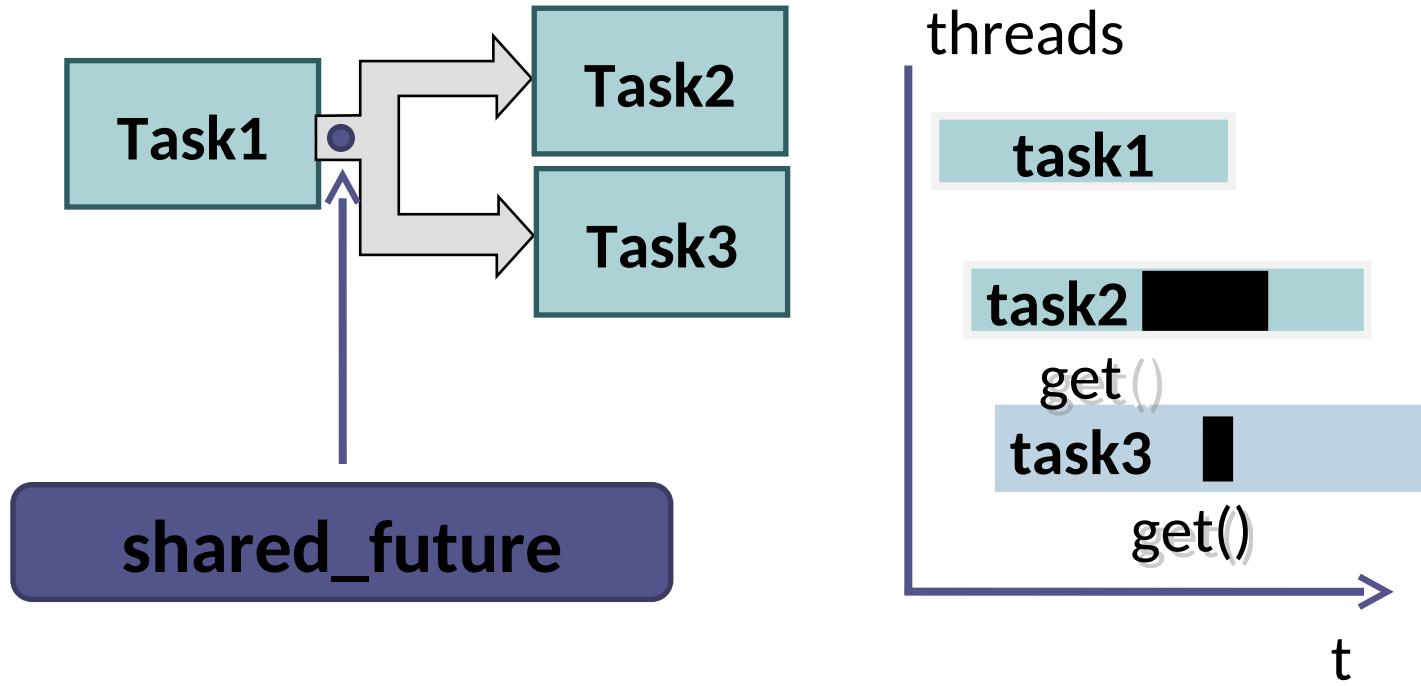
- La classe `future<T>` mette a disposizione il metodo `share()`
  - Utile se in più posti occorre poter valutare se un'operazione asincrona sia terminata e quale risultato abbia prodotto
- Restituisce un oggetto di tipo `std::shared_future<T>`
- Invalida lo stato dell'oggetto corrente
  - Che non può più essere usato

# `std::shared_future<T>`

- `std::shared_future<T>` è copiabile oltre che movibile
  - A differenza di `std::future<T>` che può solo essere mosso
- A parte `share()`, offre gli stessi metodi di `future<T>`
  - Se `get()` viene chiamato più volte, produce sempre lo stesso risultato (a parte l'attesa)
- Per realizzare catene di elaborazione asincrone
  - In cui i risultati delle fasi iniziali sono messi a disposizione delle fasi successive
  - Senza ridurre, a priori, il grado di parallelismo



# `std::shared_future<T>`



# **std::shared\_future<T>**

```
int task1(); // prima fase del calcolo

std::string task2(std::shared_future<int>); // seconda fase
double task3(std::shared_future<int>); // terza fase

int main() {
 std::shared_future<int> f1 =
 std::async(std::launch::async, task1).share();
 std::future<std::string> f2= std::async(task2, f1);
 std::future<double> f3 = std::async(task3, f1);

 try {
 std::string str = f2.get();
 double d= f3.get();
 } catch (std::exception& e) {
 //gestisci eventuali eccezioni
 }
}
```



# Accedere a dati condivisi

- Capita frequentemente che due thread differenti debbano fare accesso ad una stessa informazione
  - In assenza di sincronizzazione, gli accessi potrebbero sovrapporsi e dare origine a interferenza
- Occorre proteggere gli accessi condivisi attraverso un opportuno meccanismo
  - La principale astrazione messa a disposizione dalla libreria C++11 è quella offerta dalla classe std::mutex



# std::mutex

- Gli oggetti di questa classe permettono l'accesso controllato a porzioni di codice ad un solo thread alla volta
  - Mutex: contrazione di Mutual Exclusion
  - Definito nel file <mutex>
- Tutto il codice che fa accesso ad una data informazione condivisa deve fare riferimento ad uno stesso oggetto mutex
  - E racchiudere le operazioni tra le chiamate ai metodi lock() e unlock()



# std::mutex

- lock() e unlock() entrambe includono una barriera di memoria
  - Garantisce la visibilità delle operazioni eseguite fino a quel punto
- Se un thread invoca il metodo lock() di un mutex mentre questo è posseduto da un altro thread, il primo thread si blocca in attesa che l'altro chiami unlock()
- Occorre fare in modo che, se si è preso possesso di un mutex tramite lock(), prima o poi lo si rilasci con unlock()



```
std::list<int> l;
std::mutex m;
```

```
void add(int i){
```

```
 m.lock();
```

```
 l.push_back(i);
```

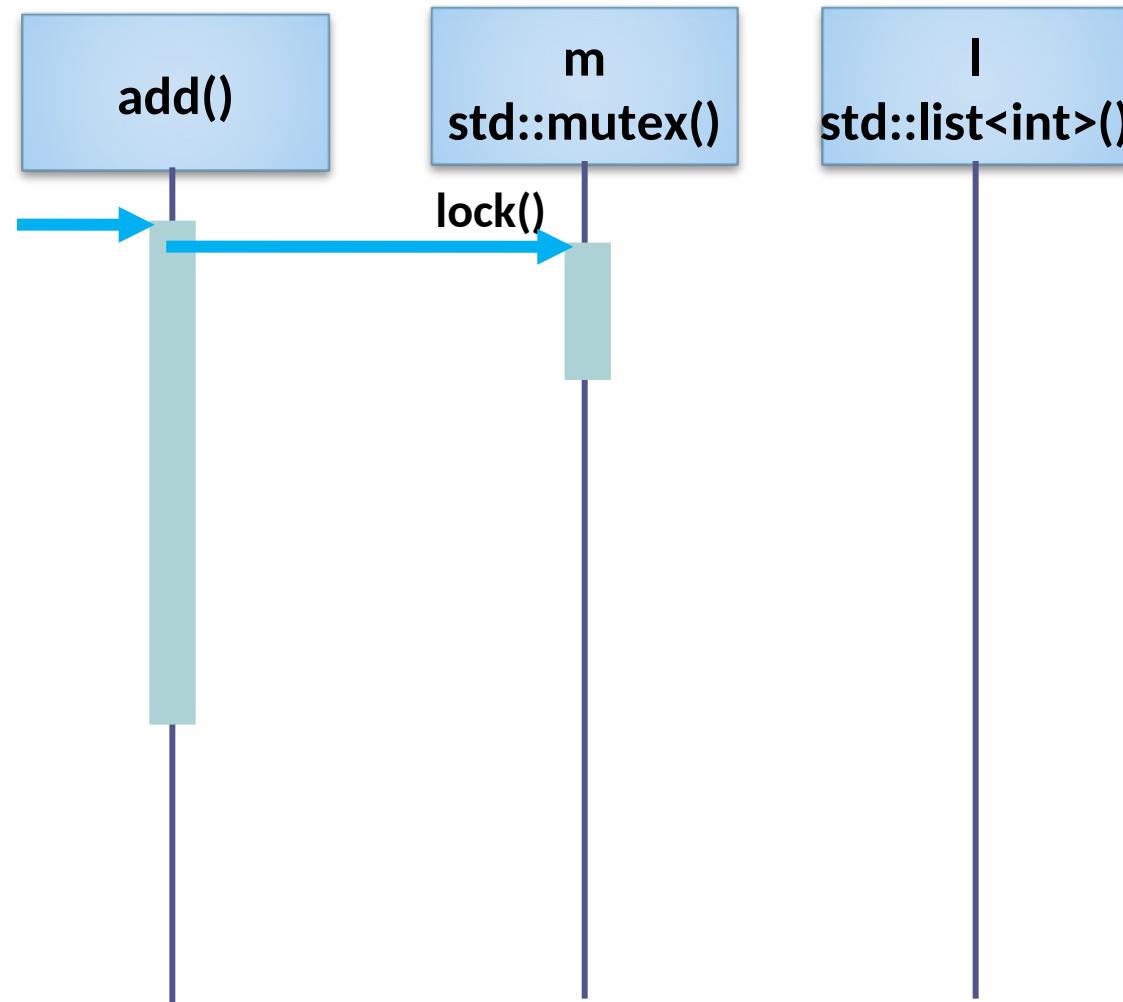
```
 m.unlock();
```

```
}
```

add()

m  
std::mutex()

l  
std::list<int>()





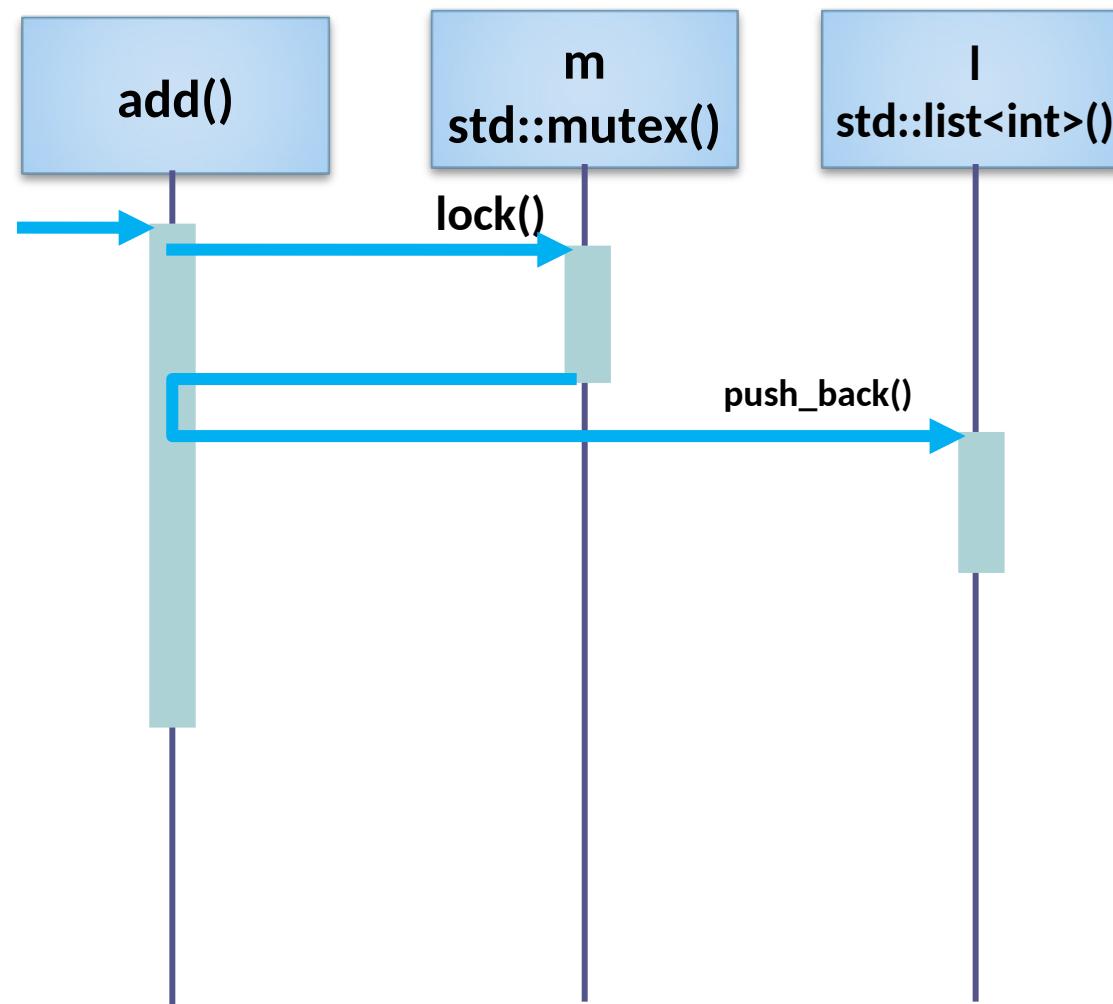
```
std::list<int> l;
std::mutex m;

void add(int i){

 m.lock();

 l.push_back(i);

 m.unlock();
}
```



```
std::list<int> l;
std::mutex m;
```

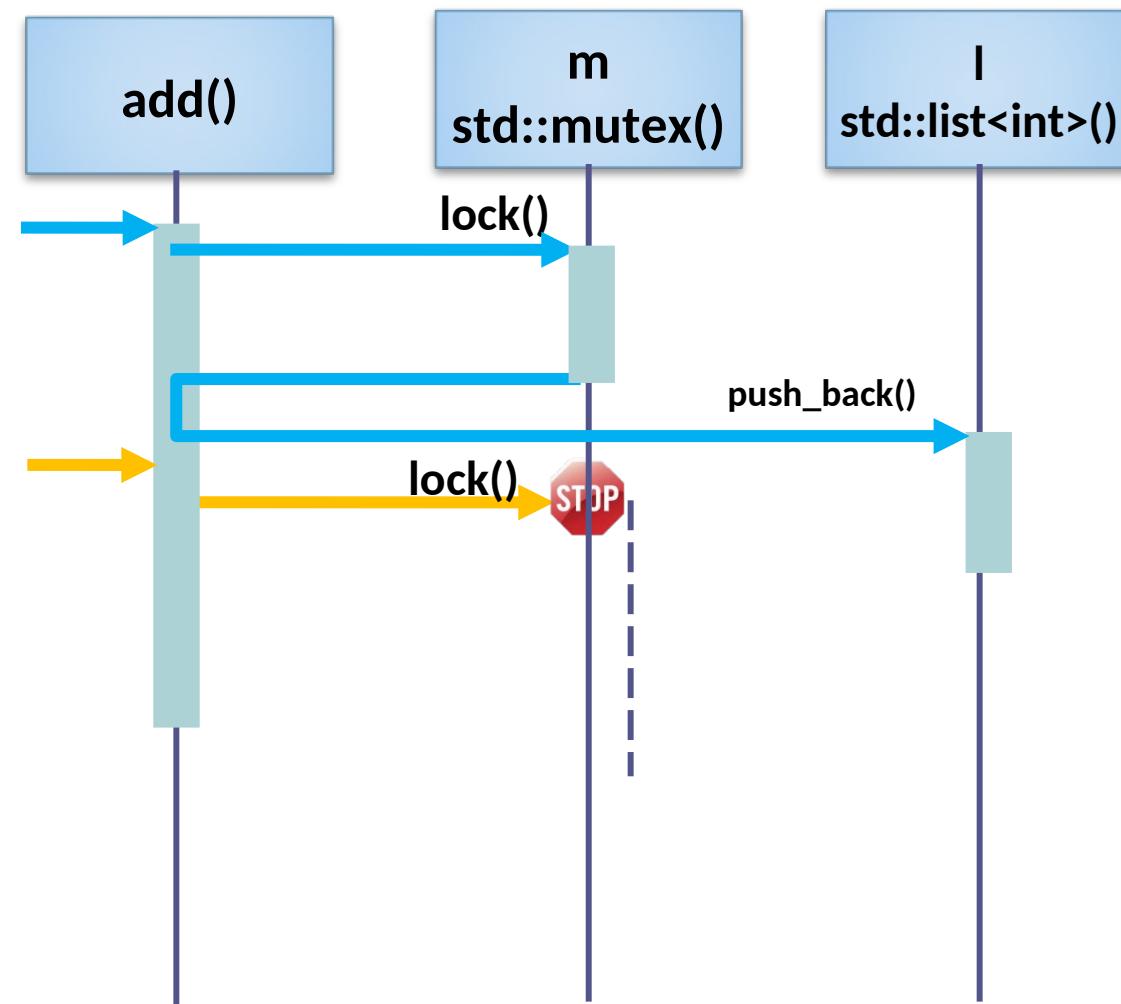
```
void add(int i){
```

```
 m.lock();
```

```
 l.push_back(i);
```

```
 m.unlock();
```

```
}
```

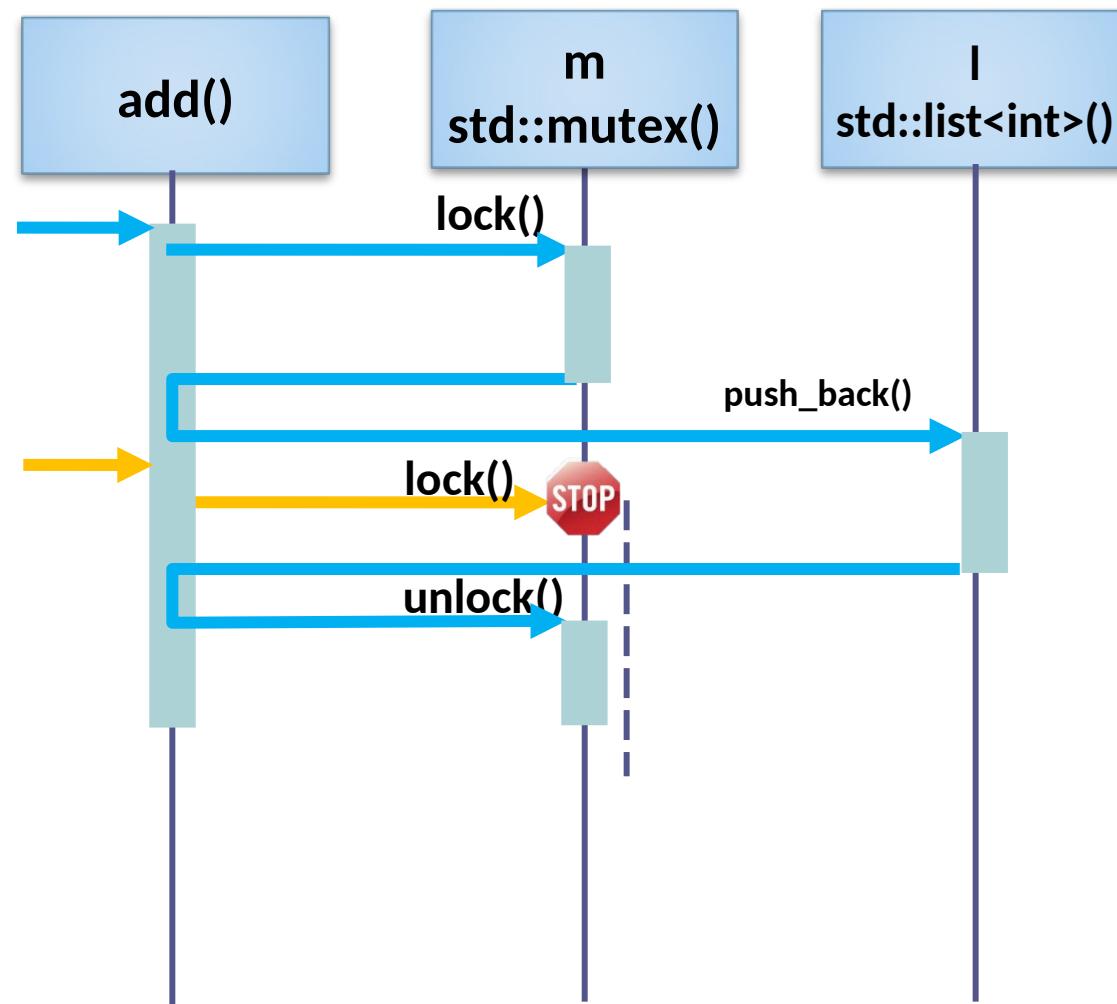


```

std::list<int> l;
std::mutex m;

void add(int i){
 m.lock();
 l.push_back(i);
 m.unlock();
}

```

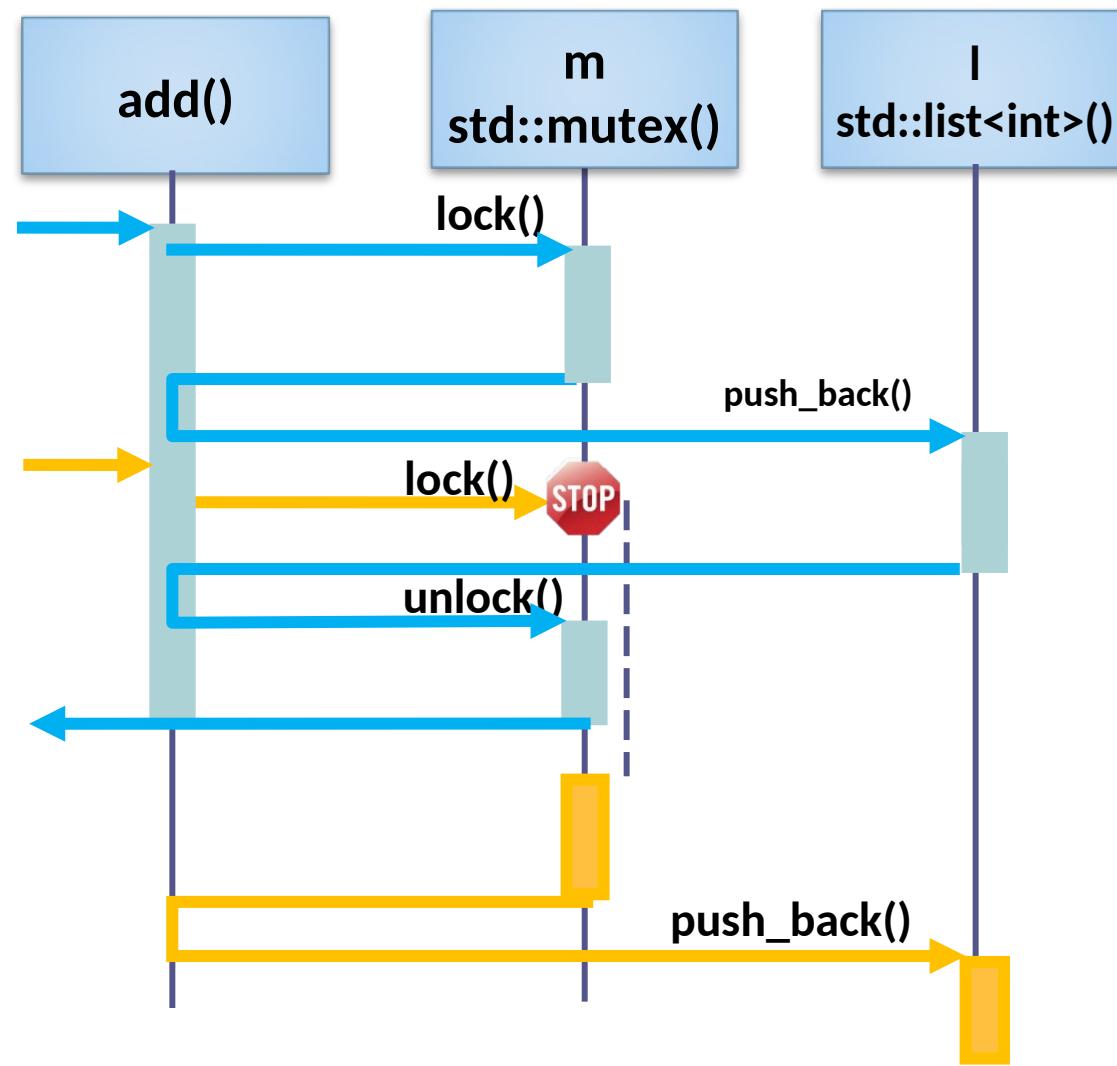


```

std::list<int> l;
std::mutex m;

void add(int i){
 m.lock();
 l.push_back(i);
 m.unlock();
}

```



# std::mutex

- Non c'è corrispondenza diretta tra l'oggetto mutex e la struttura dati che esso protegge
  - La relazione è nella testa del programmatore
- Tutti gli accessi alla struttura vanno protetti acquisendo dapprima il mutex
  - Anche le operazioni in sola lettura
- Un oggetto mutex può proteggere molte strutture diverse
  - Ma riduce il grado di parallelismo complessivo del programma
- Un mutex non è ricorsivo
  - Se un thread cerca di acquisirlo (tramite lock()) due volte, senza rilasciarlo, il thread si blocca per sempre



# Regolare l'attesa

- `std::recursive_mutex` può essere acquisito più volte consecutivamente dallo stesso thread
  - Dovrà essere rilasciato altrettante volte, prima di consentire ad un altro thread di acquisirlo
- `std::timed_mutex` aggiunge i metodi `try_lock_for()` e `try_lock_until()`
  - Pone un limite al tempo di attesa massimo
  - Se il tempo scade senza che lock venga acquisito, restituiscono false



# Regolare l'attesa

- `std::recursive_timed_mutex` unisce i due comportamenti
  - Il lock può essere acquisito più volte da parte dello stesso thread
  - È possibile limitare il tempo massimo di attesa



# std::mutex

- Ogni volta che si acquisisce un mutex, occorre rilasciarlo
  - Anche se si verifica un'eccezione



# `std::lock_guard<Lockable>`

- Per semplificare il codice e garantire che un mutex sia sempre rilasciato, viene messa a disposizione la classe generica `std::lock_guard<Lockable>`
  - Utilizza il paradigma RAII
- Il costruttore invoca il metodo `lock()` dell'oggetto passato come parametro
  - Il distruttore invoca `unlock()`
- Non offre nessun altro metodo



# Esempio

```
template <class T>
class shared_list {
 std::list<T> list;
 std::mutex m;

 T& operator=(const shared_list<T>& that);
 shared_list(const shared_list<T>& that);

public:

 int size() { std::lock_guard<std::mutex> l(m); return
list.size(); }

 T front() { std::lock_guard<std::mutex> l(m); return
list.front(); }

 void push_front(T t) {
 std::lock_guard<std::mutex> l(m);
 list.push_front(t);
 }

}; //ecc.
```

# Blocco condizionale

- In alcune situazioni, un programma non intende bloccarsi se non può acquisire un mutex
  - Ma fare altro nell'attesa che si liberi
- La classe mutex mette a disposizione il metodo `try_lock()`
  - Restituisce un valore booleano per indicare se è stato possibile acquisirlo o meno
- Se l'acquisizione ha avuto successo, il mutex può essere "adottato" da un oggetto `lock_guard`
  - Così da garantirne il rilascio al termine dell'utilizzo



# Esempio

```
#include <mutex>

std::mutex m;

...
void someFunction() {
 while (m.try_lock() == false) {
 do_some_work();
 }

 std::lock_guard<std::mutex> l(m, std::adopt_lock);

 // l registra m al proprio interno, senza cercare di
 // acquisirlo

 ...
}

// quando l viene distrutto, rilascia il possesso del mutex
}
```



# `std::unique_lock<Lockable>`

- Estende il comportamento di `lock_guard`
  - È possibile rilasciare e riacquisire l'oggetto `Lockable`
  - Tramite i metodi `lock()` e `unlock()`
- I metodi `lock()` e `unlock()` lanciano un'eccezione di tipo `std::system_error`
  - se si cerca di rilasciare qualcosa che non si possiede o viceversa



# `std::unique_lock<Lockable>`

- Il costruttore offre numerose politiche di gestione selezionate in base al secondo parametro
  - `adopt_lock` verifica che il thread possieda già il Locakble passato come parametro e lo adotta
  - `defer_lock` si limita a registrare il riferimento al Lockable, senza cercare di acquisirlo

# Spunti di riflessione

- Si scriva un programma C++11 che utilizzi la funzione `async` per effettuare le statistiche sul numero di vocali presenti in due file di testo e stampi i risultati nel thread principale





# Programmazione concorrente in C++11 - Parte II

Programmazione di Sistema  
A.A. 2017-18

# Argomenti



- Creazione di thread secondari
- Restituzione dei risultati
- Accedere al thread corrente

# Gestire la concorrenza a basso livello

- `async()` e `future<T>` permettono di creare facilmente composizioni di attività ad alto livello
  - A patto che i diversi compiti in cui l'algoritmo può essere suddiviso siano poco interconnessi
- In altri casi, occorre accedere alle funzionalità di basso livello
  - Gestendo esplicitamente la creazione dei thread, la sincronizzazione, l'accesso a zone di memoria condivise, l'uso delle risorse



# La classe std::thread

- Modella un oggetto che rappresenta un thread del sistema operativo
  - Offrendo un'interfaccia omogenea e portabile per la sua creazione e gestione
- Se, come parametro del costruttore, riceve un oggetto di tipo Callable...
  - Un puntatore a funzione o un oggetto che implementa operator()
- ... crea un nuovo thread all'interno del S.O. e ne inizia subito l'esecuzione



# Esempio

```
#include <thread>
void f() {
 std::cout<<"Up &
Running!"<<std::endl;
}

int main() {
 std::thread t(f); //inizia ...
 //altre operazioni
 //nel thread principale
 t.join(); // Si blocca fino a che
 // il thread t termina
}
```



# Specificare il compito da eseguire

- Il costruttore di thread riceve un oggetto chiamabile e, facoltativamente, una serie di parametri
  - Tali parametri saranno inoltrati all'oggetto chiamabile attraverso la funzione `std::forward` che mette a disposizione del destinatario (la funzione che deve essere invocata all'interno dell'oggetto chiamabile) un riferimento al dato originale o un riferimento RVALUE in funzione della natura del dato passato (RVALUE o LVALUE)
  - Per passare un dato come riferimento, occorre incapsularlo in un oggetto di tipo `std::reference_wrapper<T>` (tramite le funzioni `std::ref(v)` e `std:: cref(v)` definite in `<functional>`)



# Specificare il compito da eseguire

- I parametri passati per riferimento possono essere utilizzati per ospitare i valori di ritorno della funzione
  - In questo caso occorre garantire che il ciclo di vita di tali parametri sopravviva alla terminazione del thread
  - Ed attendere che il thread sia terminato prima di esaminarne il contenuto

# Specificare il compito da

```
#include <thread>
void f(int& result) {
 result = ... ;
}

int main() {
 int res1,res2;
 std::thread t1(f,std::ref(res1));
 std::thread t2(f,std::ref(res2));

 t1.join();
 t1.join();
 std::cout<<res1<<" "<<res2<<"\n";
}
```



# Specificare il compito attraverso un funtore

- Spesso è comodo utilizzare una funzione lambda come parametro
  - I parametri catturati come reference potranno essere modificati dal thread creato e resi visibili al codice del thread creante
- Questo può creare problemi di sincronizzazione e di accesso alla memoria
  - Se, nel frattempo, i parametri escono dal proprio scope

# Differenze tra std::thread e std::async

- Non c'è scelta sulla politica di attivazione
  - Creando un oggetto di tipo thread con un parametro chiamabile, la libreria cerca di creare un thread nativo del sistema operativo
  - Se non ci sono le risorse necessarie, la creazione dell'oggetto std::thread fallisce lanciando un'eccezione di tipo std::system\_error



# Differenze tra std::thread e std::async

- Non c'è un meccanismo standard per accedere al risultato
  - L'unica informazione che viene associata al thread è un identificativo univoco, accessibile tramite il metodo `get_id()`
- Se durante la computazione del thread si verifica un'eccezione non recuperata da un blocco `catch`, l'intero programma termina
  - Viene chiamato il metodo `std::terminate()`



# Ciclo di vita di un thread

- Quando viene creato un oggetto `std::thread` occorre alternativamente
  - Attendere la terminazione della computazione parallela, invocando il metodo `join()`
  - Informare l'ambiente di esecuzione che non si è interessati all'esito della sua computazione, invocando il metodo `detach()`
  - Trasferire le informazioni contenute nell'oggetto `thread` in un altro oggetto, tramite movimento



# Ciclo di vita di un thread

- Se nessuna di queste azioni avviene, e si distrugge l'oggetto thread, l'intero programma termina
  - Sempre invocando std::terminate()
- Se il thread principale di un programma termina, tutti i thread secondari ancora esistenti terminano improvvisamente
  - Senza possibilità di effettuare nessuna forma di salvataggio

# Gestire la terminazione

```
#include <thread>
class thread_guard {
 std::thread& t;
public:
 thread_guard(std::thread& t_): t(t_) {}

 ~thread_guard() {
 if(t.joinable()) t.join();
 }

 thread_guard(thread_guard const&) =delete;
 thread_guard& operator=(thread_guard const&)=delete;
};
```



# Restituire un risultato

- Se un thread calcola un risultato, come fa a metterlo a disposizione degli altri thread?
  - La soluzione richiede appoggiarsi ad una variabile condivisa
- Questo introduce un secondo problema:
  - Come fanno gli altri thread a sapere che il contenuto della variabile condivisa è stato aggiornato?
- La soluzione banale di introdurre un'ulteriore variabile (booleana) che indica la validità del dato non è una



# Restituire un risultato

- Diventerebbe infatti un'ulteriore variabile condivisa che avrebbe bisogno di un indicatore di validità...
  - Problema del riordinamento
- Il modo più semplice di restituire un valore calcolato all'interno di un thread è basato sull'utilizzo di un oggetto di tipo `std::promise<T>`

# std::promise<T>

- Rappresenta l'impegno, da parte del thread, a produrre, prima o poi, un oggetto di tipo T da mettere a disposizione di chi lo vorrà utilizzare...
- ...oppure di notificare un'eventuale eccezione che abbia impedito il calcolo dell'oggetto
- Dato un oggetto promise, si può conoscere quando la promessa si avvera, richiedendo l'oggetto std::future<T> corrispondente
  - Attraverso il metodo `get_future()`

# Esempio

```
#include <future>

void f(std::promise<std::string> p) {
 try {
 //Calcolo il valore da
 restituire...

 std::string result = ...;
 p.set_value(std::move(result));
 } catch (...) {
 p.set_exception(
 std::current_exception());
 }
}
```

# Esempio

```
int main() {
 std::promise<std::string> p;
 std::future<std::string>

 f=p.get_future();
 //creo un thread, forzando p ad
 essere
 //passata per movimento
 std::thread t(f,std::move(p));
 t.detach();

 // faccio altro...

 // accedo al risultato del thread
 std::string res=f.get();
```

# Thread distaccati

- Se si crea un oggetto thread e si invoca il metodo `detach()`, l'oggetto thread si "stacca" dal flusso di elaborazione corrispondente...
- ... e può continuare la propria esecuzione senza, però, offrire più nessun meccanismo specifico per sapere quando termini
- Se il thread distaccato fa accesso a variabili globali o statiche, queste potrebbero essere distrutte mentre la computazione è in corso

Programmazione di Sistemi Perché sta terminando il thread principale



# Thread distaccati

- Se il thread principale termina normalmente (il main() ritorna)
  - L'intero processo viene terminato, con tutti i thread distaccati eventualmente presenti
- Se la terminazione del thread principale avviene per altre cause (si invoca l'API del S.O. che termina il thread corrente) questo può portare a errori sulla memoria
- std::quick\_exit(...) permette di far terminare un programma senza invocare i distruttori della variabili globali e statiche
  - Che può essere un rimedio peggiore del male



# Promesse e corse critiche

- Se si utilizza un oggetto di tipo promise per restituire un valore, si introduce un livello di sincronizzazione
- Il thread interessato ai risultati, invocando `wait()` o `get()` sul future corrispondente resta bloccato fino a che il thread `detached` non ha assegnato un valore
- Questo, però, non significa che il thread `detached` sia terminato
  - Potrebbe avere ancora del codice da eseguire (ad esempio, tutti i distruttori degli oggetti finora utilizzati)



# Promesse e corse critiche

- La classe promise offre due metodi per evitare potenziali corse critiche
  - Tra la pubblicazione di un risultato nell'oggetto promise e la continuazione degli altri thread in attesa dello stesso
  - `set_value_at_thread_exit(T val);`
  - `set_exception_at_thread_exit(std::exception_ptr p);`

# `std::packaged_task<T(Args... .) >`

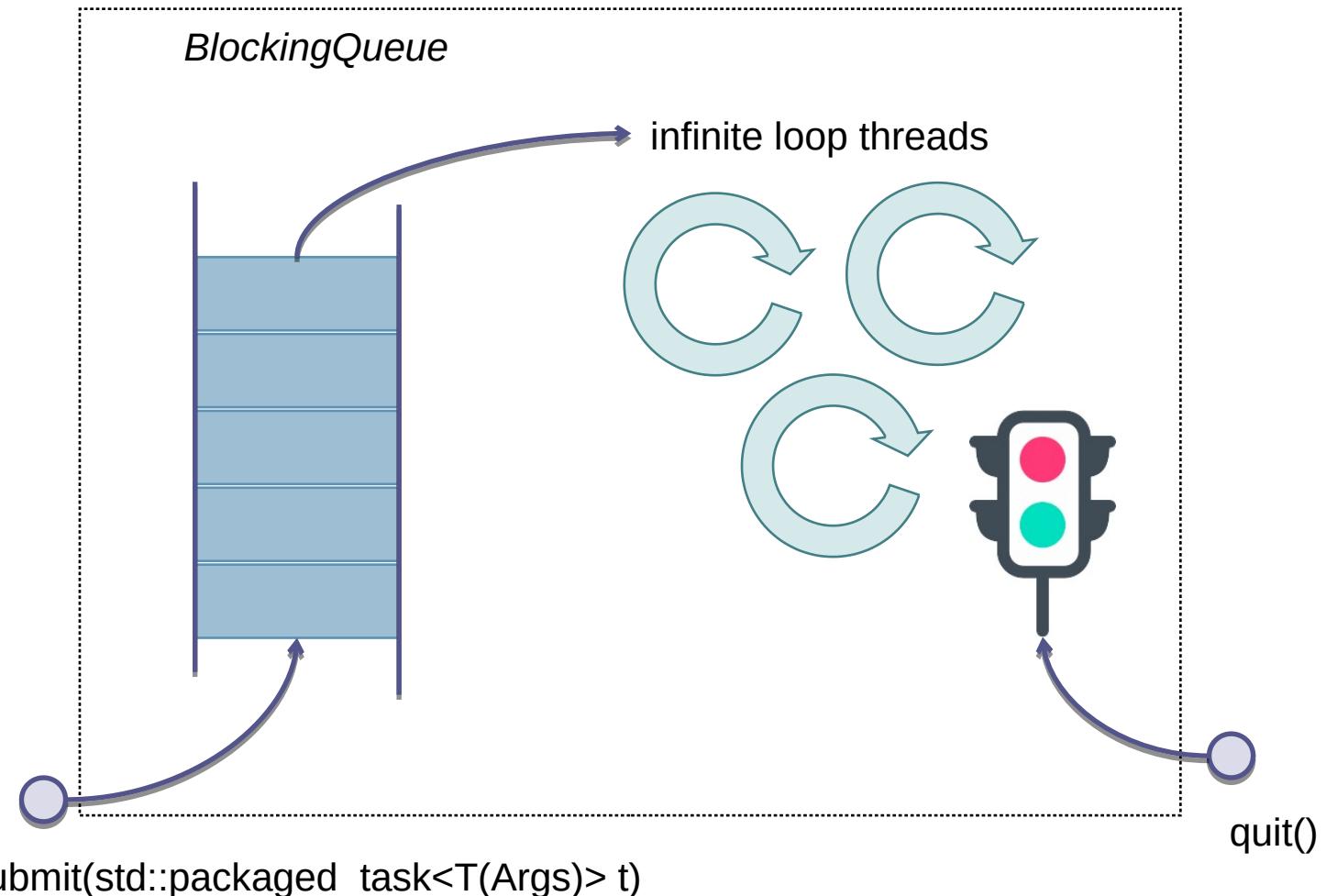
- Astrazione di un'attività in grado di produrre, prima o poi, un oggetto di tipo T
  - Come conseguenza dell'esecuzione di una funzione (o di un funzionale) che viene incapsulata nell'oggetto stesso, insieme ai suoi argomenti
- Quando un `packaged_task` viene eseguito, la funzione incapsulata viene invocata
  - Il risultato prodotto (o l'eventuale eccezione generata) viene utilizzato per valorizzare l'oggetto `std::future` associato al task

# Thread pool

- In molte situazioni, è conveniente creare un numero limitato di thread (legato al numero di core disponibili) cui demandare l'esecuzione di compiti puntuali
  - Tali compiti non sono noti a priori e possono essere eseguiti da uno qualsiasi dei thread appositamente creati
  - La classe `packaged_task` si presta particolarmente per la realizzazione di una coda in cui ospitare le attività richieste e da cui i diversi thread del pool possono attingere le attività da svolgere
- Il numero di core disponibili può essere stimato attraverso la funzione
  - `std::thread::hardware_concurrency();`



# Thread pool



# Conoscere la propria identità

- Lo spazio dei nomi `std::this_thread` offre un insieme di funzioni che permettono di interagire con il thread corrente
- La funzione `get_id()` restituisce l'identificativo del thread corrente

# Sospendere l'esecuzione

- La funzione `sleep_for(duration)` sospende l'esecuzione del thread corrente per almeno il tempo indicato come parametro
- La funzione `sleep_until(time_point)` interrompe l'esecuzione almeno fino al momento indicato
- La funzione `yield()` offre al sistema operativo la possibilità di rischedulare l'esecuzione del thread

# Spunti di riflessione

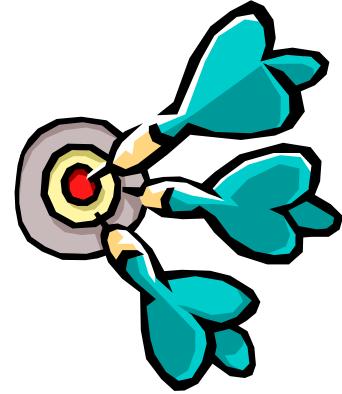
- Si realizzi un programma che ricerchi una stringa in un elenco di file di testo
  - La ricerca in ciascun file è affidata a un thread separato, creato servendosi della classe std::thread
  - Il thread principale raccoglie e stampa i risultati



# Programmazione concorrente in C++11 - Sincronizzazione

Programmazione di Sistema  
A.A. 2017-18

# Argomenti



- Operazioni atomiche
- Condition variable
- Esecuzione di task singoli

# Operazioni atomiche

- Le normali operazioni di accesso in lettura e scrittura non offrono nessuna garanzia sulla visibilità delle operazioni che sono eseguite in parallelo da più thread
- I processori supportano alcune istruzioni specializzate per permettere l'accesso atomico ad un singolo valore

# La classe std::atomic<T>

- Offre la possibilità di accedere in modo atomico al tipo T
  - Garantendo che gli accessi concorrenti alla variabile sono osservabili nell'ordine in cui avvengono
  - Questo garantisce il meccanismo minimo di sincronizzazione
- Le operazioni di lettura e scrittura di questi oggetti contengono al proprio interno istruzioni di memory fence
  - Che garantiscono che il sottosistema di memoria non mascheri il valore corrente della variabile



# La classe std::atomic<T>

- Le operazioni non possono essere riordinate
  - Utili per segnalare condizioni di terminazione
  - Oppure per generare dipendenze di tipo "happens\_before" tra attività differenti

# La classe std::atomic<T>

```
std::atomic<boolean> done=false;

void task1() {
 //continua ad elaborare fino a che non viene detto di
 smettere
 while (! done.load()) {
 process();
 }
}

void task2() {
 wait_for_some_condition();
 //segnala che il task1 deve finire
 done.store(true);
 //...
}

void main() {
 auto f1=std::async(task1);
 auto f2=std::async(task2);
}
```



# La classe std::atomic<T>

- Le operazioni di inizializzazione non sono atomiche
  - Quelle di accesso tramite load() e store(T t), sì
- Operazioni atomiche di tipo Read/Modify/Write
  - `fetch_add(val)` - aggiunge val al valore corrente (`+=`)
  - `fetch_sub(val)` - sottrae val dal valore corrente (`-=`)
  - `operator++()` - equivalente a `fetch_add(1)`



# La classe std::atomic<T>

- Operazioni atomiche di tipo Read/Modify/Write
  - operator--() - equivalente a fetch\_sub(1)
  - exchange(val) - assegna val e ritorna il valore precedente
- Il template offre alcune specializzazioni per i tipi int e boolean

# Gestire il risveglio

- Spesso un thread deve aspettare uno o più risultati intermedi prodotti altri thread
  - Per motivi di efficienza, l'attesa non deve consumare risorse e deve terminare non appena un dato è disponibile
- La coppia di classi promise/future offrono una soluzione limitata del problema
  - Valida quando occorre notificare la disponibilità di un solo dato



# Gestire il risveglio

- La presenza di dati condivisi richiede come minimo l'utilizzo di un mutex
  - Per garantire l'assenza di interferenze tra i due thread che devono fare accesso ai dati
- Il polling ha due limiti
  - Consuma capacità di calcolo e batteria in cicli inutili
  - Introduce una latenza tra il momento in cui il dato è disponibile e il momento in cui il secondo thread si sblocca



# Gestire il risveglio

```
bool ready;
std::mutex readyFlagMutex;

// cicla fino a che ready vale true
{
 std::unique_lock<std::mutex>
 ul(readyFlagMutex);

 while (!ready) {
 //rilascio il lock per permettere all'altro thread di
 //prenderlo
 ul.unlock();

 std::this_thread::sleep_for(std::chrono::milliseconds(100));

 ul.lock();
 }
 // uso la risorsa
} // rilascia il lock
```



# **std::condition\_variable**

- Modella una primitiva di sincronizzazione che permette l'attesa condizionata di uno o più thread
  - Fino a che non si verifica una notifica da parte di un altro thread, scade un timeout o si ferifica una notifica spuria
- Richiede l'uso di un **std::unique\_lock<Lockable>**
  - Per garantire l'assenza di corse critiche nel momento del risveglio



# **std::condition\_variable**

- Offre il metodo `wait(unique_lock)` per bloccare l'esecuzione del thread
  - Fino a quando non giunge una notifica
  - Senza consumare cicli di CPU
- Un altro thread può informare uno o tutti i thread attualmente in attesa che la condizione si è verificata
  - Attraverso i metodi `notify_one()` e `notify_all()`



# Implementazione

- Mantiene una lista di thread in attesa della condizione
  - Inizialmente la lista è vuota
  - Quando un thread esegue il metodo `wait(...)`, viene sospeso e aggiunto alla lista
  - Quando sono eseguiti i metodi `notify_one()` o `notify_all()`, uno o tutti i thread presenti nella lista sono risvegliati
  - Si basa sul S.O. per sospendere/risvegliare i thread



# La funzione di attesa

- È basata su un sistema a “doppia porta”
  - Attesa della segnalazione
  - Riacquisizione del Mutex
  - Solo quando entrambe sono state superate, il metodo ritorna
- 1. Aggiunge il thread corrente alla lista di quelli da risvegliare
- 2. Rilascia il lock
- 3. Sospende il thread
- 4. (attesa passiva)
- 5. Riacquisisce il lock
- Il metodo `notify_one()` sceglie un thread dalla lista di attesa e lo risveglia
  - `notify_all()` li risveglia tutti



# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ...
 lock_guard<mutex>
lg(m);
 dato= ...;
 cv.notify_one();
}
...
}

void consume() {
 unique_lock<mutex>
ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner

condition\_variable

wating

# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 {
 lock_guard<mutex>
lg(m);
 dato= ...;
 cv.notify_one();
 }
 ...
}

void consume() {
 unique_lock<mutex>
ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner

condition\_variable

wating

# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 lock_guard<mutex>
 lg(m);
 dato= ...;
 cv.notify_one();
}
...
}

void consume() {
 unique_lock<mutex>
 ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner



condition\_variable

wating



# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 lock_guard<mutex>
 lg(m);
 dato= ...;
 cv.notify_one();
}
...

void consume() {
 unique_lock<mutex>
 ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner

condition\_variable

wating

# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
{
 lock_guard<mutex>
lg(m);
 dato= ...;
 cv.notify_one();
}
...
}

void consume() {
unique_lock<mutex>
ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner



condition\_variable

wating



# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 {
 lock_guard<mutex>
lg(m);
 dato= ...;
 cv.notify_one();
 }
 ...
}

void consume() {
 unique_lock<mutex>
ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner

Il thread **rosso**  
attende di  
prendere il  
possesso del  
mutex

\_variable

# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 {
 lock_guard<mutex>
lg(m);
 dato=...
 cv.notify_one();
 }
 ...
}

void consume() {
 unique_lock<mutex>
l(m);
 cv.wait(w);
 //uso il dato
}
```

Mutex

owner



condition\_variable

wating



# Esempio

```
use namespace std;
mutex m;
condition_variable cv;
int dato;

void produce() {
 ... //calcola un dato
 {
 lock_guard<mutex>
lg(m);
 dato= ...;
 cv.notify_one();
 }
 ...
}

void consume() {
 unique_lock<mutex>
ul(m);
 cv.wait(ul);
 //uso il dato
}
```

Mutex

owner

condition\_variable

wating

# `std::condition_variable`

- La presenza di un unico lock fa sì che, se più thread ricevono la notifica, il risveglio sia progressivo
  - Non appena un thread rilascia il lock, un altro può acquisirlo e proseguire
- La relazione tra l'evento e la notifica è solo nella testa del programmatore
  - Per evitare notifiche spurie, si rende esplicito l'evento che si è verificato scrivendolo dentro una variabile condivisa (sotto il controllo del mutex)



# `std::condition_variable`

- Permette, ad uno o più thread, di attendere, senza consumare risorse, la ricezione di una notifica
  - Proveniente da un altro thread
- È possibile che il thread sia risvegliato per altri motivi
  - Problema delle cosiddette **notifiche spurie**
  - Occorre, al ritorno dal metodo `wait()`, controllare se la condizione attesa è verificata



# `std::condition_variable`

- Una versione overloaded del metodo `wait`, accetta come parametro un oggetto chiamabile
  - Alla ricezione di una notifica, il metodo `wait` invoca l'oggetto chiamabile
  - Ha il compito di valutare se l'evento è proprio quello atteso, restituendo `true` oppure `false`
  - Se il risultato è falso, si rimette in attesa; altrimenti ritorna al chiamante



# Esempio

```
std::mutex mut; //protegge la coda
std::queue<data_chunk> data_queue; //dato condiviso
std::condition_variable data_cond; //indica che la coda non è vuota

void data_preparation_thread() {
 while(more_data_to_prepare()) {
 data_chunk const data=prepare_data();
 std::lock_guard<std::mutex> lk(mut);
 data_queue.push(data);
 data_cond.notify_one();
 }
}

void data_processing_thread() {
 while(true) {
 std::unique_lock<std::mutex> lk(mut);
 data_cond.wait(lk, [](){return !data_queue.empty();});
 data_chunk data=data_queue.front();
 data_queue.pop();
 lk.unlock();
 process(data);
 if(is_last_chunk(data)) break;
 }
}
```

# **`wait_for(...)` e `wait_until(...)`**

- Limitano l'attesa nel tempo
  - Se chiamati senza indicare un oggetto chiamabile, restituiscono le costanti `std::cv_status::timeout` e `std::cv_status::no_timeout` per indicare l'esito dell'attesa
- Offrono l'opportunità al thread che esegue la notifica di completare la propria distruzione
  - Prima che i thread in attesa abbiano l'opportunità di osservare il dato condiviso

# Lazy evaluation

- Ci sono varie situazioni in cui è utile rimandare la creazione ed inizializzazione di strutture complesse fino a quando non c'è la certezza del loro utilizzo
- In un programma sequenziale, ci si riferisce tipicamente alla struttura in questione attraverso un puntatore inizializzato a NULL

# Lazy evaluation

- Quando occorre accedere alla struttura, si controlla se il puntatore abbia già un valore lecito
  - Nel caso in cui valga ancora NULL, si crea la struttura e se ne memorizza l'indirizzo all'interno del puntatore
- In un programma concorrente questa tecnica non può essere adottata direttamente
  - Il puntatore è una risorsa condivisa e il suo accesso deve essere protetto da un mutex



# Lazy evaluation

- Per supportare questo tipo di comportamento, C++11 offre la classe
  - `std::once_flag` e la funzione `std::call_once(...)`
- Costituisce la struttura di appoggio per la funzione `call_once`
  - Registra, in modo thread safe, se è già avvenuta o sia in corso una chiamata a `call_once`



# `std::call_once(flag,f,...)`

- Esegue la funzione f una sola volta
  - Se dal flag non risultano chiamate, inizia l'invocazione di f
  - Se un'altra chiamata è in corso, blocca l'esecuzione in attesa del suo risultato

# **std::call\_once(flag,f,...)**

```
#include <mutex>

class Singleton {
 static Singleton *instance;
static std::once_flag init;

 Singleton() {...} //privato

public:
 static Singleton *getInstance() {
 std::call_once(init, []() {
 instance=new Singleton();
 });
 return instance;
 }

 //altri metodi...
};

};
```



# Spunti di riflessione

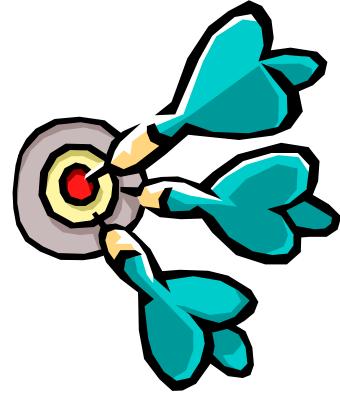
- Si realizzi la classe generica Buffer che mantiene una coda di oggetti di tipo T
  - Il metodo push(T) inserisce un nuovo elemento
  - Il metodo pop() restituisce l'elemento più vecchio e rimane bloccato finché il buffer è vuoto



# Interprocess communication

Programmazione di Sistema  
A.A. 2017-18

# Argomenti



- Concorrenza e processi
- Comunicazione tra processi

# Concorrenza e processi

- L'uso dei thread permette di sfruttare le risorse computazionali presenti in un elaboratore
  - La presenza di uno spazio di indirizzamento condiviso facilita la coordinazione e la comunicazione
- Ci sono situazioni in cui la presenza di un singolo spazio di indirizzamento non è possibile o desiderabile
  - Riuso di programmi esistenti
  - Scalabilità su più computer
  - Sicurezza



# Concorrenza e processi

- È possibile decomporre un sistema complesso in un insieme di processi collegati
  - Creandoli a partire da un processo genitore
  - Permettendo la cooperazione indipendentemente dalla loro genesi
- Ad ogni processo è associato almeno un thread (primary thread)
  - Un sistema multiprocesso è intrinsecamente concorrente
  - Solleva gli stessi problemi di interferenza e necessità di coordinamento



# Processi in Windows

- Costituiscono entità separate, senza relazioni di dipendenza esplicita tra loro
- La funzione CreateProcess(...)
  - Crea un nuovo spazio di indirizzamento
  - Lo inizializza con l'immagine di un eseguibile
  - Attiva il thread primario al suo interno
- Il processo figlio può condividere variabili d'ambiente ed handle a file, semafori, pipe ...
  - ... ma non può condividere handle a thread, processi, librerie dinamiche e regioni di memoria



# Processi in Linux

- Si crea un processo figlio con l'operazione fork()
  - Crea un nuovo spazio di indirizzamento «identico» a quello del processo genitore
  - I due processi condividono i riferimenti alle stesse pagine di memoria fisica
- Dopo l'esecuzione di fork(), tutte le pagine sono marcate con il flag «CopyOnWrite»
  - Eventuali scritture comportano la duplicazione della pagina e la separazione tra i due spazi di indirizzamento



# Creazione di processi

- La funzione exec\*() sostituisce l'attuale immagine di memoria dello spazio di indirizzamento
  - Ri-inizializzandola a quella descritta dall'eseguibile indicato come parametro



# Esempio

```
int main (const int argc, const char* const
argv[]) {
 pid_t ret = fork();
 switch (ret) {
 case -1:
 puts("parent: error: fork
failed!");break;
 case 0:
 puts("child: here (before execl)!");
 if (execl("./ch.exe", "./ch.exe",
0)==-1)
 perror("child: execl failed:");
 puts("child: here (after execl)!");
 //non si dovrebbe arrivare qui
 break;
 default:
 printf("par: child pid=%d \n",
ret);
 break;
 }
}
```

# Fork() e thread

- Nel caso di programmi concorrenti, l'esecuzione di fork() crea un problema
  - Il processo figlio conterrà un solo thread
  - Gli oggetti di sincronizzazione presenti nel padre possono trovarsi in stati incongruenti
- ```
int pthread_atfork(
    void (*prepare)(void),
    void (*parent)(void),
    void (*child)(void));
```

 - Registra un gruppo di funzioni che saranno chiamate in corrispondenza delle invocazioni a fork()



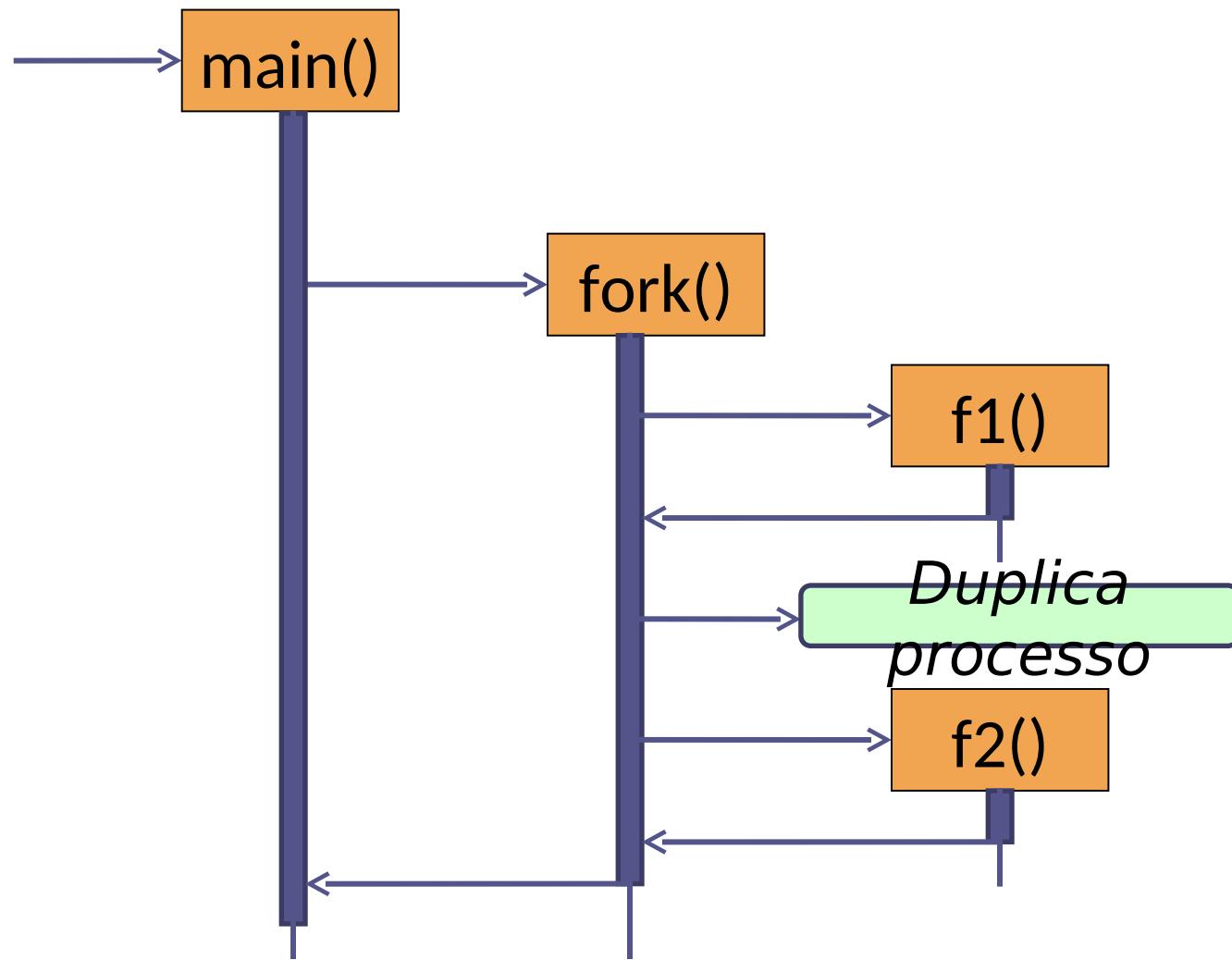
Esempio

```
void f1() { ... }
void f2() { ... }
void f3() { ... }

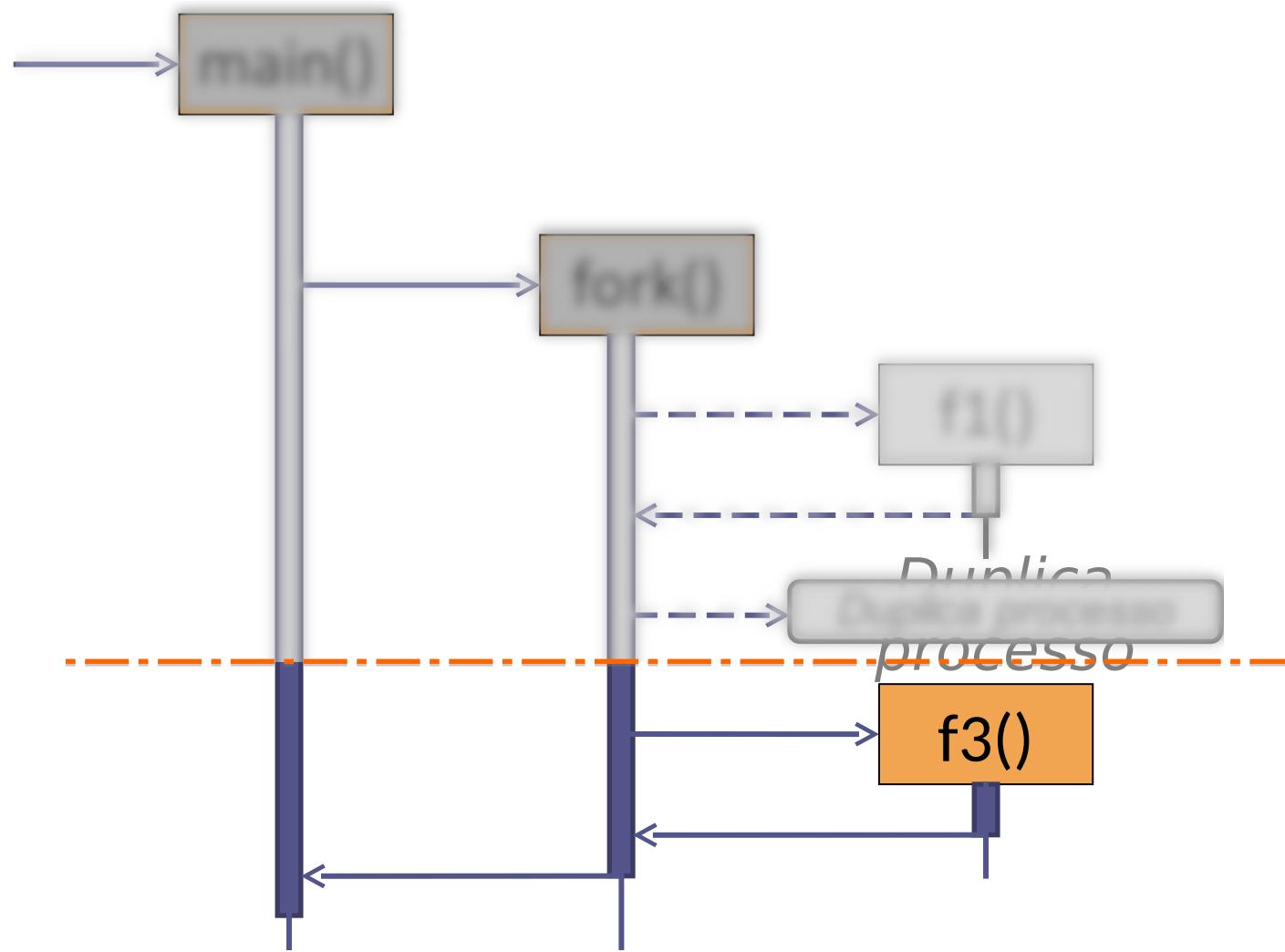
int main() {
    pthread_atfork(f1,f2,f3);
    //...
    int res= fork();
    if (res== -1 ) { /* errore */ }
    else if (res == 0) { /* child */ }
    else { /* parent */ }
}
```



Processo genitore



Processo figlio



IPC - InterProcess Communication

- Il S.O. impedisce il trasferimento diretto di dati tra processi
 - Ogni processo dispone di uno spazio di indirizzamento separato
 - Non è possibile sapere cosa sta capitando in un altro processo
- Ogni S.O. offre alcuni meccanismi per superare tale barriera in modo controllato
 - Permettendo lo scambio di dati...
 - ...e la sincronizzazione delle attività



Rappresentazione delle informazioni scambiate

- Indipendentemente dal tipo di meccanismo adottato, occorre adattare le informazioni scambiate
 - Così da renderle comprensibili al destinatario



Rappresentazione interna

- Internamente, un processo può usare una varietà di rappresentazioni
 - Tipi elementari (numerici, logici, caratteri, ...)
 - Tipi strutturati (record, array, classi, ...)
 - Puntatori per strutture dati complesse (alberi, grafi, ...)
- La rappresentazione interna non è adatta ad essere esportata
 - I puntatori non hanno senso al di fuori del proprio spazio di indirizzamento
 - Alcune informazioni (handle) non sono esportabili



Rappresentazione esterna

- Formato intermedio che permette la rappresentazione di strutture dati arbitrarie
 - Sostituendo i puntatori con riferimenti indipendenti dalla memoria
- Formati basati su testo
 - XML, JSON, CSV, ...
- Formati binari
 - XDR, HDF, ...



Serializzazione

- Le rappresentazioni esterne possono essere trattate come blocchi compatti di byte
 - Possono essere duplicati e trasferiti senza comprometterne il significato
- I dati vengono scambiati nel formato esterno
 - La sorgente esporta le proprie informazioni (marshalling)
 - Il destinatario ricostruisce una rappresentazione su cui può operare direttamente (unmarshalling)
- Le operazioni di marshalling e unmarshalling possono essere codificate esplicitamente
 - O essere eseguite da codice generato automaticamente dall'ambiente di sviluppo



Tipi di IPC

- Code di messaggi
- Pipe
- Memoria condivisa
- Altro
 - File, socket, segnali, ...

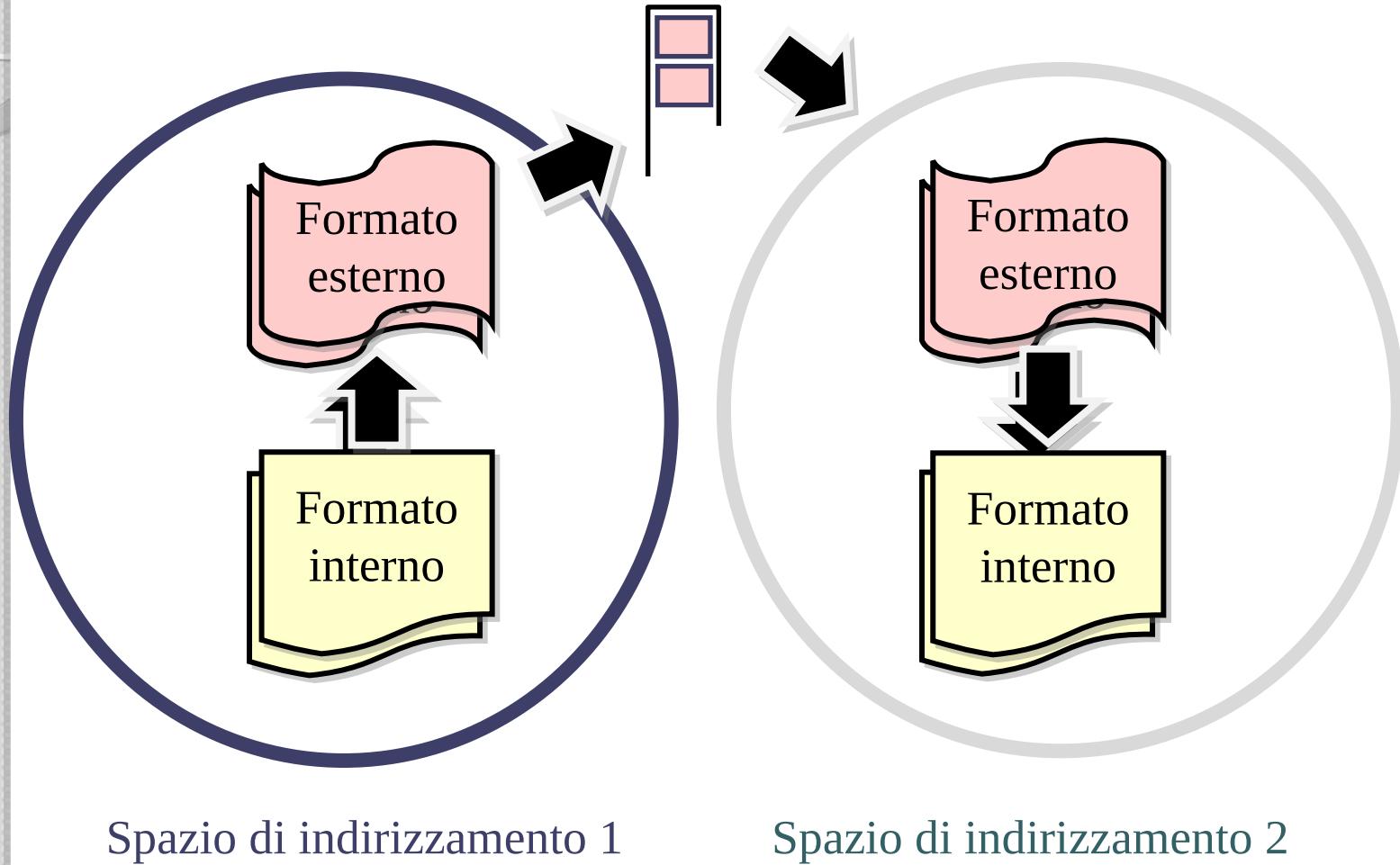


Code di messaggi

- Permettono il trasferimento atomico di blocchi di byte
 - Comportamento FIFO
 - Sono possibili più mittenti
 - L'inserimento di ciascun blocco sincronizza mittente e destinatario



Code di messaggi



Spazio di indirizzamento 1

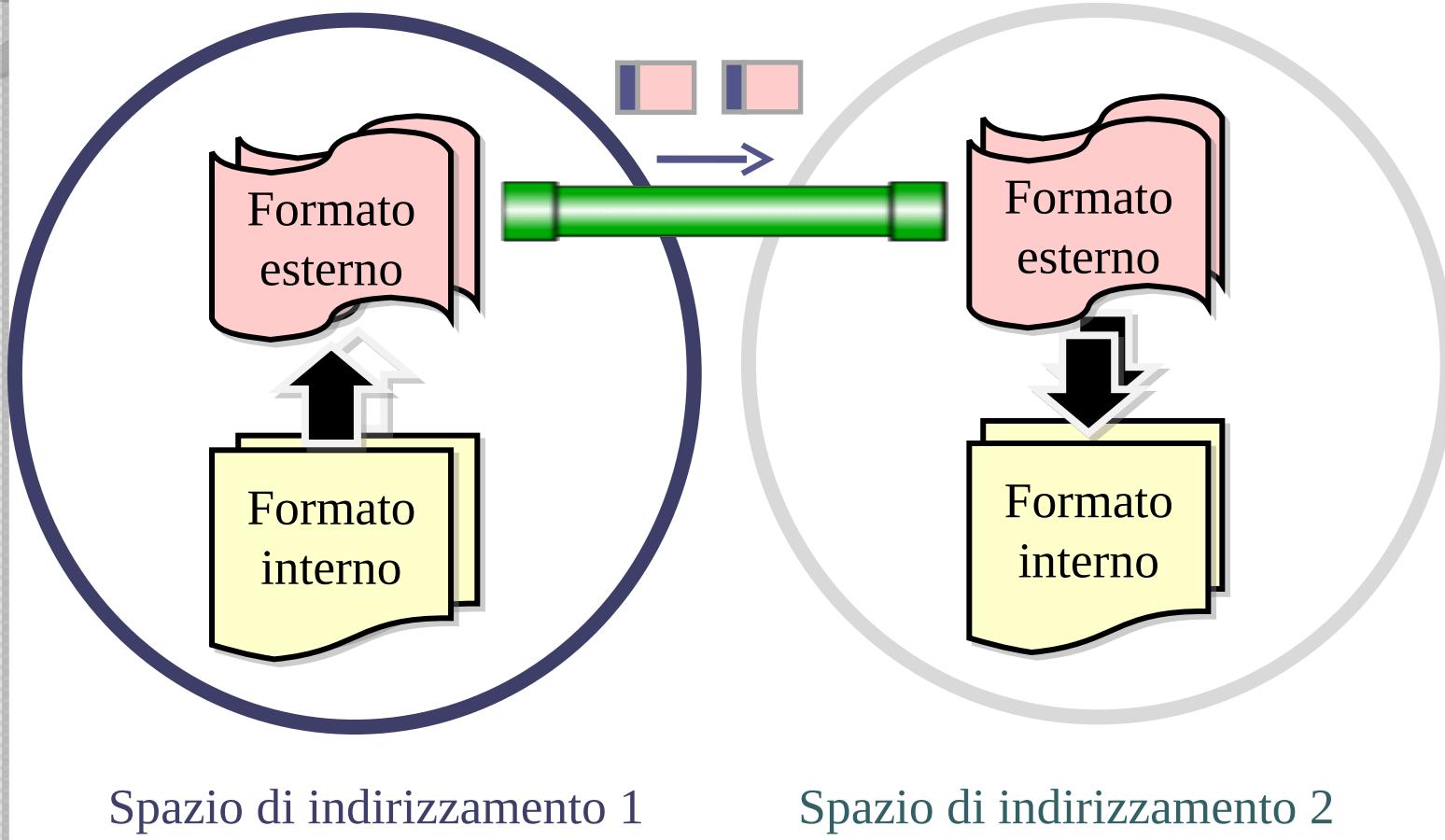
Spazio di indirizzamento 2

Pipe

- «Tubi» che permettono il trasferimento di sequenze di byte di dimensioni arbitrarie
 - Occorre inserire marcatori che consentano di delimitare i singoli messaggi
 - Comunicazione sincrona 1-1



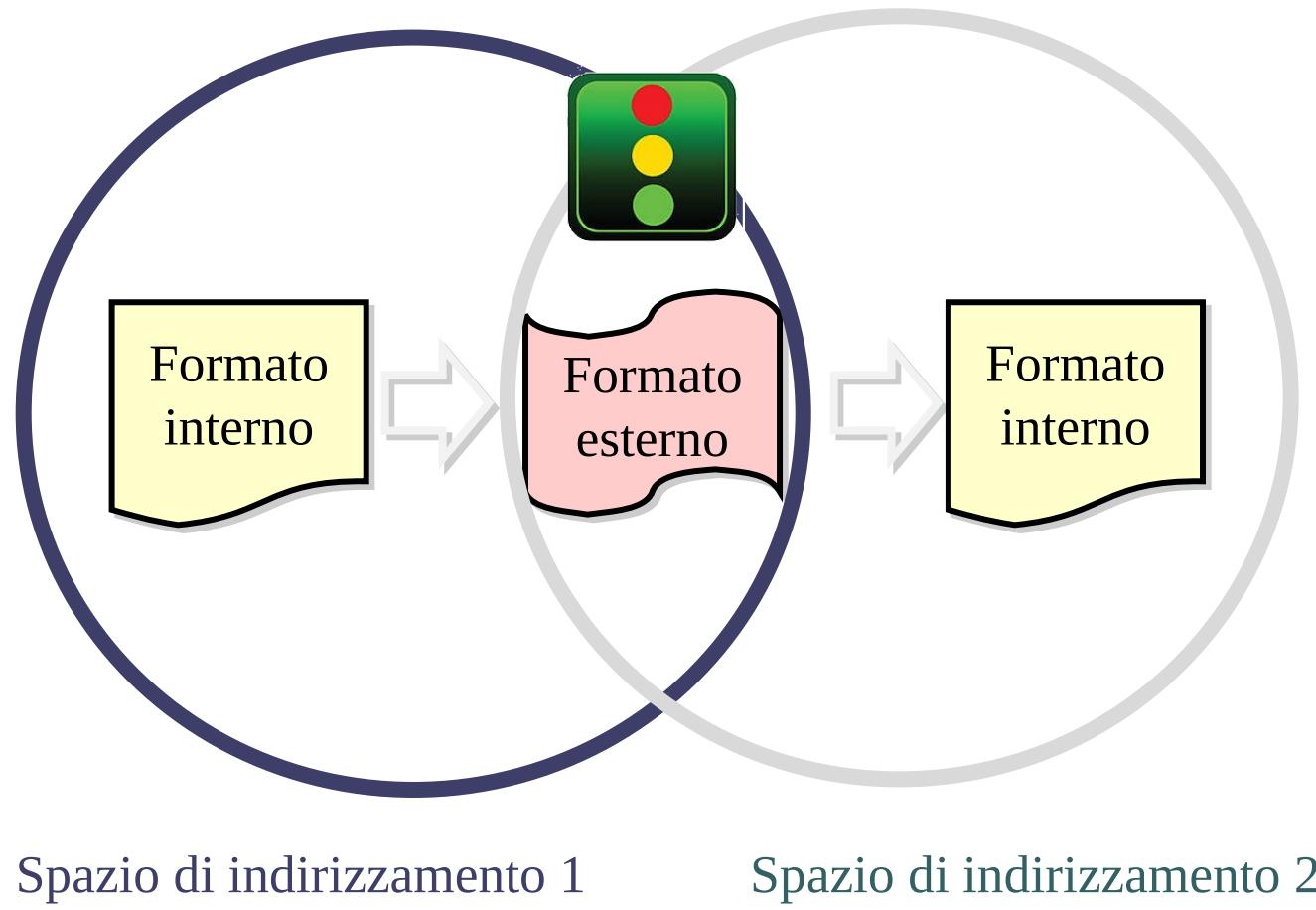
Pipe



Memoria condivisa

- Insieme di pagine fisiche mappate in due o più spazi di indirizzamento
 - Richiedono ulteriori meccanismi di sincronizzazione per evitare interferenze tra i thread dei processi coinvolti





Semafori

- Costrutti di sincronizzazione a basso livello
 - Basati sulla manipolazione atomica di un valore intero
 - Il valore è gestito dal S.O. e non può mai diventare negativo
- Due operazioni di base:
 - Up()
 - incrementa il valore
 - Down()
 - Se il valore è = 0, si blocca in attesa di un incremento
 - Decrementa il valore



Semafori e sincronizzazione

- Un semaforo inizializzato ad 1 può essere visto come un mutex
 - Si proteggono le sezioni critiche di codice racchiudendole tra le operazioni down() e up()



Identità dei canali

- I canali di comunicazione sono creati dal S.O. su richiesta dei singoli processi
 - Ad ogni canale viene associato un identificativo univoco a livello di sistema
- I processi che cooperano devono accordarsi sul tipo e sull'identità del canale usato per la comunicazione /sincronizzazione



Fattori che influenzano la scelta del meccanismo

- Relazione tra i processi
 - Dipendenza esplicita
 - Nessuna dipendenza
- Tipo di comunicazione richiesto
 - Mono-/bidirezionale
- Numero di processi coinvolti



Spunti di riflessione

- Si realizzi un programma Windows che legga una stringa da linea di comando e la utilizzi come nome di un eseguibile da lanciare in un processo separato

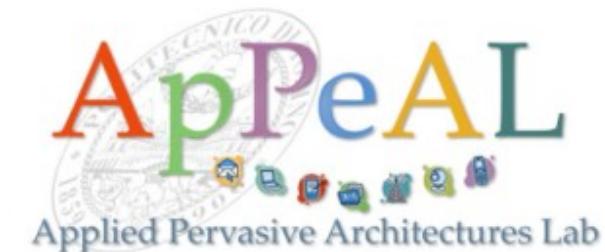




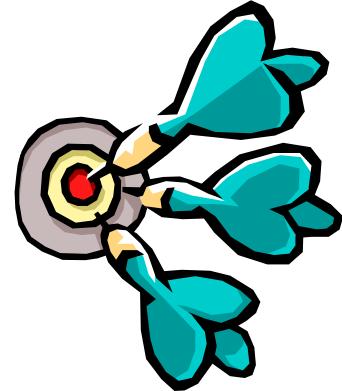
IPC in Windows

Programmazione di Sistema

A.A. 2017-18



Argomenti



- Sincronizzazione tra processi
 - Eventi
 - Semafori
 - Mutex
 - Meccanismi congiunti
- Comunicazione tra processi
 - MailSlot
 - Pipe
 - FileMapping
 - Altri meccanismi

Sincronizzazione e oggetti kernel

- La piattaforma win32 offre una ricca serie di meccanismi di sincronizzazione
 - Permettono di bloccare un thread fino a quando non si è verificato qualcosa in un altro thread
- Questi meccanismi si basano sull'uso di oggetti kernel condivisi
 - Il S.O. permette un accesso controllato al loro stato
 - Sono utilizzabili da thread appartenenti a processi differenti

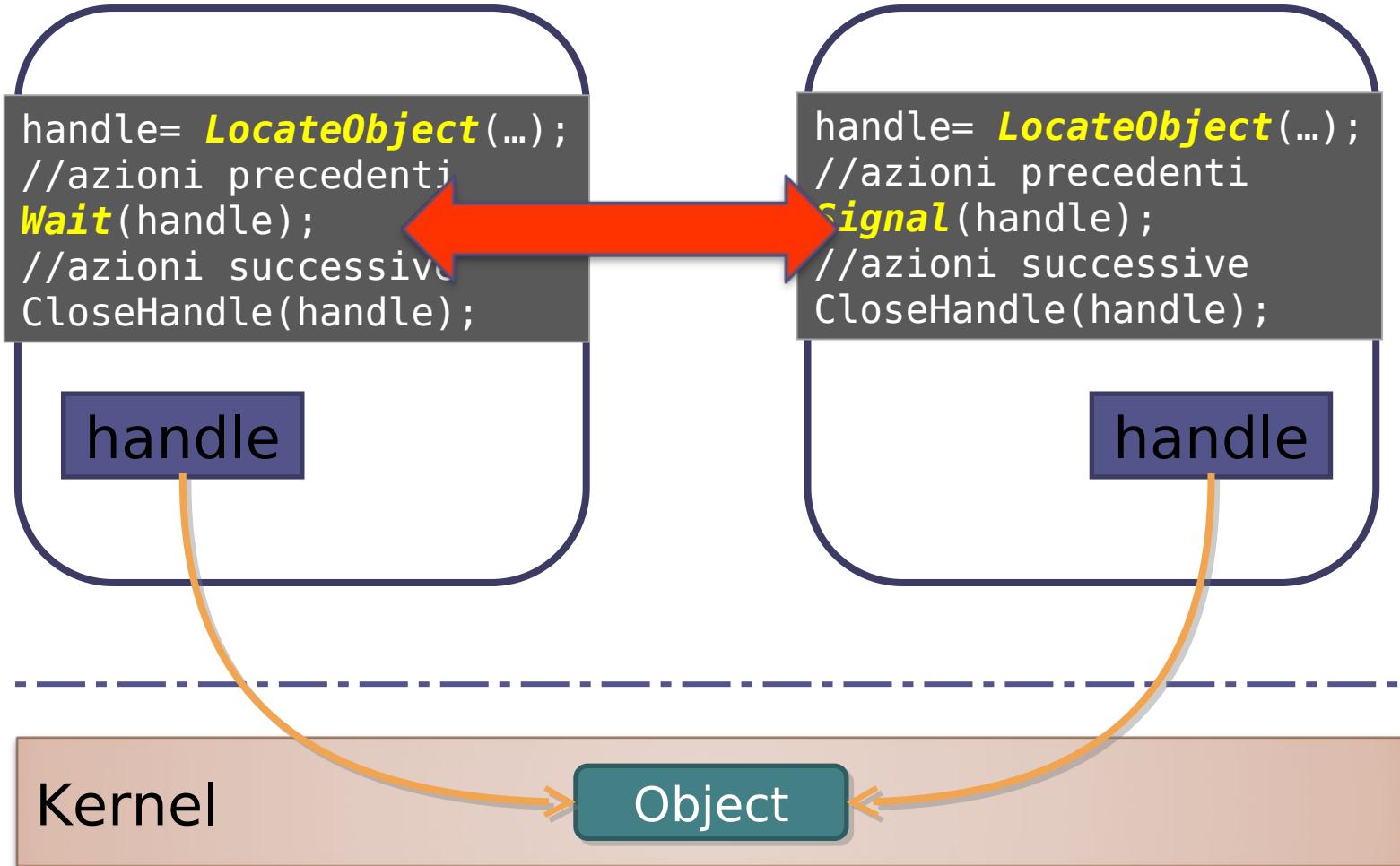


Sincronizzazione e oggetti kernel

- Rispetto ai costrutti di sincronizzazione usabili in un singolo processo, sono più generali ma meno efficienti
 - Tempi di sblocco maggiori, in quanto richiedono un passaggio alla modalità supervisore
- Tutte le tecniche si basano sull'uso delle funzioni di attesa offerte da Windows
 - `WaitForSingleObject(...)`
 - `WaitForMultipleObjects(...)`
- Ogni tipo di oggetto definisce le proprie politiche di attesa e risveglio
 - Permettendo la realizzazione di comportamenti diversi, adatti alle diverse esigenze



Meccanismo generale



Stato di Segnalazione

- La maggior parte degli oggetti kernel può trovarsi in due stati differenti
 - Segnalato
 - Non segnalato
- Nel caso di processi e thread
 - Lo stato non segnalato indica che l'elaborazione è ancora in corso
 - Una volta raggiunto lo stato segnalato, non è possibile tornare indietro
- Gli oggetti di sincronizzazione (eventi, semafori, mutex) possono alternare i due stati di segnalazione
 - In funzione delle proprie politiche e delle richieste eseguite nel programma



Eventi

- Oggetti kernel che modellano il verificarsi di una condizione
 - Segnalata esplicitamente dal programmatore tramite opportuni metodi
- Gli eventi di tipo manual-reset permettono ad un numero indefinito di thread in attesa di svegliarsi
 - Quelli auto-reset tornano automaticamente allo stato non segnalato se causano lo sblocco di un thread



Eventi

- Ci si collega ad un evento tramite
 - CreateEvent(...)
 - OpenEvent(...)
- Due processi possono sincronizzarsi condividendo un evento (tramite il nome)



Ciclo di Vita degli Eventi

- Si modifica lo stato di segnalazione di un evento attraverso le primitive
 - SetEvent(...)
 - ResetEvent(...)
 - PulseEvent(...)

	AUTO_RESET	MANUAL_RESET
<i>SetEvent</i>	Un solo thread tra quelli in attesa viene rilasciato. Se nessuno è in attesa, il prossimo che effettuerà un attesa sull'evento sarà rilasciato	Tutti i thread in attesa vengono rilasciati. L'evento resta segnalato fino ad una chiamata a ResetEvent
<i>PulseEvent</i>	Un solo thread tra quelli in attesa viene rilasciato. Se nessuno è in attesa, non capita nulla	Tutti i thread in attesa vengono rilasciati. L'evento viene posto nello stato non segnalato

Semafori

- Mantengono al proprio interno un contatore
 - Stato segnalato se il contatore è > 0
 - Stato non segnalato se il contatore è = 0
 - Non può valere mai meno di zero
- WaitForSingleObject decrementa il contatore se >0
 - Altrimenti blocca il thread in attesa che un altro incrementi il contatore
- Il contatore è incrementato quando un thread invoca ReleaseSemaphore(...)



Utilizzo dei semafori

- Usati tipicamente quando si hanno più copie di una risorsa disponibili
- CreateSemaphore(...)
 - Crea un semaforo
- OpenSemaphore(...)
 - Recupera un handle di un semaforo precedentemente creato
- WaitForSingleObject / WaitForMultipleObjects
 - Decrementa il contatore o si accoda in attesa che venga incrementato
- ReleaseSemaphore
 - Incrementa il contatore



Mutex

- Assicurano a più thread (anche di processi differenti) l'accesso in mutua esclusione ad una data risorsa
 - Può essere specificato un tempo massimo di attesa
- Conservano l'ID del thread che li ha acquisiti ed un contatore
 - Se ID = 0, la risorsa non è stata acquisita ed il mutex è nello stato segnalato
- Il contatore mantiene il numero di volte che il thread ha acquisito la risorsa
 - Se il thread che possiede il mutex termina, il mutex viene resettato



Ciclo di vita di un mutex

- CreateMutex, OpenMutex
 - Crea (od ottiene) l'handle ad un mutex
- ReleaseMutex
 - Verifica che il thread corrente sia il possessore
 - Decrementa il contatore
 - Se arriva a 0 segnala il mutex
- WaitForSingleObject / WaitForMultipleObjects
 - Acquisisce un mutex se è nello stato segnalato o se è già in possesso del thread, e ne incrementa il contatore
 - Altrimenti attende che il mutex diventi segnalato



Uso di più primitive

- Due o più oggetti kernel possono essere usati in modo congiunto per implementare particolari politiche di sincronizzazione
 - Nel caso di memoria condivisa, si può usare un Mutex per regolare l'accesso ed uno o più eventi per indicare il completamento di azioni





```
hE=CreateEvent("done"
,...);
hMut=CreateMutex("m"
);
hFM=CreateFileMapping(
...);
WaitForSingleObject(h
Mut);

ptr=MapViewOfFile(hFM,
...);
//write to shared
memory
SetEvent(hE);
UnmapViewOfFile(hFM);
ReleaseMutex(hMut);
CloseHandle(...);
```

Kernel

FileMapping

Mutex

Event

```
hE=CreateEvent("done"
,...);
hMut=CreateMutex("m"
);
hFM=CreateFileMapping(
...);
WaitForMultipleObjects(
    [hE, hMult] ,WAIT_ALL,
...);
ptr=MapViewOfFile(hFM,
...);
//read shared memory
UnmapViewOfFile(hFM);
ReleaseMutex(hMut);
CloseHandle(...);
```

Mailslot

- Coda di messaggi asincrona per la comunicazione tra processi
 - Il mittente può essere un processo della macchina stessa...
 - ...oppure di un elaboratore appartenente allo stesso dominio di rete
- Un processo può essere contemporaneamente sia un mailslot client che un mailslot server
 - Ciò permette di realizzare canali di comunicazione bidirezionali
- È possibile inviare messaggi broadcast a tutte le mailslot con lo stesso nome nello stesso dominio di rete



Mailslot server

- Si crea una mailslot
 - Associandole un nome univoco

```
HANDLE hSlot;
LPTSTR SlotName = TEXT("\\\\.\\mailslot\\ms1");

hSlot = CreateMailslot(SlotName,
    0, // no maximum message size
    MAILSLOT_WAIT_FOREVER, // no time-out
    (LPSECURITY_ATTRIBUTES)NULL); //  
default  
security //
```

Mailslot server

- I messaggi vengono letti come normali file
 - Con le API ReadFile(...) e ReadFileEx(...)
- Il messaggio viene conservato finché non viene letto
- GetMailslotInfo(...)
 - Restituisce il numero di messaggi accodati e la dimensione del primo da leggere



Mailslot server

```
DWORD cbMessage, cMessage, cbRead;  
LPTSTR lpszBuffer;  
  
BOOL fResult = GetMailslotInfo(hSlot,  
    (LPDWORD)NULL, &cbMessage,  
    &cMessage, (LPDWORD)NULL);  
  
if (fResult && cbMessage!= MAILSL0T_NO_MESSAGE) {  
  
    fResult = ReadFile(hSlot,lpszBuffer,cbMessage,  
        &cbRead,NULL);  
        //...  
}
```



Mailslot Client

- Accoda i messaggi ad una mailslot
 - CreateFile(...) apre la mailslot
 - WriteFile(...) e WriteFileEx(...) scrivono atomicamente un messaggio



Mailslot Client

```
LPTSTR Slot = TEXT("\\\\.\\mailslot\\ms1");
HANDLE hSlot = CreateFile(Slot, GENERIC_WRITE,
                           FILE_SHARE_READ,
                           (LPSECURITY_ATTRIBUTES)NULL,
                           OPEN_EXISTING,
                           FILE_ATTRIBUTE_NORMAL,
                           (HANDLE)NULL);
//...
LPTSTR lpszMessage = TEXT("Message one");

BOOL fResult = WriteFile(hSlot, lpszMessage,
                         (DWORD)(lstrlen(lpszMessage) +
1)*sizeof(TCHAR),
                         &cbWritten, (LPOVERLAPPED)NULL);
```



Rilascio delle risorse

- `CloseHandle(...)`
 - Chiude la mailslot e rilascia le relative risorse nel momento in cui tutti i suoi handle siano stati chiusi



Pipe

- Possono essere di tipo
 - Anonimo
 - Dotato di nome (named pipe)



Anonymous pipe

- Mezzo efficiente di comunicazione tra 2 processi “parenti”
 - Ridirezione dello standard input e/o dello standard output tra processo padre e figlio
 - Sono solo monodirezionali
- Un processo padre, dopo aver creato una pipe, può passare uno dei due handle al figlio

Anonymous Pipe

- Una handle può anche essere duplicata nel processo con cui si intende comunicare
 - Tramite `DuplicateHandle`
- La sua identità può anche essere passata attraverso un segmento di memoria condivisa
 - Facendoglielo ereditare all'atto della `CreateProcess`
- `ReadFile` e `WriteFile`
 - Servono a leggere e scrivere dalla pipe
 - Le pipe anonymous non supportano l'input-output asincrono



Named Pipe

- Permettono comunicazioni tra due processi qualsiasi
 - Che risiedono sia sulla stessa macchina che su macchine differenti
- Possono essere create bidirezionali
- Il nome di una pipe è così specificato
 - \\ServerName\pipe\PipeName
 - PipeName è case-insensitive e deve essere unico all'interno del S.O.
 - “\\.\\"Indica il calcolatore locale



Named Pipe

- Creazione/apertura
 - CreateNamedPipe(...), OpenFile(...), CallNamedPipe(...)
- Si opera sulle named pipe come sui file
 - ReadFile(...), WriteFile(...), ecc.



Named pipe server

```
TSTR lpszPipename = TEXT("\\\\.\\pipe\\  
mynamedpipe");  
  
hPipe = CreateNamedPipe( lpszPipename  
    PIPE_ACCESS_DUPLEX,  
    PIPE_TYPE_MESSAGE |  
    PIPE_READMODE_MESSAGE | PIPE_WAIT,  
    PIPE_UNLIMITED_INSTANCES,  
    BUFSIZE, BUFSIZE,  
    0, NULL);  
//...  
BOOL fSuccess = ReadFile(  
    hPipe,           // handle to pipe  
    pchRequest,     // buffer to receive data  
    BUFSIZE*sizeof(TCHAR), // size of buffer  
    &cbBytesRead, // number of bytes read  
    NULL);         // not overlapped I/O
```



Named pipe client

```
TSTR lpszPipename = TEXT("\\\\.\\pipe\\  
mynamedpipe");  
  
HANDLE hPipe = CreateFile(  
    lpszPipename, GENERIC_READ |  
GENERIC_WRITE,  
    0, NULL, OPEN_EXISTING, 0 NULL);  
//...  
LPTSTR lpvMessage=TEXT("Default message from  
client.");  
BOOL fSuccess = WriteFile(  
    hPipe, // pipe handle  
    lpvMessage, // message  
    cbToWrite, // message length  
    &cbWritten, // bytes written  
    NULL); // not overlapped
```



Pipe

- Modalità di lettura
 - All'atto della creazione si può specificare la modalità di lettura
 - Stream di byte o a messaggio
- Modalità di attesa
 - Determina il comportamento delle operazioni di lettura, scrittura e di connessione
 - ReadFile, WriteFile, ConnectNamedPipe
 - In modalità bloccante
 - La funzione attende per un tempo indefinito che il processo con cui si vuole comunicare termini le operazioni sulla pipe
 - In modalità non bloccante
 - La funzione ritorna immediatamente nelle situazioni che richiederebbero un'attesa indefinita



File Mapping

- Meccanismo adatto per condividere ampie zone di memoria e a realizzare aree condivise persistenti
- Permette di accedere al contenuto di un file come se fosse un blocco di memoria
 - Allocato nello spazio di indirizzamento del processo
 - Scritture in tale blocco comportano la modifica del file
- Occorre sincronizzare accessi concorrenti allo stesso file-mapping
- È un meccanismo efficiente per condividere dati tra più processi sullo stesso computer



Ciclo di vita di un File Mapping

- **CreateFileMapping(...)**
 - Crea un file mapping
 - Se esiste già un file mapping con il nome passato, ne ritorna l'handle
 - Consente di specificare i diritti di accesso al file
- **MapViewOfFile**
 - Crea una vista del file mapping nello spazio di indirizzamento del processo
 - Ritorna un puntatore che può essere direttamente usato per accedere alla area di memoria condivisa
- **UnmapViewOfFile**
 - Rilascia la vista sul file mapping
- **CloseHandle**
 - Chiude la handle al file mapping



Altri meccanismi

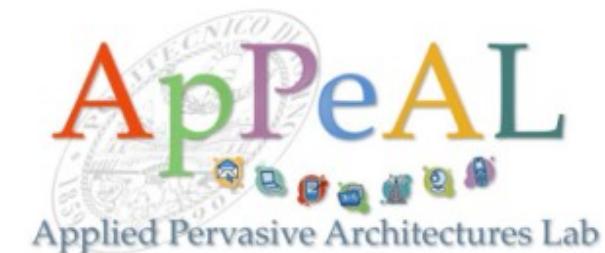
- Socket
 - Permettono la comunicazione tra macchine dotate di sistemi operativi differenti
 - Supportano direttamente solo il trasferimento di array di byte
- RPC – Remote Procedure Call
 - Tecnica basata sui socket
 - Fornisce un meccanismo di alto livello per il trasferimento di dati strutturati



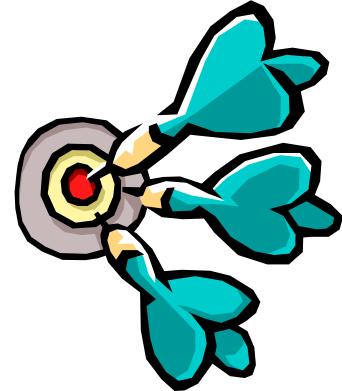


IPC - Linux

Programmazione di Sistema
A.A. 2016-17



Argomenti



- Identificativi
- Message Queue
- Pipe
- Shared Memory
- Memory Mapped File
- Semafori

Identificativi

- Ciascuna struttura IPC è identificata nel S.O. da un intero non negativo
 - All'atto della creazione di un oggetto IPC, si fornisce una chiave
 - Il S.O. converte questa chiave nell'ID associato
- Un processo può creare una nuova struttura IPC con chiave diversa da 0
 - Tutti i processi che conoscono la chiave possono ottenere l'ID corrispondente



Condivisione della chiave

- Se, all'atto della creazione, come chiave viene indicato IPC_PRIVATE (0)
 - Non sarà possibile ad altri ottenere la corrispondenza con l'ID
 - L'ID generato dovrà essere condiviso in altri modi
- Un processo può creare una nuova struttura IPC specificando come chiave un valore predefinito
 - Gli altri processi devono conoscere questa chiave
 - Uso di file header



Message Queues

- I processi che intendono comunicare si accordano
 - Sul pathname di un file esistente
 - E su un project-ID (0-255)
- Tramite ftok() convertono questi valori in una chiave univoca
- L'ultimo processo che fa accesso ad una struttura dati di IPC deve occuparsi della rimozione della stessa
 - Altrimenti continua ad essere un oggetto kernel valido...



Message Queues

- Permettono lo scambio di messaggi tra processi
- I messaggi sono composti da un tipo e da un payload
 - Il processo che riceve può specificare il tipo dei messaggi a cui è interessato
- I messaggi sono puntatori a strutture

```
struct message {  
    long type ;  
    char messagetext  
[ MESSAGESIZE ];  
};
```



Creazione/accesso

```
int msgget(key_t key, int msgflg)
```

- Crea una nuova message queue
- Ottiene l'id della message queue associata alla chiave specificata

Invio di un messaggio

```
int msgsnd(int msqid,  
          const void *msgp,  
          size_t msgsz,  
          int msgflg)
```

- msqid: id della coda
- msgp: puntatore al messaggio
- Il mittente deve avere il permesso di scrittura sulla coda



Lettura di un messaggio

```
ssize_t msgrcv(int msqid, void *msgp,  
                size_t msgsz,  
                long msgtyp, int  
                msgflg)
```

Copia nel buffer msgp un messaggio della coda identificata da msqid

- msgsz indica la dimensione del corpo del messaggio
- msgtyp indica il messaggio di interesse
 - 0 viene restituito il primo messaggio della coda
 - >0 viene restituito il primo messaggio del tipo indicato
 - <0 il messaggio con il più basso valore del campo tipo



Controllare una coda

```
int msgctl(int msqid, int cmd,  
           struct msqid_ds *buf)
```

- Esegue l'operazione di controllo specificata da cmd sulla coda msqid
- Con comando IPC_RMID rilascia le risorse

Pipe

- Permettono la IPC tra processi padre-figlio
 - Creati con la funzione pipe()

```
int pipe_fds[2];
int read_fd, write_fd;
pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Esempio

```
int fds[2];
pid_t pid;
pipe (fds);
pid = fork ();
if (pid == (pid_t) 0) {
    /* processo figlio */
    close (fds[1]);
    // ... Lettura dalla pipe
    close (fds[0]);
} else { /* processo padre */
    close (fds[0]);
    // ... Scrittura sulla pipe
    close (fds[1]);
}
```

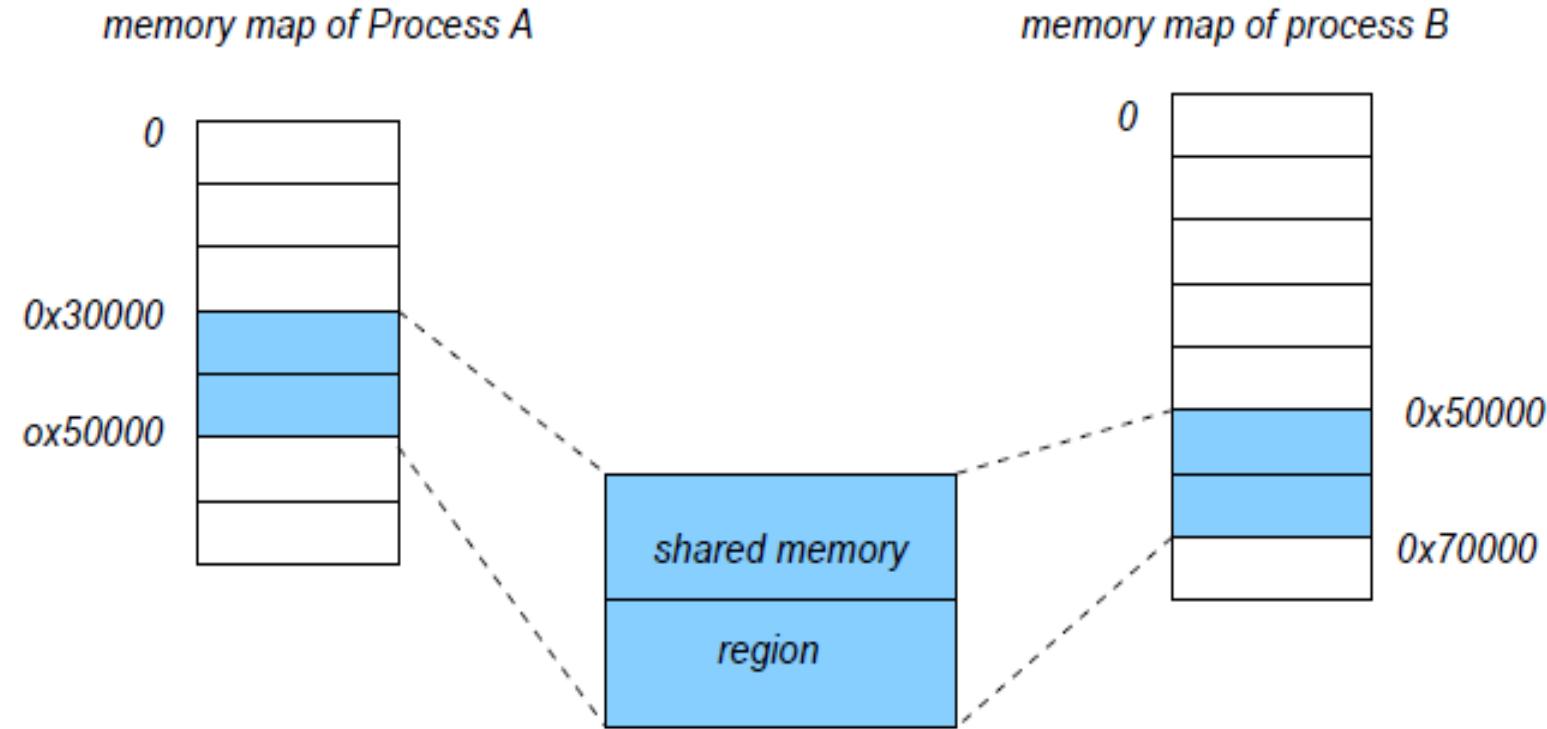


Named pipe: FIFO

- Permette la comunicazione tra processi generici
- Create con la funzione `int mkfifo(const char *path, mode_t mode);`
- Si accede come ad un file «normale»



Shared Memory



Creazione/accesso

```
int shmget(key_t key,  
           size_t size,  
           int shmflg)
```

- Restituisce l'id del segmento condiviso associato alla chiave key
- Eventualmente alloca il segmento in base ai parametri shmflag
- La dimensione restituita da shmget è uguale alla dimensione effettiva del segmento, arrotondata ad un multiplo di PAGE_SIZE



Operazioni di controllo

```
int shmctl(int shmid, int cmd,  
           struct shmid_ds *buf)
```

- Ottenere informazioni sul segmento
- Impostare i permessi, il proprietario e il gruppo
- Rilasciare le risorse associate

Possibili operazioni

- **IPC_STAT**

- Copia le informazioni dalla struttura dati del kernel associata alla memoria condivisa all'interno della struttura puntata da buf

- **IPC_SET**

- Scrive i valori contenuti nella struttura dati puntata da buf nell'oggetto kernel corrispondente alla memoria condivisa specificata

- **IPC_RMID**

- Marca il segmento come da rimuovere. Il segmento viene rimosso dopo che dell'ultimo processo ha effettuato il detach



Connessione e

```
void *shmat(int shmid,  
            const void *shmaddr,  
            int shmflg)
```

- Connnette il segmento identificato da shmid allo spazio di indirizzamento del processo chiamante

```
int shmdt(const void *shmaddr)
```

- Disconnnette il segmento specificato dallo spazio di indirizzamento del chiamante



Memory mapped file

- Porzioni di file mappate in memoria
 - Per condividere il contenuto del file tra processi in lettura/scrittura
 - Semplificano operazioni tipo fseek()

```
void* mmap ( void* start, size_t n,  
            int prot, int flags,  
            int fd, off_t off);
```

- Mappa n byte
- Del file specificato da fd
- A partire dall'offset off
- Preferibilmente all'indirizzo start

Rilascio

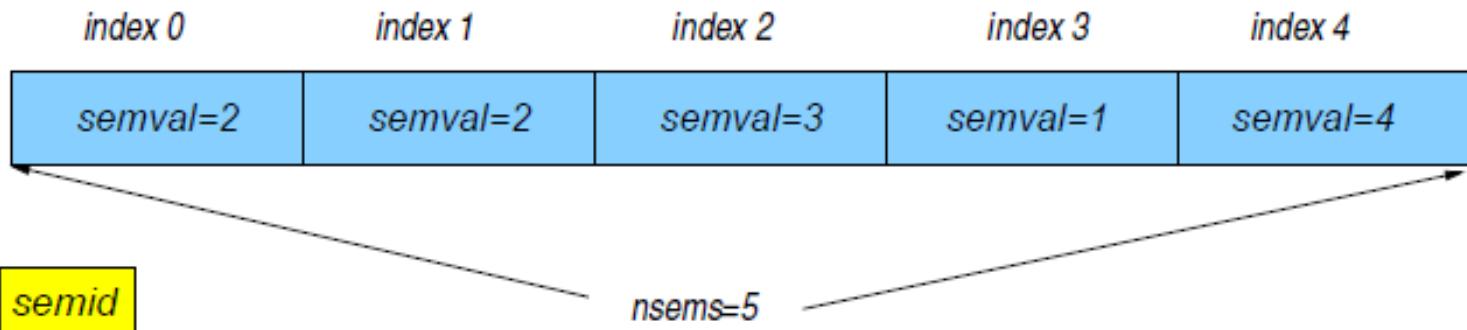
```
int munmap ( void* start,  
              size_t length );
```

- Rilascia il mapping specificato dai parametri
- Invalida gli indirizzi corrispondenti
- La regione viene rilasciata automaticamente quando il processo termina
 - La chiusura del file descriptor non causa il rilascio della regione



Semafori System-V

- I semafori System-V vengono utilizzati per la sincronizzazione di thread di processi differenti
 - Contrariamente all'utilizzo dei semafori posix
- Sono costituiti da un array di contatori
 - Se è necessario proteggere più risorse si



Creazione/accesso

- int semget(key_t key, int nsems, int semflg)
 - Restituisce l'id dell'insieme di semafori associati alla chiave

```
int sid = semget( mySemKey, 1,  
                  IPC_CREAT|0700  
)
```

Operare su un semaforo

- `int semop(int semid, struct sembuf *sops, unsigned n)`
 - `semid`: id del semaforo
 - `sops`: operazioni da compiere
 - `n`: numero elementi del semaforo



Struttura sembuf

```
struct sembuf {  
    unsigned short sem_num ;  
    //indice del semaforo su cui  
    operare  
  
    short sem_op ;  
    // operazione da effettuare  
    // >0 per acquisire  
    // <0 per rilasciare  
  
    sem_flg ;  
    // solitamente 0  
};
```



Esempio

```
struct sembuf  sem_lock;  
  
sem_lock.sem_num = 0;  
sem_lock.sem_op  = -1;  
sem_lock.sem_flg  = 0;  
  
if (semop(sid, &sem_lock, 1) == -1) {  
    perror("semop ");  
    exit(-1);  
}
```

Operazioni di controllo

- `int semctl(int semid, int i, int cmd, [union semun arg])`
 - Effettua l'operazione cmd sull'i-esimo semaforo del set identificato da semid



Union semun

```
union semun {  
    int val ;  
    /* Value for SETVAL */  
    struct semid_ds * buf ;  
    /* Buffer for IPC_STAT , IPC_SET */  
    unsigned short * array ;  
    /* Array for GETALL , SETALL */  
    struct seminfo * buf ;  
    /* Buffer for IPC_INFO */  
};
```

Struttura di semid_ds

```
struct semid_ds {  
    struct ipc_perm sem_perm ;  
    /* Ownership and permissions */  
    time_t sem_otime ;  
    /* Last semop time */  
    time_t sem_ctime ;  
    /* Last change time */  
    unsigned short sem_nsems ;  
    /* No . of semaphores in set */  
};
```

Possibili operazioni

- **IPC_STAT**
 - Copia le informazioni dalla struttura dati del kernel associata al semaforo all'interno della struttura `semid_ds` puntata da `arg.buf`
- **IPC_SET**
 - Scrive i valori contenuti nella struttura dati `semid_ds` puntata da `arg.buf` nell'oggetto kernel corrispondente all'insieme di semafori
 - Aggiorna `sem_ctime`
- **IPC_SETALL**
 - Imposta `semval` per tutti i semafori del set utilizzando `arg.array`
 - Aggiorna `sem_ctime`
- **IPC_GETALL**
 - Restituisce i `semval` correnti di tutti i semafori

Rimozione di un semaforo

- Per rilasciare le risorse relative ad un semaforo si utilizza la funzione semctl con parametro IPC_RMID

```
semctl( sid, 0, IPC_RMID, 0 );
```

Spunti di riflessione

- Si confrontino i meccanismi di comunicazione tra processi offerti da Linux con quelli di Windows e si provi a compilare una scaletta riassuntiva che mostri similitudini e differenze





Introduzione al linguaggio C#

Anno Accademico 2017-18

Obiettivi

- Conoscere le caratteristiche base del linguaggio
 - Metodi, proprietà, eventi, attributi
 - Pattern di programmazione
- Comprendere l'utilizzo delle principali classi legate all'ambiente .NET
 - Oggetti, stringhe, file, interfacce grafiche
- Sapere realizzare semplici applicazioni dotate di interfaccia grafica



Un linguaggio “a componenti”...

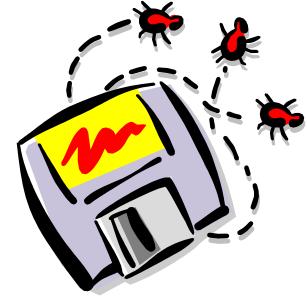


- Supporto di astrazioni di livello elevato
 - Metodi, eventi, proprietà, campi
 - Attributi (metadati)
- Permette la realizzazione di pacchetti entro-contenuti ed autodescrittivi (assembly)
 - Non richiede l'uso di linguaggi ad hoc per la descrizione delle interfacce
 - Supporta la documentazione integrata con il codice

... con accesso uniforme ai dati...

- In Java poca uniformità tra i dati
 - I tipi elementari (int, byte, char, boolean,...) non sono oggetti
 - Esistono classi “wrapper” (Integer, Byte, ...) con accesso in sola lettura:
 - Necessari per gestire collezioni (liste, vettori, ...) di dati elementari
- In C#, le differenze sono meno nette
 - La conversione da valore elementare ad oggetto è trasparente (boxing, anche in Java dalla versione 1.5)
 - La conversione inversa (unboxing) è esplicita (richiede uso di casting, idem)
 - Le istanze delle classi wrapper (Int32, Byte,...) possono essere modificate
 - Le collezioni di dati sono più semplici da gestire

... robusto...

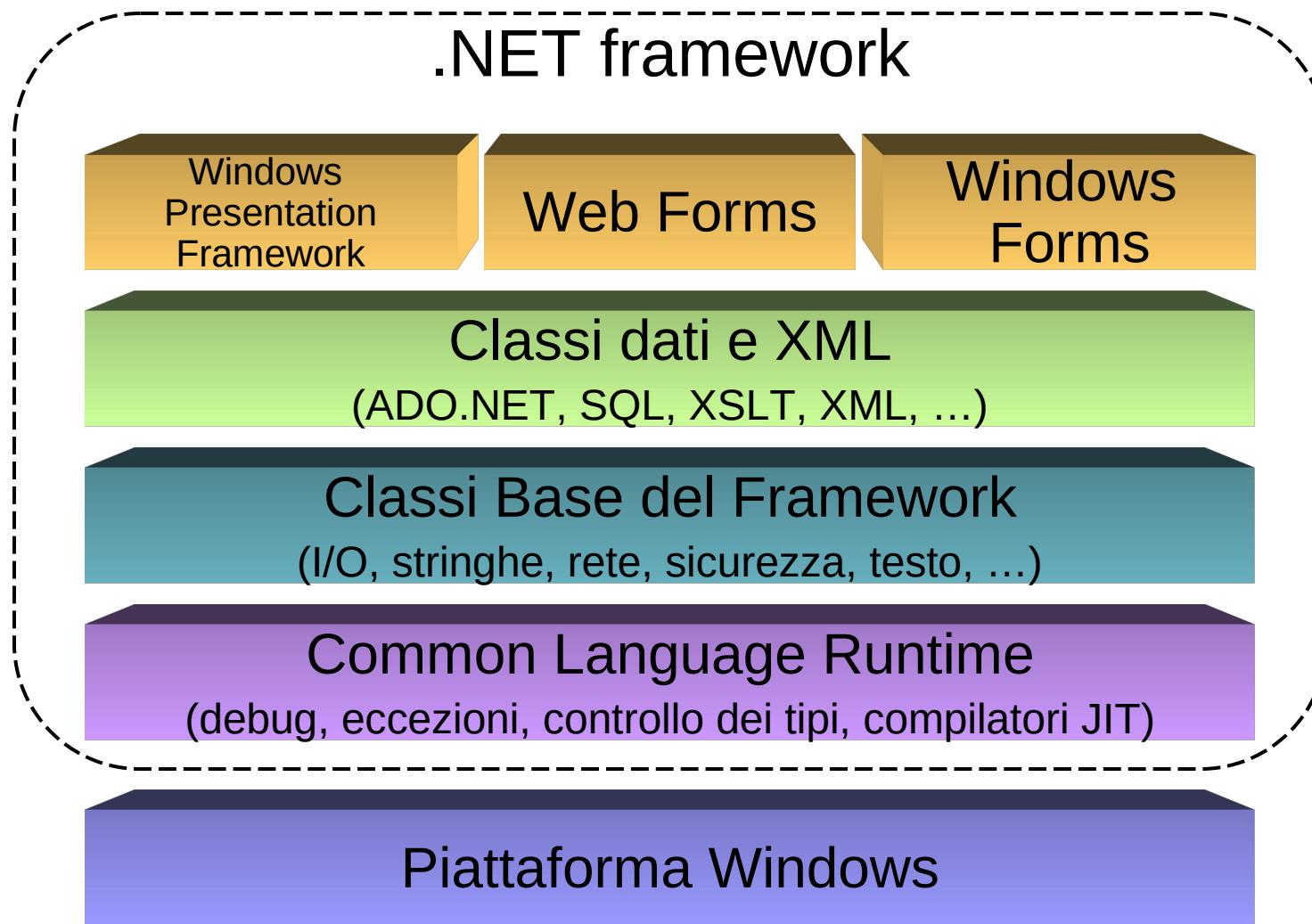


- Gestione automatica dell'allocazione di memoria
 - Non è una novità per i programmatori Java!
- Il compilatore verifica la corretta inizializzazione delle variabili
 - Idem
- Il concetto di eccezione è cablato nel linguaggio
 - Ma non è obbligatorio né dichiararne né gestire eventuali malfunzionamenti
- Ogni modulo binario ha esplicitamente una versione
 - È compito del programmatore gestire eventuali conflitti

...compatibile con gli investimenti precedenti

- Sintassi simile al C/C++/Java
 - Concetti simili, nomi differenti
 - Supporto della modalità non gestita: accesso ai puntatori
- Alto livello di interoperabilità
 - Con gli altri linguaggi .NET
 - Con altri standard (XML, COM, ...)
- Curva di apprendimento rapida
 - .NET è costituito da milioni di righe di codice C#

Architettura .NET



Alla base di C#: Common Language Runtime

(1)

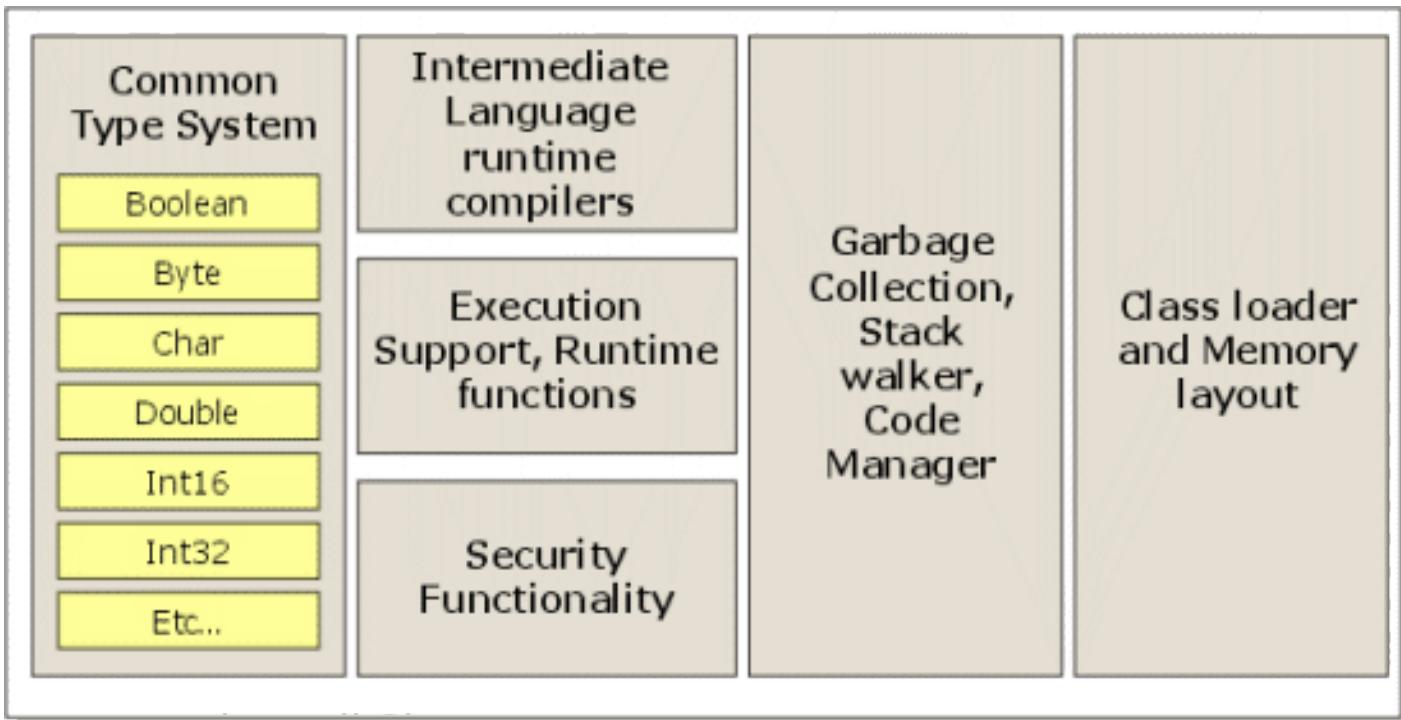
Strato software che si interpone tra il sistema operativo e le applicazioni .NET

- La fase di compilazione genera dei moduli espressi in un linguaggio intermedio comune (CIL)
- Il CLR contiene un modulo che traduce il codice intermedio nel linguaggio macchina del processore che ospita l'applicazione, permettendo così l'esecuzione
- Esiste un sistema comune di tipi e di API utilizzabili dai linguaggi supportati dal framework

Alla base di C#: Common Language Runtime

(2)

- Fornisce anche ulteriori strumenti per lo sviluppo, riducendo i problemi di installazione e compatibilità



Managed Code

- Il codice utilizzato per la costruzione di applicazioni .NET è detto “managed”
 - Gestione dell'esecuzione delle singole istruzioni virtuali
 - Gestione automatica del ciclo di vita degli oggetti e della relativa distruzione da parte del garbage collector
 - Gestione strutturata delle eccezioni

- Common Intermediate Language
 - Set di istruzioni di un elaboratore virtuale in cui vengono compilati tutti i sorgenti dei linguaggi .NET
 - Elimina le dipendenze dalla CPU
 - Equivalente del bytecode di Java
- Contiene le istruzioni per il caricamento, la memorizzazione, l'inizializzazione delle classi e le chiamate ai metodi
- Combinato con i metadati e il sistema di tipi comune, permette l'integrazione moduli scritti in linguaggi differenti
- Per poter eseguire una applicazione, il codice intermedio corrispondente è convertito in codice macchina (compilazione just in time)

C# in sintesi

- Linguaggio ad oggetti fortemente tipato...
 - L'unità minima di programmazione è la classe
- ...ad ereditarietà semplice...
 - Tutte le classi derivano da "System.Object"
- ...con supporto della riflessione...
 - È possibile esaminare ogni oggetto, scoprirne le caratteristiche, costruire nuove classi in fase di esecuzione
- ...che incorpora nella propria sintassi i principali pattern di programmazione
 - Eventi, eccezioni, iterazione, gestione della memoria e delle risorse, proprietà, metadati, ...

Un esempio

```
using System;

class Hello {
    public static void Main() {
        Console.WriteLine("Hello World!");
    }
}
```

Hello.cs

C:\>csc Hello.cs
C:\>hello
Hello World!

Struttura di esecuzione

hello.exe (assembly)

```
_EntryStub:                                PE Header
    JMP [mscoree.dll!_CorExeMain]

.method static void Main(string[] args) {      // IL
    ldstr "Hello, World"
    call void [mscorlib]System.Console.WriteLine(string)
}
```

mscoree.dll (DLL Win32)

```
_CorExeMain:
    Si seleziona la VM in base alla configurazione
    rtLib = LoadLibrary("mscorwks.dll" o "mscorsvr.dll")
    pCorExeMain = GetProcAddress(rtLib, "_CorExeMain")
    JMP [pCorExeMain]
```

mscorwks.dll o mscorsvr.dll (DLL Win32)

```
_CorExeMain:
    Inizializza l'ambiente di esecuzione
    Compilazione JIT del metodo Main in un buffer
    JMP [buffer]
```

Modello di esecuzione

- Il linguaggio intermedio (IL) viene generato appoggiandosi ad un motore di esecuzione basato su una macchina con stack "infinito"
 - Può essere interpretato simulando tale astrazione o essere compilato *just-in-time* in codice eseguibile di uno specifico processore
- Lo stack contiene lo spazio di valutazione di tutte le espressioni temporanee
 - Le istruzioni IL tipicamente presuppongono che sullo stack sia presenti i propri parametri e depositano qui i loro risultati



Modello di esecuzione

```
static int Add(int a, int b)
{
    return a + b;
}
```

CSC

```
.method static int32 Add(
    int32 a, int32 b)
{
    ldarg.0
    ldarg.1
    add
    ret
}
```

```
7ff8`1c8100d0  mov  dword ptr [rsp+10h],edx
7ff8`1c8100d4  mov  dword ptr [rsp+8],ecx
7ff8`1c8100d8  mov  ecx,dword ptr [rsp+10h]
7ff8`1c8100dc  mov  eax,dword ptr [rsp+8]
7ff8`1c8100e0  add  eax,ecx
7ff8`1c8100e2  jmp  00007ff8`1c8100e4
7ff8`1c8100e4  nop
7ff8`1c8100e5  ret
```

JIT

```
.method static int32 Add(
    int32 a, int32 b)
{
    ldarg.0
    ldarg.1
    add
    ret
}
```

Modello di esecuzione

- La compilazione just in time può essere sostituita dal processo NGEN
 - Native image GENeration
 - Compilazione Ahead Of Time, tipicamente eseguita in fase di installazione del codice su una data piattaforma
- I singoli moduli eseguibili (Assembly) vengono trasformati in immagini native per la piattaforma corrente
 - Memorizzate nelle cartelle
 %windir%\assembly\
 NativeImages_v4.0.30319_32



NGEN

```
static void Main() {  
    Console.WriteLine(Add(1, 2));  
}  
  
static int Add(int a, int b) {  
    return a + b;  
}
```



```
0:000> !U 00007FFD85E70090  
Normal JIT generated code  
Arith.Main()  
Begin 00007ffd85e70090, size 14  
00007ffd`85e70090  sub     rsp,28h  
00007ffd`85e70094  mov     ecx,3  
00007ffd`85e70099  call    mscorelib_ni+0xd24780 (00007ffd`e4b44780)  
                      (System.Console.WriteLine(Int32), mdToken: 000000000600099d)  
00007ffd`85e7009e  nop  
00007ffd`85e7009f  add     rsp,28h  
00007ffd`85e700a3  ret
```

$$1 + 2 = 3$$

Confronto

	Win32/COM	.NET
Accesso alla piattaforma	API, oggetti COM	Librerie di classi
Formato del codice	X86	IL
Supporto ai tipi di dati	ad-hoc	Ad oggetti
Gestione della memoria	ad-hoc, conteggio dei rif.	Controllata da un GC
Configurazione	Registry	File XML
Gestione delle versioni	ad-hoc	integrata
Origine del codice	Disco	Disco, rete
Unità di distribuzione	DLL, EXE	Assembly
Unità di esecuzione	Processo	AppDomain
Sicurezza	Basata sui ruoli	Basata sui ruoli e sull'evidenza
Accesso alla rete	Socket, DCOM	Socket, remoting, servizi web, WCF
Gestione degli errori	HRESULT, eccezioni, GetLastError()	Modello di eccezioni unificato

Sintassi del linguaggio: commenti XML

```
class Element
{
    /// <summary>
    /// Returns the attribute with the given name and
    /// namespace</summary>
    /// <param name="name">
    ///   The name of the attribute</param>
    /// <param name="ns">
    ///   The namespace of the attribute, or null if
    ///   the attribute has no namespace</param>
    /// <return>
    ///   The attribute value, or null if the attribute
    ///   does not exist</return>
    /// <seealso cref="GetAttr(string)" />
    ///
    public string GetAttr(string name, string ns) {
        ...
    }
}
```

Sintassi del linguaggio: istruzioni ed espressioni

- Sostanzialmente simili a C/C++/Java
 - L'istruzione **switch (...)** non ha il comportamento *fall-through*
 - Non sono ammessi salti all'interno di blocchi
 - Introdotta l'istruzione **foreach (...)** per iterare su array e classi che implementano l'interfaccia System.IEnumerator<T>
 - È possibile controllare la generazione di *overflow* nelle espressioni mediante i costrutti **checked/unchecked**
 - Uso della parola chiave **var** per dedurre il tipo di una variabile dal valore che le viene assegnato

Sintassi del linguaggio: l'istruzione foreach

```
public static void Main(string[] args) {  
    foreach (string s in args) Console.WriteLine(s);  
}
```

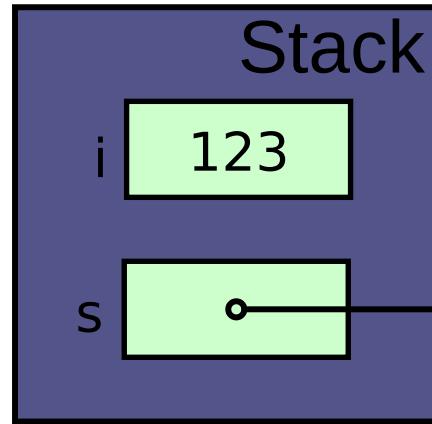
```
IList customers =...;  
foreach (Customer c in customers.OrderBy("name")) {  
    if (c.Orders.Count != 0) {  
        ...  
    }  
}
```

Tipi di dato

- Tutti i dati utilizzati in C# hanno un tipo
 - Il tipo definisce al gamma di valori consentita e l'insieme di operazioni lecite su un determinato dato
 - Tutti i tipi sono organizzati in una gerarchia di ereditarietà, la cui radice è `System.Object`
- Tipi valore:
 - Contengono direttamente il dato
 - Non possono valere “null”
 - Quando vengono copiati, si effettua una copia del valore
- Tipi riferimento:
 - Contengono un puntatore al valore
 - Il valore si trova sull'*heap gestito*
 - Possono valere “null”
 - Quando vengono copiati, si effettua una copia del puntatore

Tipi valore/riferimento

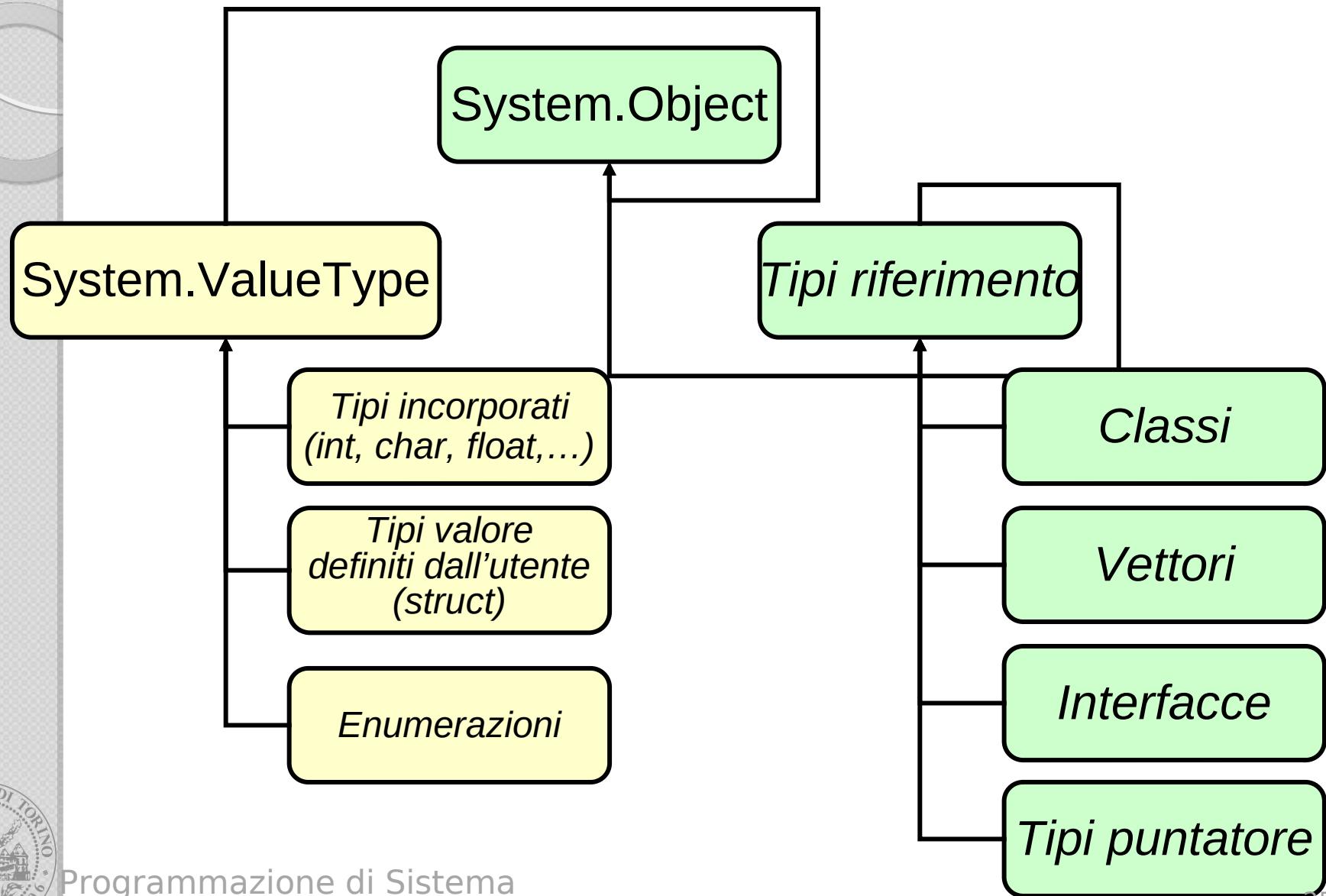
```
int i = 123;  
string s = "Hello world";
```



Heap

Flag di raggiungibilità:
quando diventa “false”,
l’oggetto può essere
eliminato

Organizzazione dei tipi

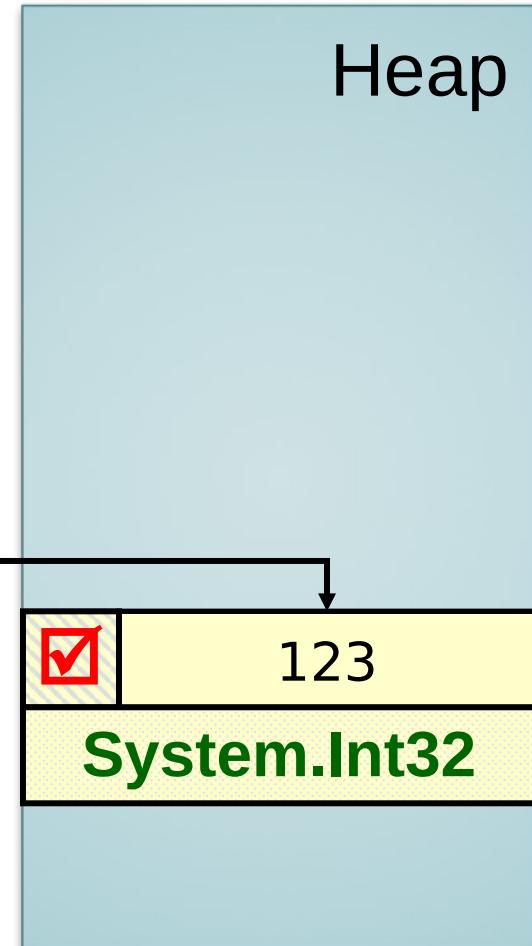
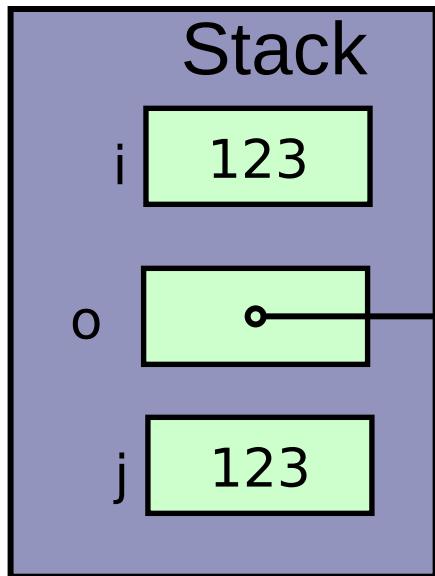


Boxing/unboxing (1)

- Anche i tipi valore hanno un corrispondente tipo riferimento
 - Il compilatore automaticamente converte tra i due formati quando necessario
 - Si facilita l'utilizzo di classi generiche (array, liste, ...) con tipi elementari
- **Boxing**
 - Trasformazione da valore a riferimento
 - Allocazione automatica dello spazio
- **Unboxing**
 - Trasformazione da riferimento a valore
 - Viene eseguito un controllo sul tipo: l'ambiente di esecuzione genera un'eccezione in caso di incompatibilità

Boxing/unboxing (2)

```
int i = 123;  
Object o = i;  
int j = (int) o;
```



Tipi predefiniti

- Numerici interi
 - Con segno: **sbyte**, **short**, **int**, **long**
 - Senza segno: **byte**, **ushort**, **uint**, **ulong**
- Numerici reali
 - **float**, **double**, **decimal**
- Non numerici
 - **char** (formato Unicode), **bool**
- Riferimento
 - **object** base di tutti i tipi (System.Object)
 - **string** sequenza immutabile di caratteri Unicode (System.String)

Strutture

- Dati aggregati simili alle classi
 - Possono avere campi, metodi, costruttori
 - Non supportano l'ereditarietà, solo l'implementazione di interfacce
- Informazioni di tipo valore
 - Allocate sullo stack, e non sull'heap
 - La chiamata al costruttore comporta la sola inizializzazione dei campi, non l'acquisizione di memoria dallo heap gestito
 - La copia comporta la replica dei dati
 - Permettono una gestione più efficiente della memoria

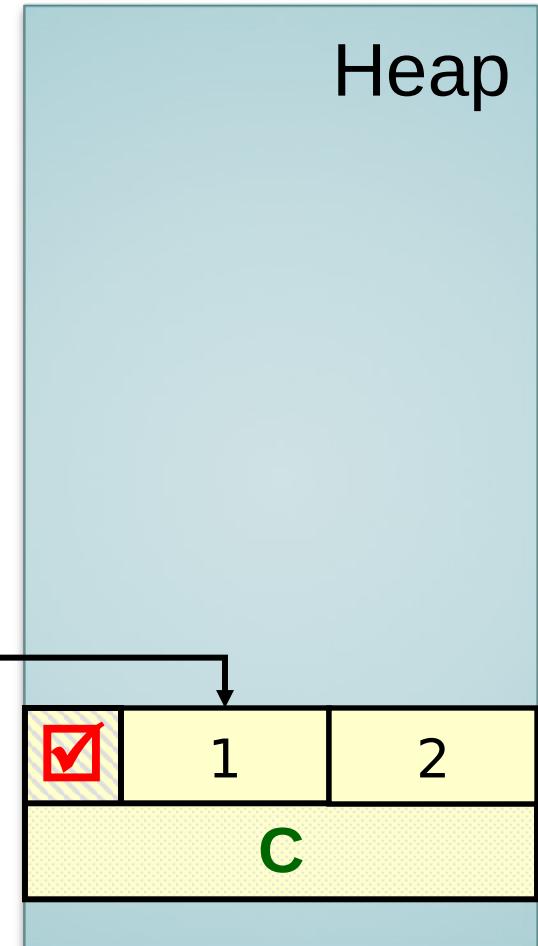
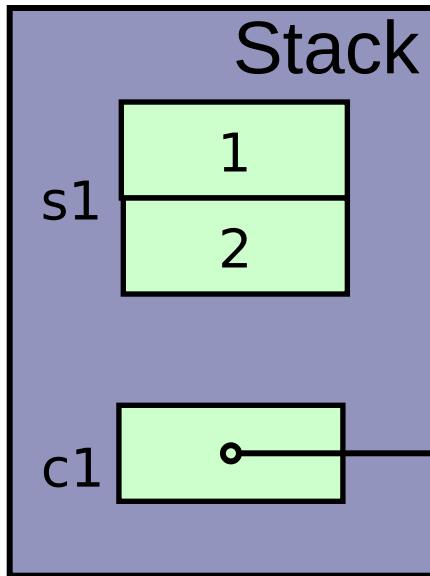
Classi

- Organizzate in una gerarchia di ereditarietà semplice
 - Ogni classe può però implementare molte interfacce
- Identificazione
 - **Spazio dei nomi**: identifica il “cognome” della classe, analogo al concetto di “package” in java
 - **Nome**: identifica la classe all’interno del proprio spazio
- Quattro modificatori di visibilità:
 - **public, protected, private, internal**
- Elementi delle classi
 - Costanti, campi, metodi, proprietà, indicizzatori, eventi, operatori, costruttori, distruttori
 - Relativi alle singole istanze o all’intera classe (**static**)

Classi e strutture

```
struct S {int a,b;...}  
class C {int a,b;...}
```

```
S s1=new S(1,2);  
C c1=new C(1,2);
```



Campi e costanti

- Campi
 - Variabili legate alle singole istanze
 - `public class Color { int r,g,b; }`
- Costanti
 - Campi preceduti dalla parola chiave **const**
 - Non possono essere modificati dopo l'inizializzazione
 - `public class Math {
 const static double pi=3.1415928;
}`

Metodi

- Funzioni che fanno riferimento all'istanza della classe
 - Possono avere parametri (anche in numero variabile) ed un tipo di ritorno

```
public class Color {  
    int r,g,b;  
    void reset( ) {  
        r=0; g=0; b=0;  
    }  
}
```

Interfacce

- Astrazioni del comportamento di un oggetto
 - Possono definire metodi, proprietà, indicizzatori ed eventi
- Ogni classe può implementare molte interfacce
 - Occorre fornire il codice corrispondente a tutte le funzioni dichiarate dalle singole interfacce
- interface IDataBound {
 void Bind(IDataBinder binder);
}

```
class EditBox: Control, IDataBound {  
    void IDataBound.Bind(IDataBinder binder) {...}  
}
```



Enumerazioni

- Tipi di dato con dominio dichiarato in modo esplicito
 - Ogni valore viene implementato da un numero
 - Sono supportate operazioni tra valori dello stesso tipo (+, -, ++, --, &, |, ^, ~)
 - È possibile indicare la quantità di memoria che occorre allocare
 - A differenza di altri linguaggi, i valori non sono convertibili automaticamente in interi
- ```
enum Color: byte {
 Red = 1,
 Green = 2,
 Blue = 4,
 Black = 0,
 White = Red | Green | Blue,
}
```



# Proprietà

- Campi virtuali di un oggetto per i quali si esplicitano le operazioni di assegnazione e lettura
  - Permettono di utilizzare una sintassi naturale, preservando il controllo sul codice generato
  - Normalmente si appoggiano su una variabile privata dello stesso tipo (che può essere creata automaticamente dal compilatore)

```
public class Button: Control {
 private string _caption;
 public string Caption {
 get {return _caption;}
 set {
 _caption = value;
 Refresh();
 }
 }
}
```

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

```
public class Test {
 public string Caption {
 get; set;
 }
}
```

# Indicizzatori

- Array virtuali associati ad un oggetto, per i quali si esplicitano le operazioni di accesso in lettura e scrittura alle singole celle
  - Gli indici possono essere non numerici
  - Sono possibili versioni *overloaded* dello stesso indicizzatore
  - Sono possibili indici multidimensionali

```
public class ListBox: Control {
 private string[] items;
 public string this[int index] {
 get {return items[index];}
 set {
 items[index] = value;
 Repaint();
 }
 }
}
```

```
ListBox lb = new ListBox();
lb[0] = "hello";
Console.WriteLine(lb[0]);
```

# Callback e delegati

- L'esecuzione di un algoritmo può richiedere che un metodo chiamato richiami un metodo del chiamante
  - Perché questo avvenga in modo parametrico, occorre che il chiamato conosca sia l'identità del chiamante che il metodo da invocare (con i relativi parametri)
- L'implementazione del pattern “callback” richiede una certa quantità di codice
  - In C/C++ questo si gestisce tramite liste di puntatori a funzioni ed ai relativi parametri
  - In Java si utilizzano interfacce “listener”, oggetti di tipo lista, metodi per registrare e cancellare gli ascoltatori e codice per iterare le notifiche

# Delegati

- Tipi di dato che encapsulano il pattern di chiamata a *callback*
  - Modellano liste di coppie (oggetto, metodo) da invocare a richiesta
- Si utilizzano per definire variabili locali, campi e/o parametri
  - Tali variabili contengono liste di istanze del delegato create mediante l'operatore *new*
  - Gli operatori `=`, `+=` e `-=` permettono di manipolare il contenuto associato a tali variabili
- Tali variabili possono essere invocate come metodi
  - Per ogni oggetto presente nella lista, invocano il metodo relativo, passando i parametri ricevuti
  - In caso di eccezione, il procedimento si arresta

# Uso di delegati

1. Si dichiara il tipo delegato
  - ❑ `delegate void Handler(string msg);`
2. Si dichiara una variabile (campo o parametro) avente il tipo del delegato
  - ❑ `Handler myHandler;`
3. Si assegna a tale variabile uno o più valori
  - ❑ `myHandler = new Handler(myObj.someMethod);`
  - ❑ `myHandler += new Handler(anotherObj.doSomething);`
  - ❑ `myHandler += evenAnotherObject.methodName;`
4. Si invoca il delegato
  - ❑ `myHandler("Message description");`

# Eventi (1)

- I delegati implementano gli elementi base necessari al pattern *callback*
  - Un'assegnazione impropria su una variabile di tipo *delegate* potrebbe cancellare possibili destinatari
  - La parola chiave **event** applicata ad una variabile *delegate* ne restringe l'utilizzo ai soli operatori **+=** e **=** (*add* e *remove*)
  - Solo il possessore dell'evento può invocare il delegato relativo (*raise*)
- I delegati associati ad un evento hanno tipicamente due parametri e ritornano void
  - Il primo parametro, di classe **System.Object**, rappresenta il mittente dell'evento
  - Il secondo parametro, di classe **System.EventArgs**, rappresenta gli eventuali dettagli associati all'evento

# Eventi (2)

```
public delegate void Handler(object sender, EventArgs e);
public class Button
{
 public event Handler Click;
 protected void OnClicked(...) {
 var clicked = Click; //in caso di accesso multithread
 if (clicked!=null)
 clicked(this,new MouseEventArgs(...));//Solleva l'evento
 }
}

public class Test
{
 public static void MyHandler(object sender, EventArgs e) {
 // React to event...
 }
 public static void Main() {
 Button b=new Button();
 b.Click += new Handler(MyHandler);
 ...
 }
}
```

# Eventi (3)

```
class Button {
 private Handler _clicked;
 public event Handler Clicked {
 add {
 Action old, @new;
 do {
 old = _clicked;
 @new = old + value;
 // chiama
 Delegate.Combine(old,value)
 } while (Interlocked.CompareExchange(
 ref _clicked, @new, old) !=
 old);
 }
 remove { ... } }
}
```

# Funzioni lambda (1)

- Ad un'istanza di delegato (o di evento) è possibile assegnare anche un'espressione od una funzione lambda
- Si usa rispettivamente la sintassi
  - (`<parametri>`) => `<espressione>`
  - (`<parametri>`) => { `<istruzioni>` ; }
- L'uso delle parentesi per delimitare i parametri è facoltativo se c'è un solo parametro
- Se nel corpo della funzione/espressione sono presenti variabili definite al suo esterno, queste vengono catturate per riferimento
  - Il ciclo di vita di tali variabili, viene automaticamente prolungato fino a che ne esiste un riferimento valido

# Funzioni lambda (2)

```
delegate bool D1();
delegate bool D2(int i);
class Test {
 D1 del1;
 D2 del2;

 public void method(int input) {
 int j=0; // j è una variabile locale inizializzata
 del1 = () => { j=10; return j>input; }
 del2 = (x) => { return x==j; }
 bool result = del1(); // true , j diventa 10
 }

 public static void Main() {
 Test test=new Test();
 test.method(5);
 bool result = test.del2(10); // true, j vale 10
 }
}
```

# Attributi (1)

- Annotazioni del codice sorgente accessibili durante l'esecuzione mediante *reflection*
  - Permettono di associare informazioni alle classi o ai loro elementi (metadati)
  - Introducono meccanismi flessibili per la gestione del codice senza richiedere soluzioni esterne (come file IDL)
  - Vengono memorizzati in istanze della classe **System.Attribute** o in classi da essa derivate
  - Il loro tipo viene controllato in fase di compilazione
- Usi tipici
  - URL della documentazione di una classe
  - Modello di esecuzione concorrente
  - Modellazione della serializzazione in XML
  - Servizi Web
  - ...



# Attributi (2)

```
public class OrderProcessor
{
 [WebMethod]
 public void SubmitOrder(PurchaseOrder order) {...}

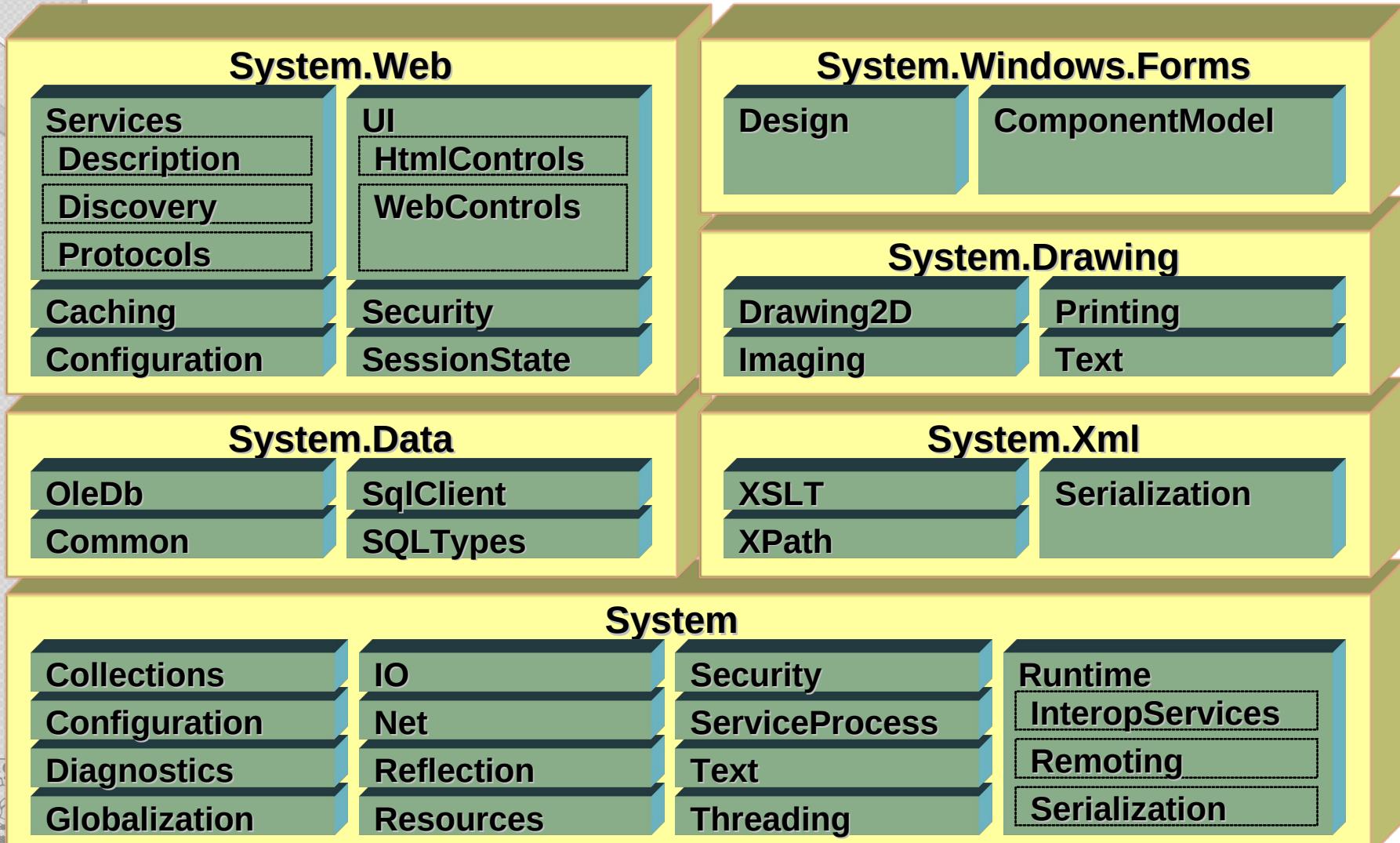
 [XmlRoot("Order", Namespace="urn:acme.b2b-schema.v1")]
 public class PurchaseOrder
 {
 [XmlElement("shipTo")] public Address ShipTo;
 [XmlElement("billTo")] public Address BillTo;
 [XmlElement("comment")] public string Comment;
 [XmlElement("items")] public Item[] Items;
 [XmlAttribute("date")] public DateTime OrderDate;
 }
 public class Address {...}
 public class Item {...}
}
```



# La libreria del framework

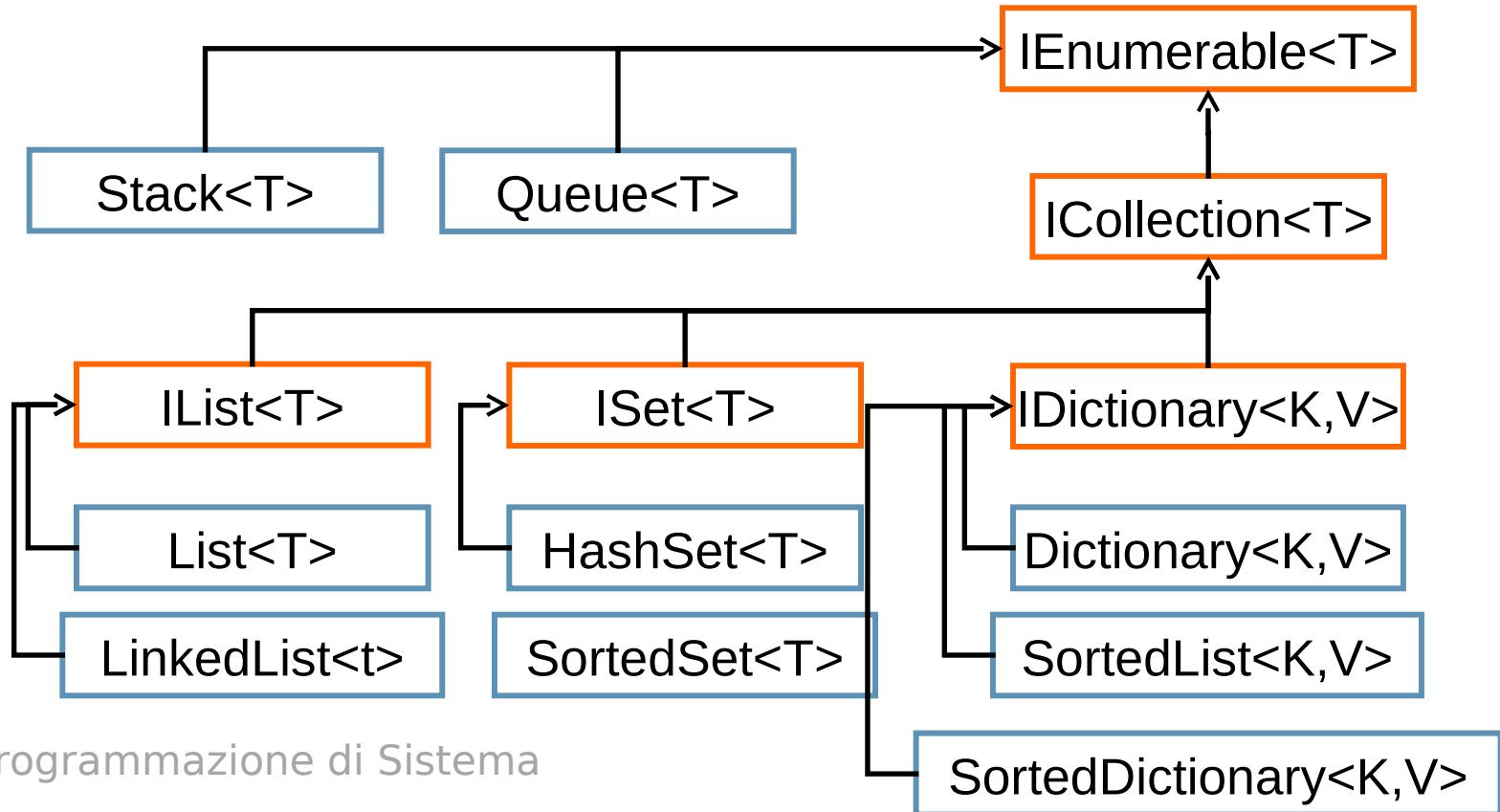
- API complessa ed articolata
  - Composta da più di 7000 tra classi, strutture, interfacce, enumerazioni e delegati
  - Circa 100 *namespace* organizzati gerarchicamente
  - La radice di tutte le classi è `System.Object`
- Molte le aree funzionali
  - Gestione di collezioni di oggetti
  - I/O legato a file e flussi
  - Manipolazione di espressioni regolari
  - Interazioni con la rete
  - Accesso ai dati
  - Riflessione
  - Interfacce grafiche
  - ...

# .NET Class Library



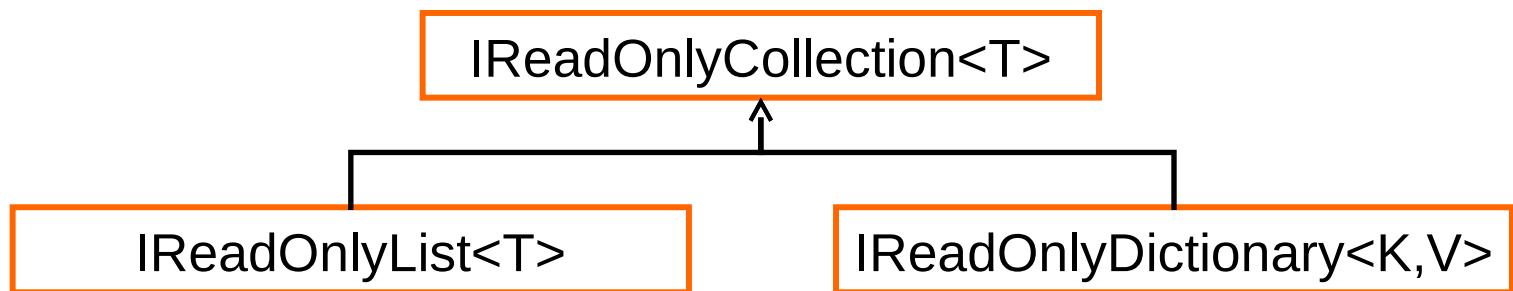
# Collezioni generiche

- Permettono l'uso di dati fortemente tipati attraverso il meccanismo di *type-erasure*
  - Definite nello spazio dei nomi System.Collections.Generics



# Collezioni in sola lettura

- Insieme di interfacce che escludono la possibilità di modificare i dati contenuti in una collezione
  - Implementate attraverso l'uso di un wrapper
  - In caso di cambiamento della collezione originale, i dati della collezione in sola lettura cambiano



# Collezioni osservabili

- Offrono eventi per annunciare l'inserimento, la cancellazione, lo spostamento e la sostituzione di elementi al loro interno
  - Definite nello spazio dei nomi System.Collections.ObjectModel

```
ObservableCollection<T>
```



# Collezioni concorrenti

- Offrono meccanismi per gestire, in modo thread-safe, collezioni di dati
  - Definite nello spazio dei nomi System.Collections.Concurrent

BlockingCollection<T>

ConcurrentBag<T>

ConcurrentDictionary<K,V>

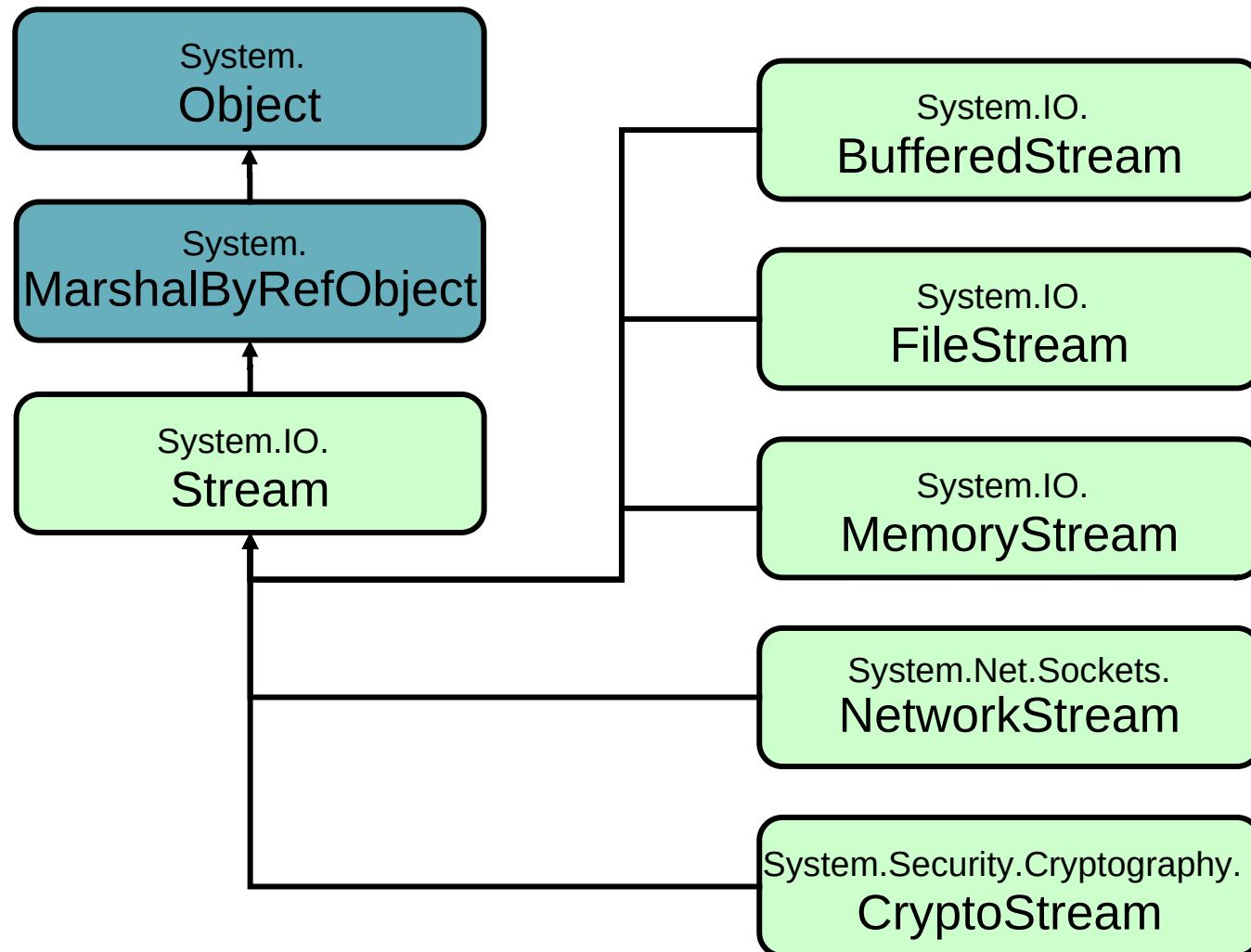
ConcurrentQueue<T>

ConcurrentStack<T>

# Input/Output

- .NET offre un ricco insieme di classi per eseguire operazioni I/O
  - Principalmente contenuto nel *namespace* System.IO
  - Estremamente raffinato e complesso se paragonato a *<Stdio.h>*
- La classe *Stream* costituisce la principale astrazione
  - Rappresenta una sequenza di byte
  - Offre i meccanismi di base per la lettura, la scrittura, il posizionamento all'interno di tale sequenza
  - Classe astratta: non può essere istanziata

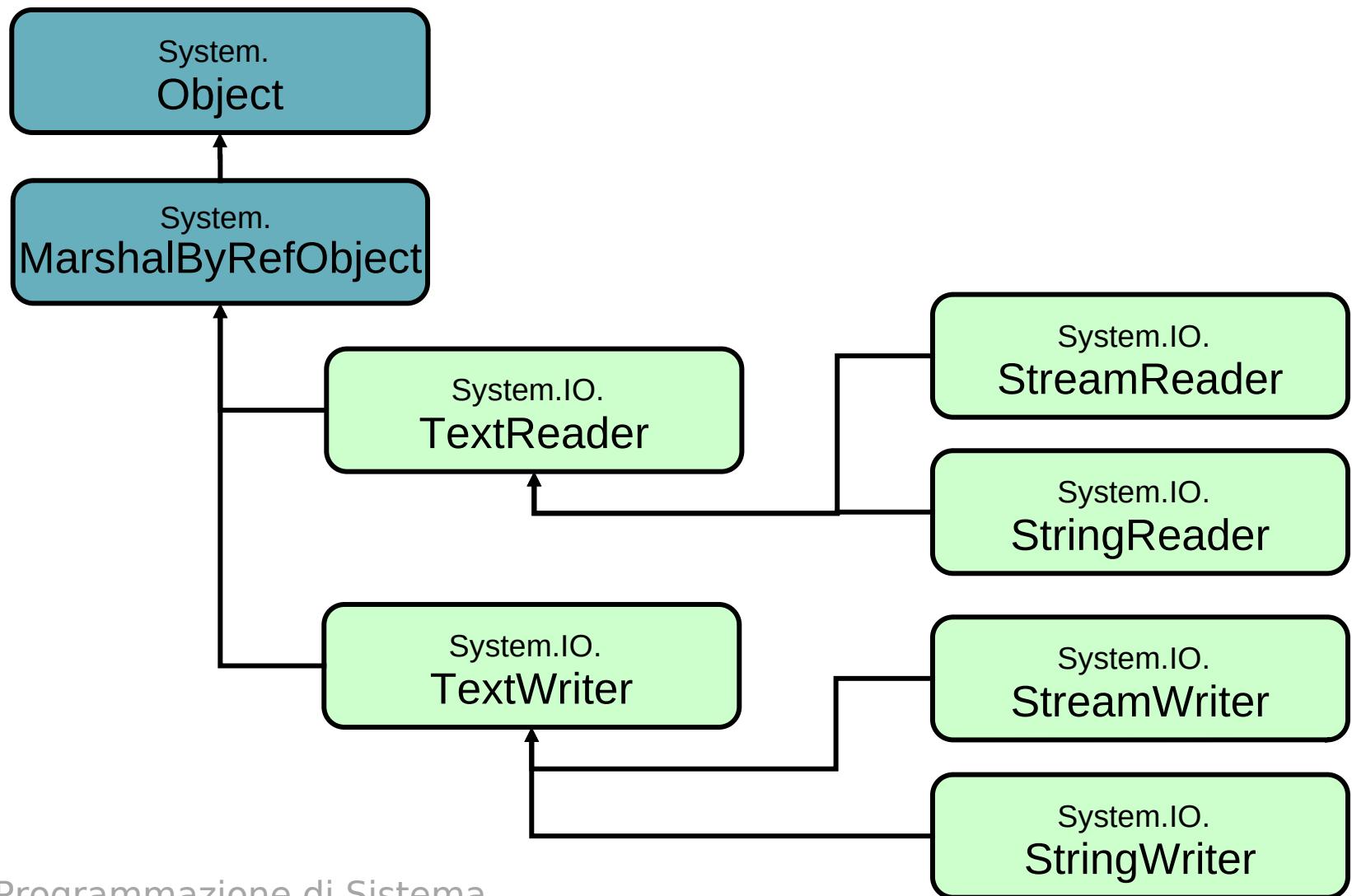
# Tipologie di flussi



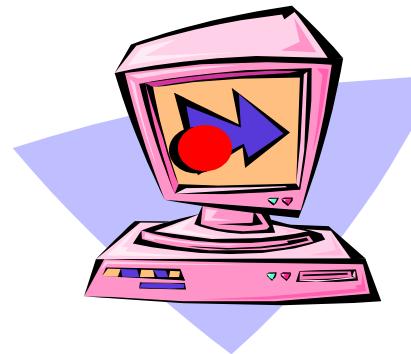
# Lettura e scrittura

- Le operazioni supportate dalla classe Stream riguardano la lettura e la scrittura di byte
- Se occorre leggere/scrivere caratteri occorre (de)codificarli
  - Possibili molti schemi: UTF-7, UTF-8, UTF-16,...
- Se occorre leggere/scrivere strutture binarie più articolate occorre (de)serializzarle
  - Molte più alternative...

# Operare con il testo



# Interfacce grafiche (I)



- A differenza di quanto accade nei programmi con interfaccia a caratteri, non c'è un unico flusso di comunicazione tra utente e programma
- Esiste uno spazio bidimensionale in cui sono **disegnati** gli “**strumenti**” necessari all'interazione
  - La comunicazione da e verso l'applicazione avviene direttamente utilizzando gli oggetti presenti sullo schermo
  - L'utente utilizza la tastiera e un dispositivo di puntamento (mouse) con cui indica e comanda l'applicazione

# Interfacce grafiche (II)



- Ciascun oggetto di ingresso (bottoni, liste, caselle di testo, ...) offre un numero limitato di alternative
  - Non è necessario introdurre analisi lessicale e sintattica del testo in ingresso
  - Non è noto però a priori in quale ordine avvenga l'interazione dell'utente con i diversi canali
- La struttura dei programmi muta radicalmente
  - Invece di richiedere l'esecuzione di una sequenza di operazioni, un programma deve adattarsi a **reagire** ad eventi esterni, cooperando con il sistema operativo
- Oltre ad implementare la propria logica interna, un'applicazione deve
  - Scegliere e comporre gli strumenti che formano le videate
  - Rispondere alle richieste dell'utente aggiornando coerentemente quanto visualizzato

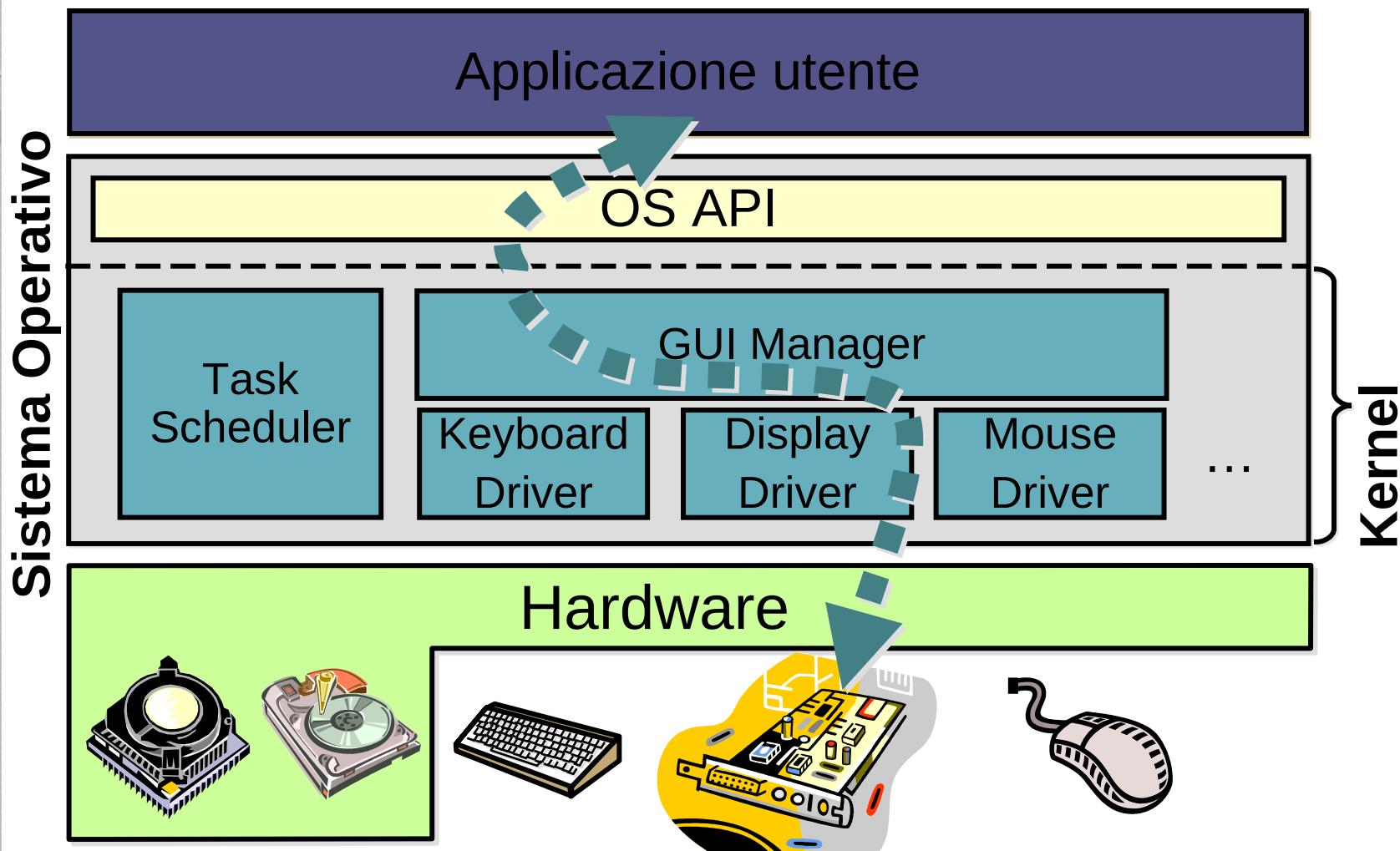
# Interfacce grafiche (III)

- Mouse, tastiera e schermo sono risorse condivise da molte applicazioni contemporaneamente
  - Nessuna interagisce direttamente con essi
  - Il sistema operativo ne scherma l'accesso offrendo opportuni meccanismi per consentire all'applicazione di interagire con tali periferiche
- Per interagire con lo schermo, un'applicazione crea una o più finestre
  - Porzioni logiche di schermo in cui il programma può disegnare
  - Ogni finestra ha un identificativo opaco univoco (handle) mantenuto dal sistema operativo
- Per interagire con mouse, tastiera, altre applicazioni, il sistema operativo mette a disposizione una coda di messaggi
  - Da essa l'applicazione attinge informazioni circa gli eventi che la riguardano e reagisce corrispondentemente

# GUI Manager (1)

- Componente software incaricato di gestire direttamente le periferiche e interagire con le applicazioni
  - Identifica “eventi significativi” e li notifica alle applicazioni sotto forma di opportune strutture dati (messaggi)
  - Gestisce lo smistamento dei messaggi verso le diverse applicazioni, identifica e risolve i conflitti
  - Permette l’accesso alle risorse grafiche

# GUI Manager (2)



# Programmazione reattiva

- Nei programmi di tipo GUI, non è prevedibile quale azione compia l'utente in quale momento
- È necessario predisporre un insieme di azioni da compiere quando si verifica un certo evento
  - Il programma “reagisce” agli eventi esterni invocando una **breve** procedura che
    - Aggiorna lo stato interno del programma
    - Richiede al GUI Manager l'aggiornamento della rappresentazione grafica



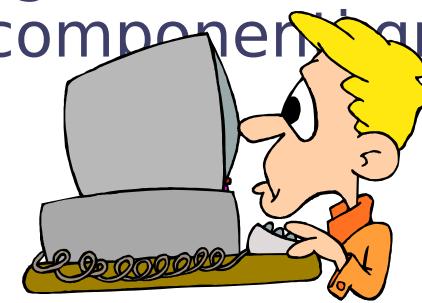
# Struttura delle applicazioni

- Per interagire con il GUI Manager un'applicazione deve
  - Iniziare una sessione di lavoro, creando la propria coda di messaggi
  - Richiedere la creazione di risorse grafiche (finestre, bottoni, campi di testo, ...)
  - Predisporre, per ogni widget ed evento atteso, un'opportuna routine di callback
  - Iterare sulla coda dei messaggi, inoltrando le richieste ricevute alla relativa callback
- Le modalità con cui queste operazioni vengono effettuate, dipendono dal sistema operativo e/o dagli eventuali strati software intermedi adottati

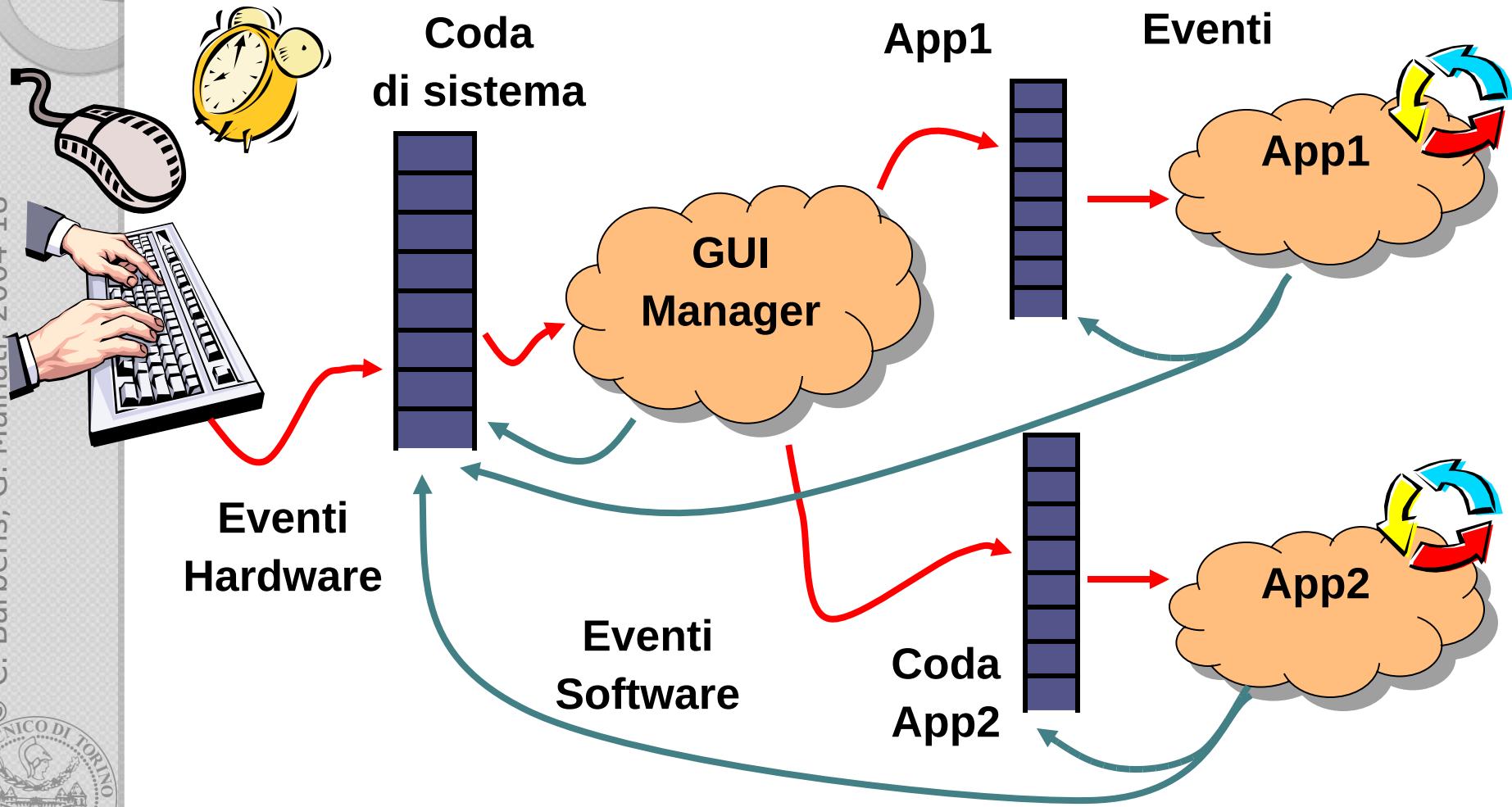


# Struttura di un programma

- Inizializzazione dell'interfaccia
  - Creazione della finestra principale e delle sue sottofinestre, indicando il comportamento che esse dovranno avere
- Attesa e reazione agli eventi
  - Si attende ciclicamente che il GUI Manager depositi un messaggio nella coda che è stata creata per l'applicazione e si esegue l'azione corrispondente
  - La libreria del framework traduce tali messaggi in invocazioni degli eventi corrispondenti nei diversi componenti grafici



# Flusso dei messaggi



# Windows form (1)

- Modello di programmazione integrato per lo sviluppo di applicazioni standard Win32
  - .NET mette a disposizione un insieme ricco e unificato di funzioni grafiche e di controllo per tutti i linguaggi
- Estendono la classe `System.Windows.Forms.Form`

# Applicazioni grafiche

- Modello di esecuzione reattiva a thread singolo
  - Per garantire la corretta consegna degli eventi, occorre invocare il metodo statico **Application.Run(...)** che incapsula il loop dei messaggi
- Molteplici opzioni per realizzare l'interfaccia grafica
  - Primitive GDI+ per l'accesso al video
  - Gestione dell'input tramite eventi legati a menu, mouse e tastiera
- Ampia rassegna di controlli predefiniti con cui personalizzare l'interfaccia
- Supporto avanzato per la realizzazione di finestre di dialogo

# Controlli grafici

- Tutti i componenti grafici derivano dalla classe `System.Windows.Forms.Control`
  - Essa definisce le proprietà, i metodi e gli eventi comuni ai componenti visuali: dimensioni, visibilità, posizione, organizzazione gerarchica, colore, font, ...
  - Gli eventi legati alla classe scandiscono il ciclo di vita di un componente grafico, informando i potenziali ascoltatori di ogni interazione e cambiamento di stato legato all'interazione del componente con l'ambiente di presentazione e/o l'utente finale



# Le principali proprietà della classe Control

- Posizione e dimensioni (in pixel)
  - Left, Right, Top, Bottom, Width, Height, Size, MinimumSize, MaximumSize, Padding, Margin,...
- Aspetto
  - BackColor, BackgroundImage, ForeColor, Font, Text, Cursor,...
- Organizzazione logica
  - Parent, Controls
- Interattività
  - Enabled, Visible

# Interfacce grafiche

- Un'interfaccia è costituita da un albero di oggetti grafici la cui radice è costituita da un'istanza della classe Form
  - Gli oggetti figli sono accessibili tramite la proprietà Controls (di tipo ControlCollections)
- L'albero viene costruito programmaticamente
  - VisualStudio offre la possibilità di “disegnare” l'interfaccia grafica sintetizzando il codice corrispondente alla rappresentazione visuale costruita dal programmatore
  - Ad ogni componente viene associato un campo della classe che estende Form
  - Il codice viene posto nel metodo “InitializeComponents()” che viene salvato in un file nascosto per evitare che sia manipolato direttamente dal programmatore

# Aggiungere l'interattività

- È possibile rendere interattivo l'albero dei componenti registrando ascoltatori sugli eventi corrispondenti
  - Interazione con il mouse (Click, DoubleClick, DragEnter, DragOver, DragLeave, DragDrop, Enter, Leave, ...)
  - Interazione con la tastiera (KeyDown, KeyUp, KeyPress, GotFocus, LostFocus, ...)
  - Interazione con il sistema di layout e visualizzazione (Paint, Print, Resize, Layout, ...)

# Interattività e azioni

- Le azioni svolte all'interno dei metodi delegati alla gestione degli eventi devono essere di breve durata per non bloccare l'interfaccia grafica
  - Se l'azione dura a lungo, occorre delegarne l'esecuzione ad un thread secondario (od al thread pool)
- E' importante notare che i metodi (le proprietà, gli indicizzatori, gli eventi,...) di un componente grafico possono essere manipolati solo nel contesto del thread che li ha creati
  - Fa eccezione il metodo Invoke(...): questo può essere chiamato da un thread secondario per richiedere l'esecuzione di un metodo da parte del thread principale

# Grafica 2D

- L'evento Paint offre l'accesso ad un contesto grafico GDI+
  - Attraverso la proprietà Graphics dell'oggetto PaintEventArgs che viene notificato al gestore
  - Tramite esso è possibile utilizzare tutte le funzionalità di disegno sullo schermo, tracciamento di scritte e manipolazione di immagini offerte dalla libreria nativa

# HelloForm.cs

```
using System;
using System.Windows.Forms;
using System.Drawing;

class HelloForm: Form {
 HelloForm() {
 Text = "HelloForm";
 Paint+= PaintWindow;
 }
 private void PaintWindow(Object s,PaintEventArgs e) {
 e.Graphics.DrawString("Hello Windows Form", Font,
 new SolidBrush(Color.Black), ClientRectangle);
 }
 public static void Main() {
 Application.Run(new HelloForm());
 }
}
```



# Windows Presentation Foundation (WPF)

Anno Accademico 2017-18

# Introduzione

- Uno dei requisiti essenziali per una buona applicazione grafica è avere un “look” accattivante
- Esistono strumenti più potenti ed efficienti di GDI
  - Libreria ormai datata
  - Poco efficiente dal punto di vista della gestione delle risorse

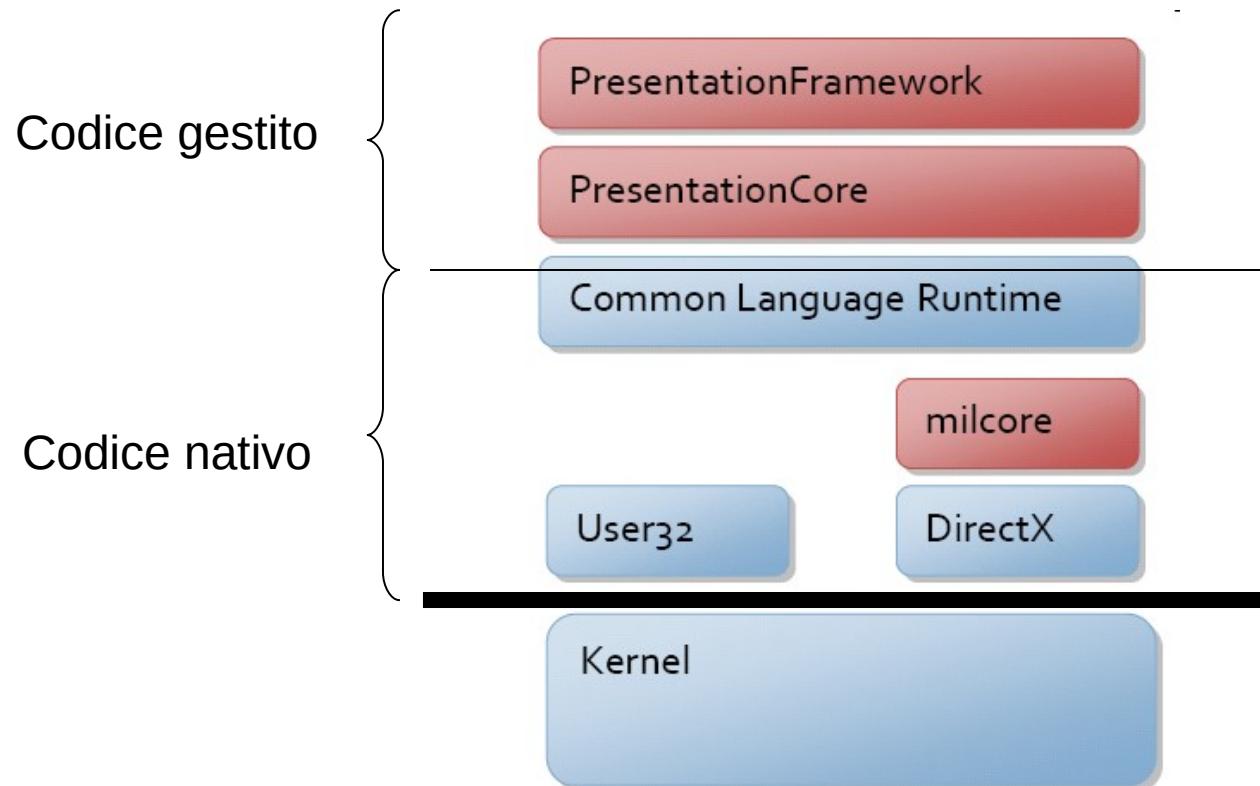


# Windows Presentation Foundation (1)

- Offre un approccio unificato per l'integrazione all'interno di applicazioni di componenti di tipo diverso:
  - Documenti
  - Contenuti multimediali (audio, video)
  - Grafica 2D/3D
  - Oggetti MFC, Win32

# Windows Presentation Foundation (2)

- Componente del framework .NET 3.0
  - disponibile a partire da XP SP2



# Windows Presentation Foundation (3)

- Interamente basato su grafica vettoriale
  - Sfrutta al meglio le potenzialità dei calcolatori moderni
- Specificamente pensato per la realizzazione di interfacce ad alto impatto visivo
- Basato sul motore grafico DirectX

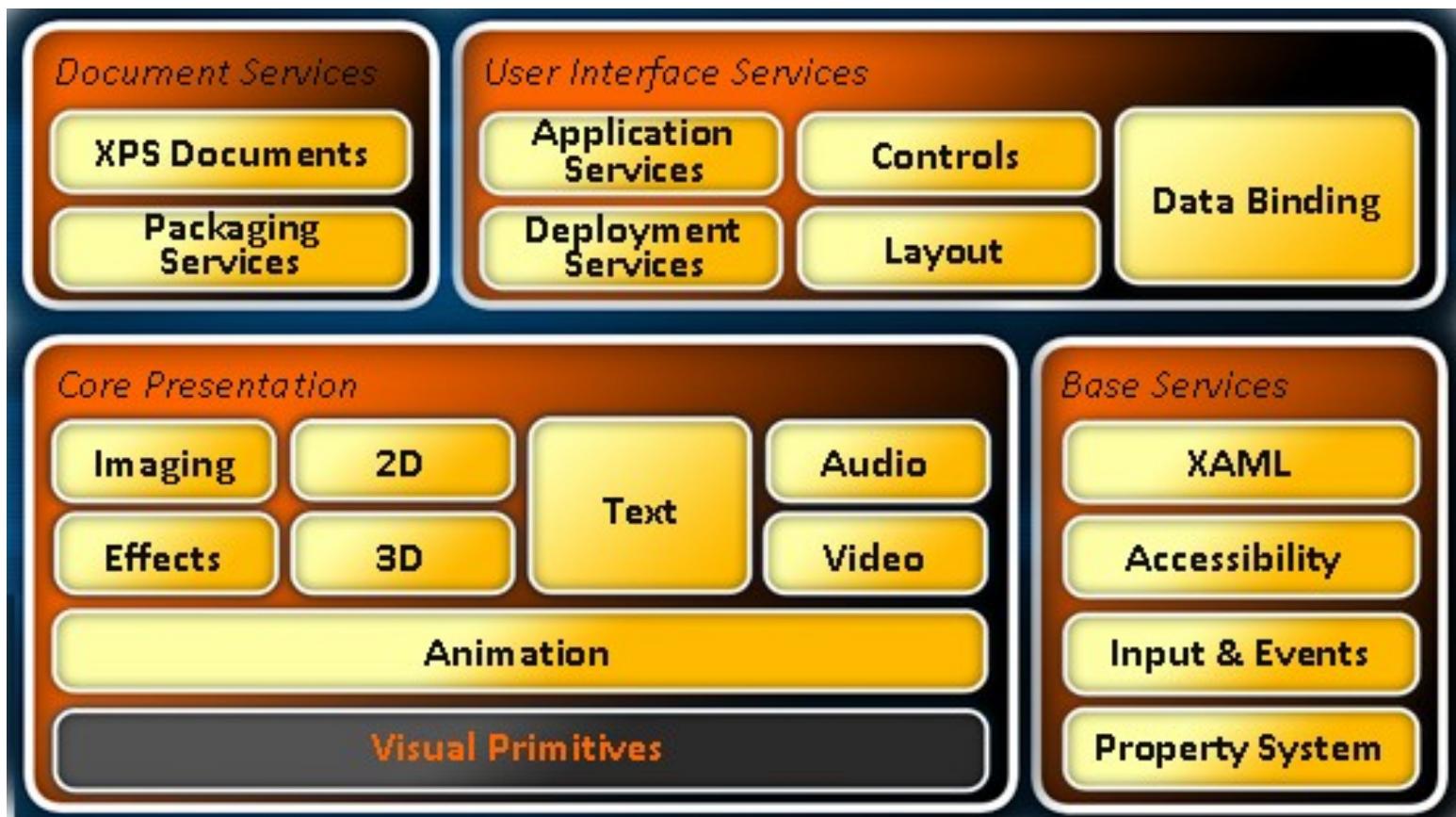


# Windows Presentation Foundation (4)

- Unisce un linguaggio dichiarativo a uno procedurale
  - Separazione esplicita tra logica applicativa e interfaccia grafica
  - Netta suddivisione dei ruoli di grafico e sviluppatore
- Procedura di installazione semplificata
  - Tramite browser
  - Applicazioni stand alone



# Architettura WPF



# Visual rendering

- WPF introduce diverse innovazioni dal punto di vista del rendering degli oggetti grafici:
  - Modalità “retained”
  - Grafica vettoriale
  - Unità di misura grafica indipendente dal dispositivo

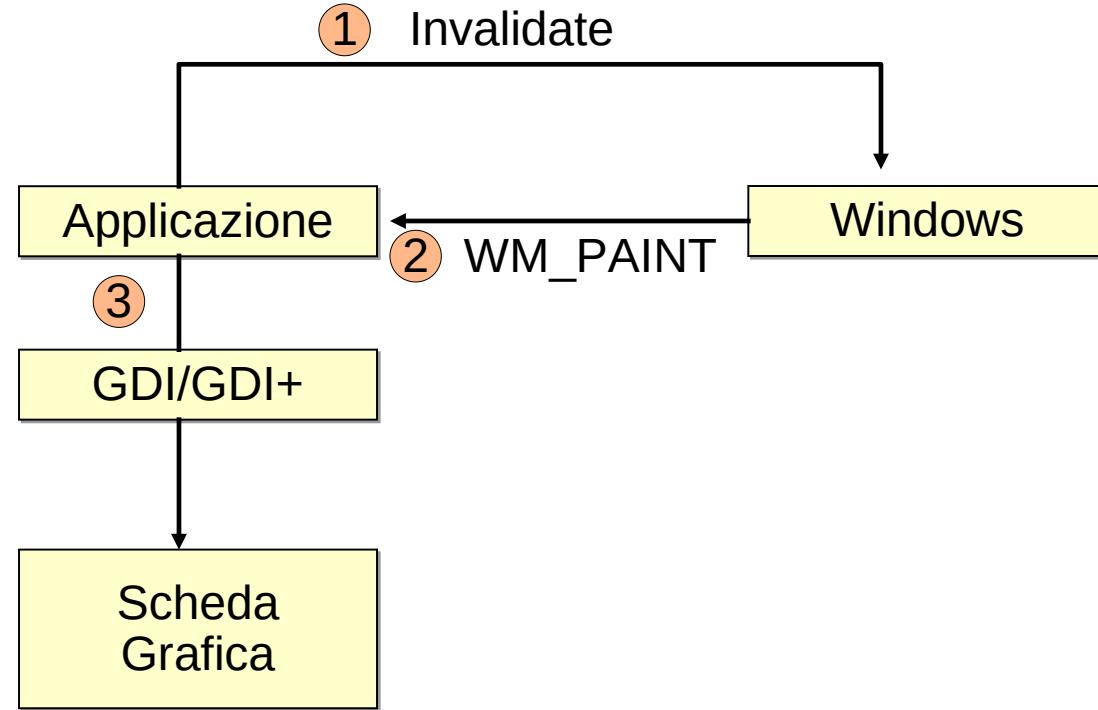


# Modalità immediate

- GDI/GDI+ disegna gli oggetti grafici secondo una modalità detta “immediate”
  - Applicazione direttamente responsabile del ridisegno della propria client area
  - Gli oggetti grafici vengono (ri)disegnati invocando una callback dell'applicazione che si occupa del rendering
  - GDI non conserva alcuna informazione sugli oggetti disegnati:
    - Riceve l'immagine dall'applicazione e la copia nel frame buffer della scheda video



# Modalità immediate



# Modalità retained

- WPF usa una modalità detta “retained”
  - Ciascun oggetto grafico contiene dati sul proprio rendering, che possono essere:
    - Dati vettoriali
    - Immagini
    - Glifi
    - Video
  - Il sistema grafico mantiene in una struttura ad albero i dati sul rendering degli oggetti che compongono la scena da visualizzare



# Modalità retained

- WPF traduce questi dati in una serie di istruzioni che passa alla GPU della scheda grafica
  - Quando arriva una richiesta di disegno, tali istruzioni vengono eseguite
- A differenza della modalità “immediate”, è il sistema grafico WPF che gestisce autonomamente le operazioni di ridisegno, non l'applicazione
  - Ottimizzando le operazioni

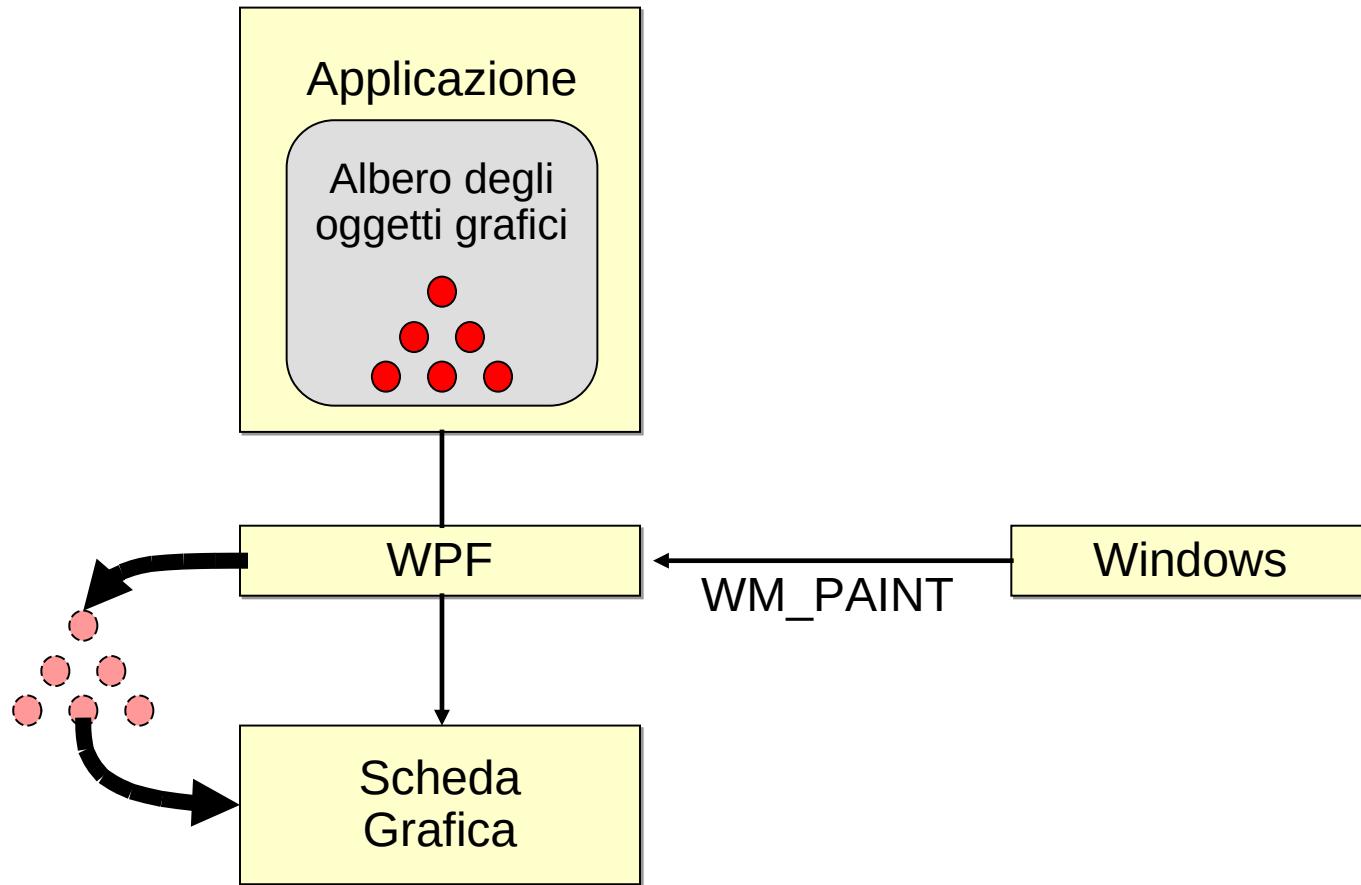


# Modalità retained

- Questo comporta alcuni vantaggi:
  - Lo sviluppo delle applicazioni grafiche viene semplificato
    - Non deve essere implementata la parte di ridisegno degli oggetti grafici
  - Maggiore efficienza
    - Caching delle istruzioni di rendering
    - Vengono sfruttate a pieno le potenzialità della scheda video



# Modalità retained

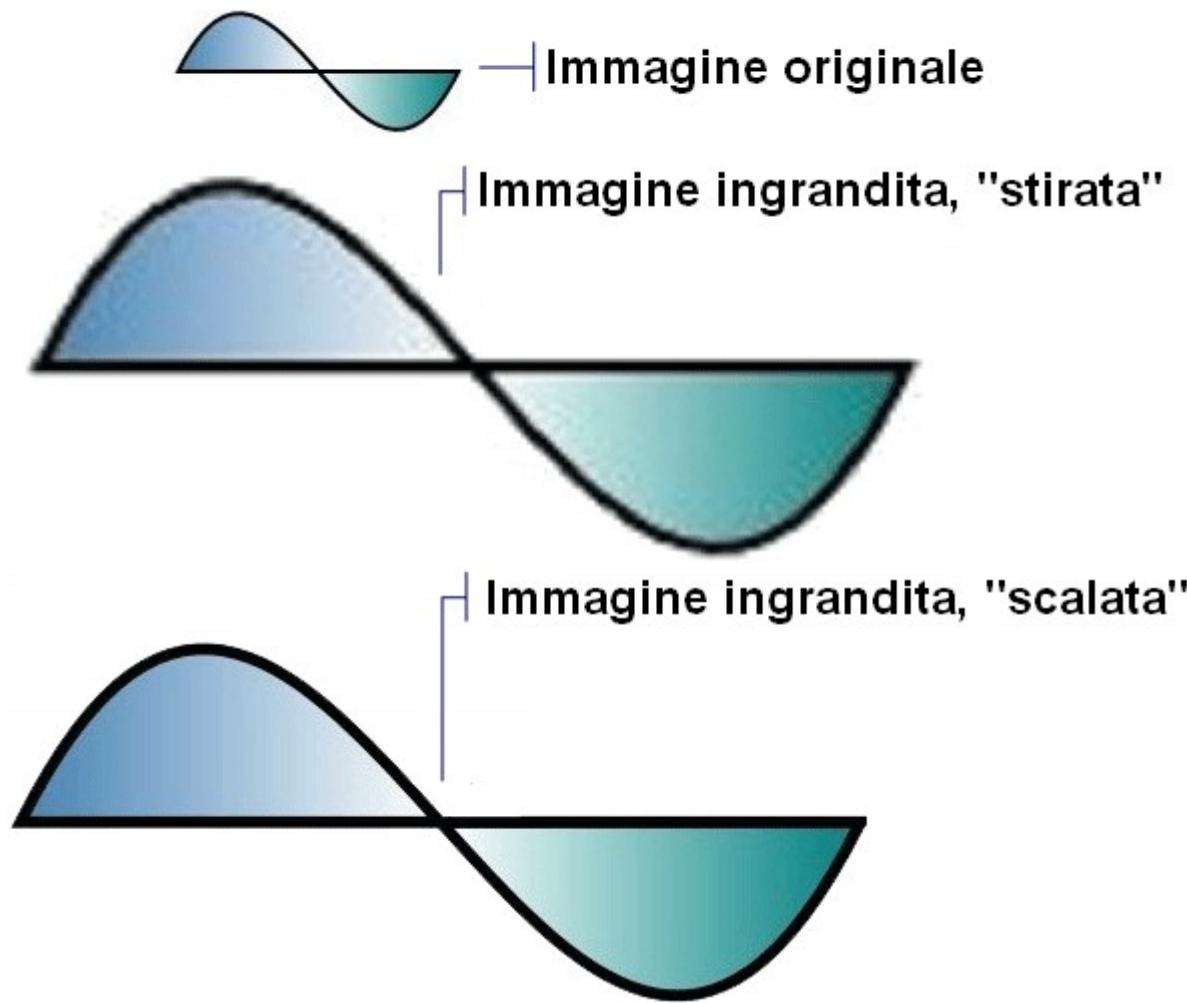


# Grafica vettoriale

- Consente di descrivere un elemento grafico in base ad una serie di primitive grafiche
  - Soluzione più sofisticata della tradizionale grafica “bitmap”
  - Es: font TrueType descritti come insieme di linee, curve e comandi di riempimento
- Un’immagine ridimensionata viene ridisegnata e non perde qualità rispetto all’originale
  - L’immagine viene “scalata” anziché “stirata”



# Grafica vettoriale



# Unità di misura

- Due fattori determinano la dimensione fisica del testo e oggetti grafici sullo schermo
  - Risoluzione dello schermo: numero di pixel visualizzati
  - Densità dei pixel (DPI, dot per inch): dimensione di un pollice ideale espressa in pixel
- WPF supporta schermi con differenti densità e usa come unità di misura primaria i “device independent pixel” anziché i pixel hardware
  - Consente di riadattare automaticamente testo ed elementi grafici a diverse risoluzioni e DPI mantenendo costante la dimensione



# Sviluppo in WPF

- E' possibile sviluppare applicazioni usando:
  - Codice C# che istanzia gli oggetti e li collega tra loro nel modo voluto
  - XAML (eXtensible Application Markup Language), linguaggio descrittivo basato su XML
  - Un mixto dei due linguaggi
    - XAML per la descrizione dell'interfaccia
    - C# per l'implementazione della logica



# XAML

- Usato in WPF per creare e inizializzare oggetti secondo una data gerarchia
- Tutte le classi in WPF hanno solo un costruttore privo di argomenti
  - Si utilizzano le proprietà per configurare il comportamento e l'aspetto degli oggetti
  - Questo facilita l'integrazione con un linguaggio descrittivo



# Vantaggi di XAML

- Codice compatto
- Istruzioni leggibili, anche per i non programmatori
- Separazione del codice per l'aspetto grafico da quello della logica applicativa
- Le proprietà degli oggetti vengono specificate come proprietà di elementi XML
- La conversione dei valori dei tipi viene fatta in automatico



# Esempio in C#

```
public partial class Window1 : Window {
 public Window1() {
 InitializeComponent();
 StackPanel stackPanel = new StackPanel();
 this.Content = stackPanel;
 this.Background = new LinearGradientBrush(Colors.Wheat, Colors.White,
 new Point(0,0), new Point(1,1));
 TextBlock textBlock = new TextBlock();
 textBlock.Margin = new Thickness(20);
 textBlock.FontSize = 20;
 textBlock.Foreground = Brushes.Blue;
 textBlock.Text = "Hello World";
 stackPanel.Children.Add(textBlock);
 Button button = new Button();
 button.Margin = new Thickness(10);
 button.Height = 25; button.Width = 80;
 button.HorizontalAlignment = HorizontalAlignment.Right;
 button.Content = "OK";
 stackPanel.Children.Add(button);
 }
}
```



# Esempio equivalente in XAML

```
<Window x:Class="hello.Window1"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="Window1" Height="145" Width="298" >
 <Window.Background>
 <LinearGradientBrush StartPoint="0 0" EndPoint="1 1" >
 <GradientStop Offset="0" Color="Wheat"/>
 <GradientStop Offset="1" Color="White"/>
 </LinearGradientBrush>
 </Window.Background>
 <StackPanel>
 <TextBlock Margin="20" FontSize="20" Foreground="Blue">
 Hello World </TextBlock>
 <Button Margin="10" HorizontalAlignment="Right"
 Height="25" Width="80">OK</Button>
 </StackPanel>
</Window>
```



# Reattività

- Le applicazioni dotate di interfaccia grafica devono essere reattive
  - Richiedono l'impiego di più di un thread
  - Un thread principale
    - raccoglie continuamente l'input dell'utente
  - Uno o più thread secondari
    - eseguono i compiti in base agli input e comunicano i risultati al thread principale il quale può aggiornare conseguentemente l'interfaccia utente



# Thread in WPF

- Ogni applicazione WPF ha associati due thread all'atto della creazione
  - *Rendering thread*
    - Eseguito in background
  - *UI thread:*
    - Raccoglie gli input
    - Aggiorna l'albero degli oggetti grafici
    - ...
- E' possibile creare altri thread per aumentare la reattività del programma
  - Sconsigliato, aumenta la complessità



# Dispatcher

- Oggetto che fa da intermediario nello scambio di messaggi tra thread differenti
- Concettualmente analogo al gestore degli eventi in Win32
- Mantiene al suo interno una coda
  - Seleziona ed inoltra i messaggi secondo criteri di priorità
  - Ogni UI Thread deve essere associato ad un Dispatcher



# Dispatcher

- UI Thread e Rendering Thread fanno accesso alla coda del Dispatcher
- UI Thread
  - Inserisce richieste a fronte del verificarsi di un evento
  - Legge i messaggi di notifica da parte degli altri thread
- Rendering Thread
  - Riceve richieste di inizio di un'attività
  - Inserisce messaggi sull'esito delle attività svolte

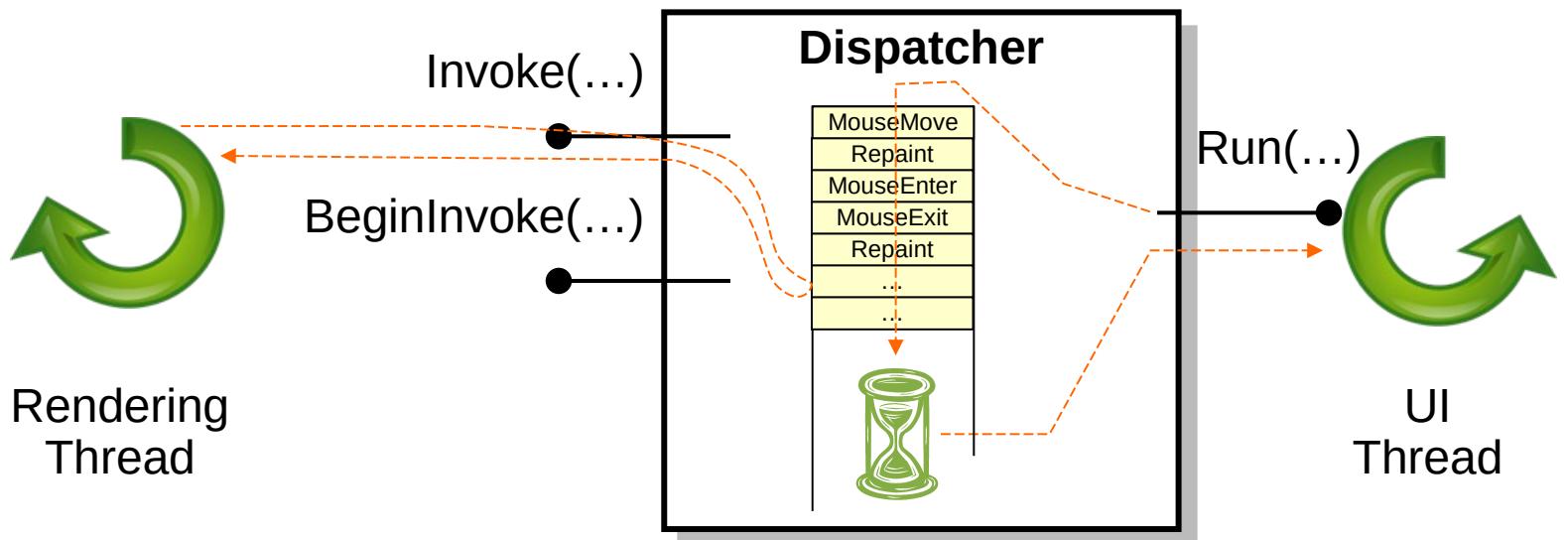


# Dispatcher

- Offre due metodi per inserire messaggi nella coda:
  - Invoke(): sincrono, bloccante
  - BeginInvoke() asincrono, non bloccante
- La coda deve essere svuotata velocemente
  - Quanto più velocemente ciò accade tanto più reattiva risulterà l'applicazione



# Dispatcher



# Eseguire compiti lenti

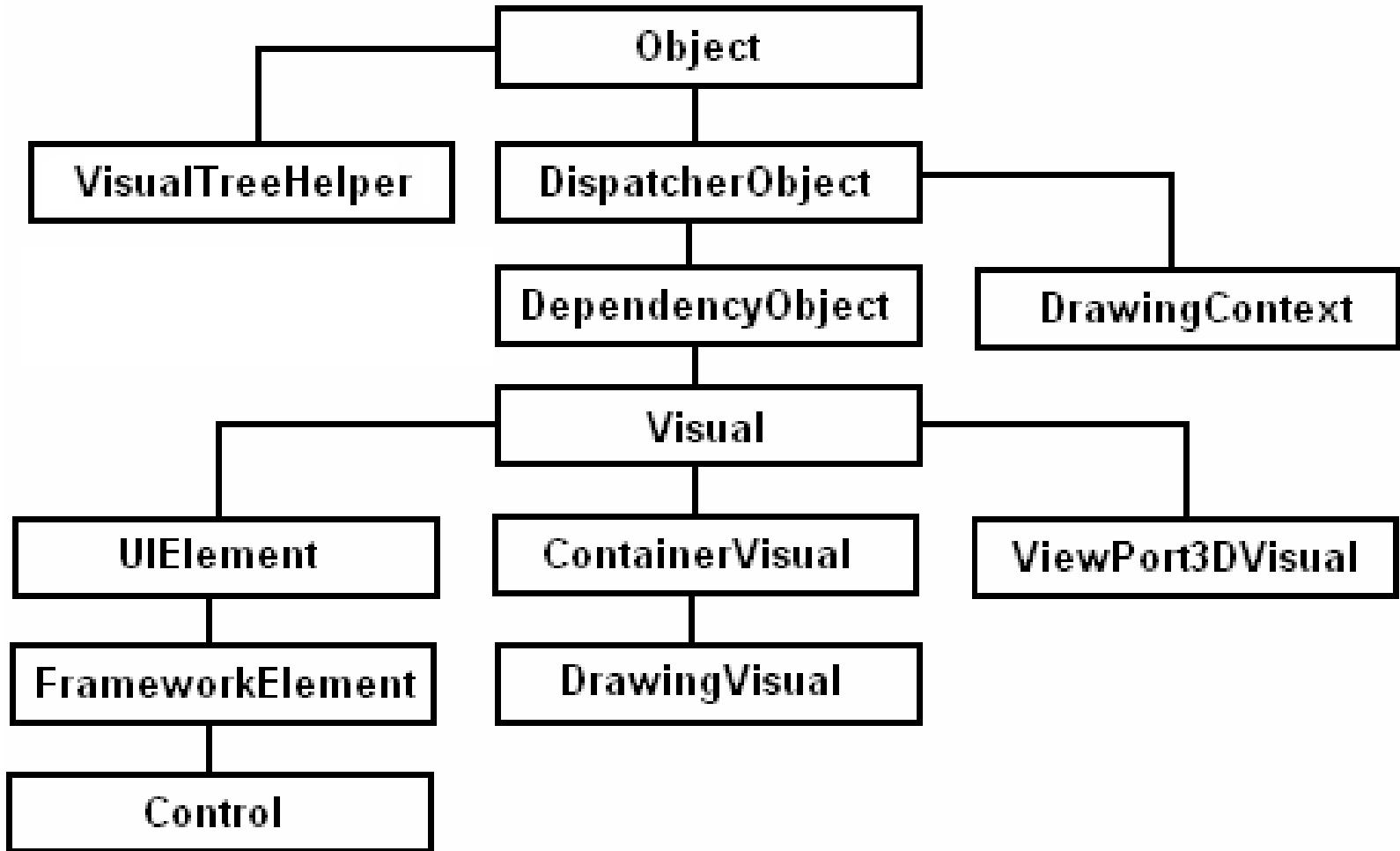
```
private void b1_Click(object sender, RoutedEventArgs e)
{
 b1.IsEnabled = false;

 Task.Run(
 () => {
 DoSlowWork();
 Dispatcher.BeginInvoke(
 DispatcherPriority.Normal,
 new Action(DoUIUpdate));
 });
}

private void DoSlowWork() {
 //...
}

private void DoUIUpdate() {
 b1.IsEnabled = true;
}
```

# Classi base in WPF



# System.Threading.DispatcherObject

- Radice di tutti gli oggetti dotati di una presentazione grafica WPF
- Mantiene un riferimento all'oggetto Dispatcher associato al thread in esecuzione
  - Garantisce che tutte le operazioni effettuate dalle proprie istanze avvengano nel contesto del thread responsabile della gestione del Dispatcher cui è associato



# DependencyProperties

- Tutte le informazioni relative al contenuto grafico sono memorizzate nelle proprietà dei suoi nodi
  - La classe System.Windows.DependencyObject offre un meccanismo efficiente per collegare tra loro due proprietà, e gestire la percolazione dei valori attraverso l'albero logico che descrive la scena
- Il valore della proprietà non viene duplicato finché non cambia, anche se appartiene a due oggetti diversi di una certa classe
  - Minore occupazione di memoria
- Gli oggetti di questo tipo sono osservabili
  - Questo permette di registrarsi su una proprietà e ricevere notifiche ogni volta che cambia



# Implementare una proprietà

- Occorre fare riferimento ad un'istanza di DependencyProperty dichiarata a livello classe
  - Il framework provvederà ad allocare memoria specifica per l'istanza corrente

```
public partial class MyWindow: Window {

 public static readonly DependencyProperty SizeProperty =
 DependencyProperty.Register("Size",
 typeof(double),
 typeof(MyWindow),
 new UIPropertyMetadata(3));

 public double Size {
 get { return (double) GetValue(SizeProperty); }
 set { SetValue(SizeProperty, value); }
 }
 //...
}
```



# Registrarsi al cambiamento

```
DependencyPropertyDescriptor sizeDescr =
 DependencyPropertyDescriptor.FromProperty(
 MyWindow.SizeProperty, typeof(MyWindow));

if (sizeDescr != null)
{
 sizeDescr.addValueChanged(this,
 delegate
 {
 // Add your property changed logic here...
 });
}
```

# Proprietà aggiunte

- Spesso gli oggetti contenitori hanno bisogno di dati appartenenti agli oggetti contenuti per definire il proprio comportamento
  - Un oggetto contenitore può possedere le definizioni delle proprietà di un altro oggetto ~~in esso contenuto~~

```
<Canvas>
 <Button Canvas.Top="20" Canvas.Left="20"
Content="Ok"/>
</Canvas>
```



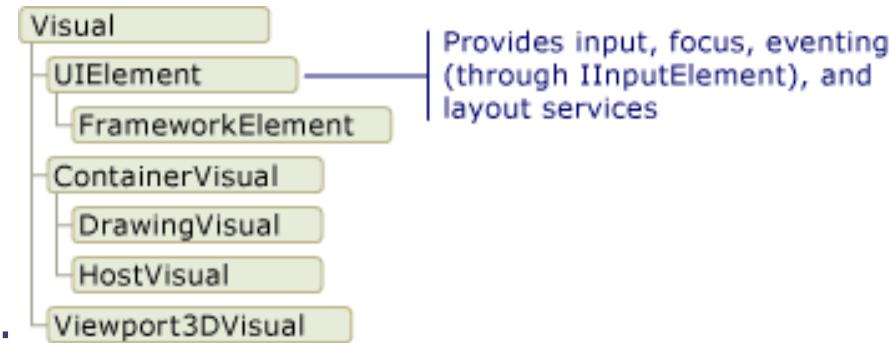
# Visualizzazione di oggetti

- Un oggetto graficamente rappresentabile deve essere istanza di una sottoclassificazione di System.Windows.Media.Visual
- Base per implementare nuovi controlli
  - Paragonabile a WindowHandle in Win32
- Introduce un meccanismo di “caching” delle istruzioni di ridisegno per massimizzare le prestazioni



# Classe Visual

- Offre supporto per:
  - Output display
  - Trasformazioni
  - Clipping
  - Hit test
  - Calcolo dei margini
- Non offre supporto per:
  - Event handling
  - Layout e stili
  - Data binding



# Gerarchie di elementi grafici

- La classe Visual consente di organizzare gli oggetti da visualizzare in una struttura ad albero
  - Ciascun oggetto può contenere le istruzioni e i metadati circa la propria visualizzazione
- Ciascun elemento Visual comunica con gli altri dello stesso albero tramite un protocollo specifico
  - Basato sullo scambio di messaggi

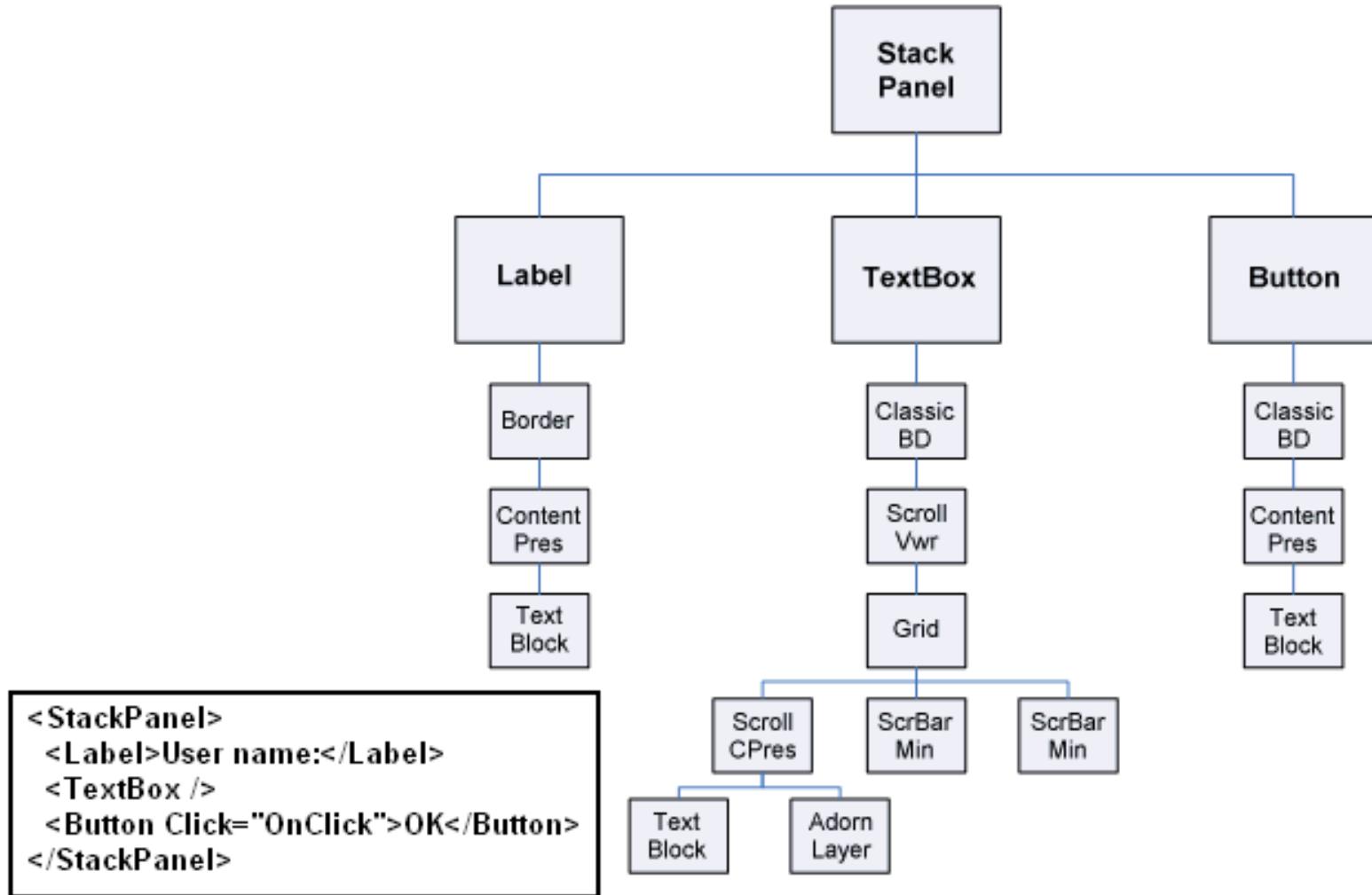


# Gerarchie di elementi grafici

- La gerarchia tra elementi grafici può essere considerata sotto due punti di vista:
  - “Visual Tree”: gerarchia degli elementi visualizzati su schermo
  - “Logical Tree”: rappresenta la gerarchia degli elementi dell’applicazione in fase di esecuzione
    - Normalmente non serve manipolarlo direttamente
    - Può rappresentare elementi non strettamente visuali
      - Ad es: “ListItem”
    - Concetto utile per comprendere meccanismi di ereditarietà e gestione degli eventi

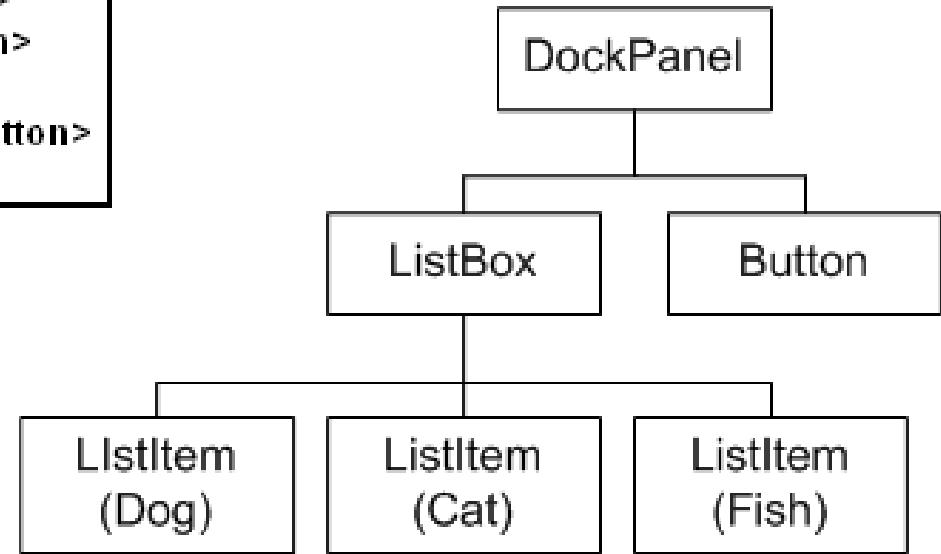


# Esempio: Visual Tree



# Esempio: Logical Tree

```
<DockPanel>
 <ListBox>
 <ListBoxItem>Dog</ListBoxItem>
 <ListBoxItem>Cat</ListBoxItem>
 <ListBoxItem>Fish</ListBoxItem>
 </ListBox>
 <Button Click="OnClick">OK</Button>
</DockPanel>
```



# VisualTreeHelper

- Classe helper statica, deriva direttamente da Object
- Fornisce funzionalità di basso livello
- Utile in casi molto specifici
  - Es. sviluppo di controlli personalizzati
- Consente di esplorare l'albero visuale degli elementi
- Fornisce metodi per effettuare “hit test”
  - Controllare che un determinato punto appartenga a un certo elemento grafico



# Classe DrawingContext

- Permette di popolare un Visual con un contenuto grafico vero e proprio
- Tramite una serie di metodi:
  - DrawLine(), DrawImage(), DrawText(), DrawVideo(), ...
  - Non disegna in realtime ma memorizza un insieme di informazioni sul disegno utilizzate in seguito
- Non viene instanziato direttamente ma viene acquisito
  - Metodo Open() di DrawingGroup
  - Metodo RenderOpen() di DrawingVisual



# Esempio

```
// Crea un oggetto DrawingVisual che conterrà un
rettangolo
private DrawingVisual
CreateDrawingVisualRectangle(Point p, Size s) {
 DrawingVisual drawingVisual = new DrawingVisual();
 // Si richiede il DrawingContext associato
 DrawingContext drawingContext =
 drawingVisual.RenderOpen();
 // Crea un rettangolo e lo inserisce nel DrawingContext.
 Rect rect = new Rect(p, s);
 drawingContext.DrawRectangle(Brushes.LightBlue,
 (Pen)null, rect);
 //Affida il contenuto del DrawingContext al sotto-sistema
 grafico
```

```
 drawingContext.Close();
 return drawingVisual;
}
```



# Layout

- Gli elementi grafici dotati di layout sono quelli derivati da UIElement
- La posizione e dimensione di ciascun oggetto viene determinata in due fasi
  - Measure: l'elemento dichiara la dimensione ottimale al contenitore
  - Arrange: il contenitore ricalcola la posizione e la dimensione degli elementi contenuti secondo i propri criteri



# Input ed eventi

- Gli input generati dalle periferiche vengono instradati attraverso il kernel di Windows e il sistema User32 fino al processo e al thread opportuno
- Un messaggio User32 viene convertito da WPF secondo il proprio formato e infiltrato al Dispatcher



# Routing degli eventi

- Ciascun input è convertito da WPF in almeno due eventi
  - Evento “preview”
  - Evento “effettivo”
- Ad ogni evento è associato il concetto di routing
  - Viene instradato attraverso l'albero logico degli elementi
  - Due possibili direzioni



# Eventi preview

- Un evento preview viene generato dall'elemento root e percorre verso il basso l'albero fino all'elemento target
  - Operazione detta “tunnel”, consente a tutti gli elementi intermedi dell'albero di filtrare o reagire all'evento



# Eventi effettivi

- Un evento effettivo viene inoltrato dall'elemento che lo ha generato all'elemento root
  - Generato dopo la ricezione di un evento “preview”
  - Operazione detta “bubble”



# Binding dei comandi

- Consente a un elemento di definire un'associazione tra un input e un comando da eseguire
  - Es: tasti di scelta rapida
- Così come il Layout, la gestione degli eventi e il binding dei comandi sono supportati dalle sottoclassi di
  - System.Windows.UIElement



# FrameworkElement

- Gli elementi che discendono da questa classe possono essere visti sotto due punti di vista:
  - Come ulteriore specializzazione di UIElement
    - Regole più stringenti per oggetti visivi con un layout consistente
  - Come base per un insieme di ulteriori sottosistemi di elementi grafici



# FrameworkElement

- Fornisce un supporto per la creazione di animazioni
  - Semplicemente specificandone le proprietà
- Introduce i concetti di data binding
  - Simile a quello dei Windows Forms
- e di stile
  - Una forma di data binding per definire l'aspetto di uno o più elementi



# Templating

- Concetto introdotto dalla classe System.Windows.Controls.Control
- Consente di stabilire in modo dichiarativo il rendering di un oggetto specificando alcuni parametri
  - Background
  - Foreground
  - Padding
  - ...



# Controlli

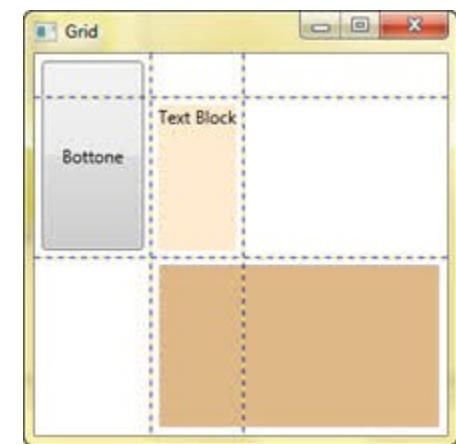
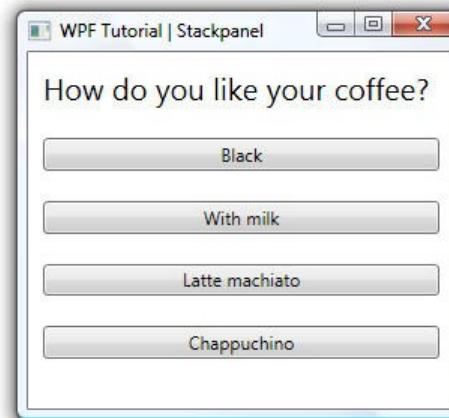
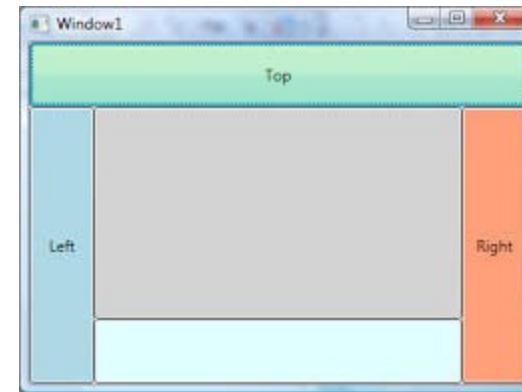
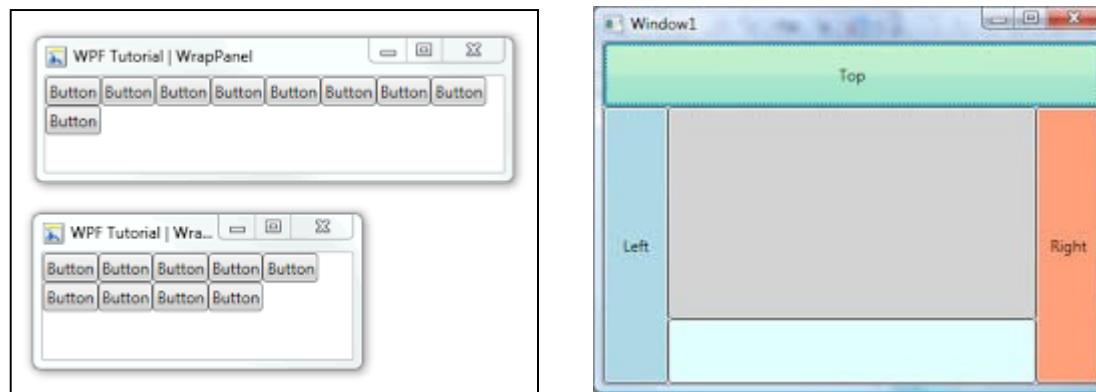
- WPF consente la personalizzazione totale del comportamento e dell'aspetto di ciascun controllo
  - Ad esempio ad un bottone si può associare un contenuto di tipo stringa...
  - ...ma anche un elemento (o albero di elementi) “disegnabile”
    - Ossia, a cui è associato un Template

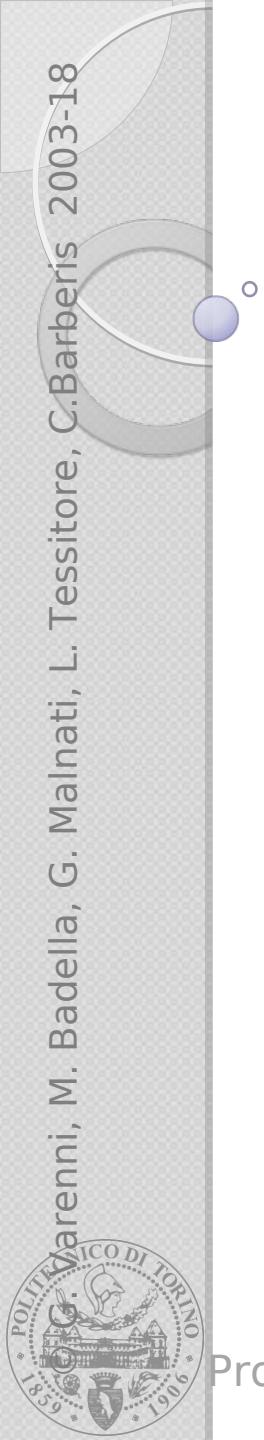
# Contenitori

- **WrapPanel**
  - Gli elementi vengono posti uno dopo l'altro, andando a capo se necessario
- **DockPanel**
  - Disposizione del contenuto seguendo un layout basato sui bordi e sul centro
- **StackPanel**
  - Lo spazio è diviso in “strisce” orizzontali o verticali: ogni componente ne occupa una
- **Canvas**
  - Posizionamento assoluto dei componenti
- **Grid**
  - Suddivisione dello spazio in celle, di dimensioni variabili



# Contenitori





# Multithreading in .NET e C#

A.A. 2017-18



# Threading in .NET

- Il namespace System.Threading contiene tutti i tipi:
  - Threads
  - Monitors
  - Eventi
  - Thread pools
  - Eccezioni relative
  - ....



# Threads

- Si crea un thread (“gestito”) attraverso la classe System.Threading.Thread
- Ogni oggetto di tale classe ingloba un thread di sistema (thread “nativo”) – NON vale il viceversa
- Due tipi di thread
  - Foreground: l’applicazione non termina finchè esiste un thread di tale tipo in esecuzione
  - Background: quando l’applicazione termina, dopo che TUTTI i thread foreground sono finiti, il runtime forza la chiusura di tali thread (con il metodo Abort() ).





# Membri della classe Thread

```
public sealed class Thread
```

```
{
```

```
 public Thread(ThreadStart start);
 public void Start();
 public void Join();
 public void Abort();
```

```
 ...
```

```
}
```

Cerca di terminare il thread lanciando una eccezione di tipo ThreadAbortException sul thread stesso.

Costruttore dell'oggetto Thread.  
Il parametro è il "metodo" che deve essere eseguito dal thread.

Crea il thread nativo vero e proprio e lo manda in esecuzione.

Si blocca, in attesa che il thread termini (indefinitamente o per un certo tempo) la propria esecuzione.

# Membri della classe Thread

```
public sealed class Thread
{
```

//proprietà

```
public IsAlive{get;}
```

```
public IsBackground{get;set;}
```

Restituisce true se il thread è in esecuzione (eventualmente in pausa)

//membri statici

```
public static Thread CurrentThread{get;}
```

```
public static void Sleep(int ms);
```

Restituisce true se si tratta di un Background thread  
Per default è false

```
}
```

Mette in pausa il thread corrente per un numero prefissato di microsecondi.  
NOTA: Sleep() effettua sempre una "yield".

Restituisce l'oggetto che rappresenta il thread corrente.



# Costruttore della classe Thread

```
public Thread(ThreadStart start);
```

```
public delegate void ThreadStart();
```

- Senza valore di ritorno
- Tale delegato può essere costruito a partire da un metodo statico o da un metodo legato all'istanza di un oggetto



# Delegato ThreadStart

```
class foo
{
 private int a;
 private int b;
 private void ThreadFcn()
 {
 int i;
 for (i = 0 ; i < this.a ; i++)
 Console.WriteLine("Number {0}", i);
 }

 public void ExecuteMyThread(int n)
 {
 a = n;
 ThreadStart MyDelegate = new ThreadStart(this.ThreadFcn);
 Thread MyThread = new Thread(MyDelegate);
 MyThread.Start();
 MyThread.Join();
 }
}
```

Output:

Number 0  
Number 1  
Number 2  
Number 3  
Number 4  
Number 5  
Number 6  
Number 7  
Number 8  
....

# Delegato ThreadStart

```
class foo
{
 private int a;
 private int b;

 private void ThreadFcnA()
 {
 int i;
 for (i = 0 ; i < this.a ; i++)
 Console.WriteLine("Number A {0}", i);
 }

 private void ThreadFcnB()
 {
 int i;
 for (i = 0 ; i < this.b ; i++)
 Console.WriteLine("Number B {0}", i);
 }

 public void ExecuteMyThread(int n)
 {
 a = n;
 b = n;
 ThreadStart MyDelegate = new ThreadStart(this.ThreadFcnA);
 MyDelegate += new ThreadStart(this.ThreadFcnB);
 Thread MyThread = new Thread(MyDelegate);
 MyThread.Start();
 MyThread.Join();
 }
}
```

Output:  
Number A 0  
Number A 1  
Number A 2  
....  
Number B 0  
Number B 1  
Number B 2  
....

Viene creato UN SOLO thread

Le due funzioni registrate sul delegato vengono eseguite SEQUENZIALMENTE



# Terminazione di un thread

- Esiste un metodo ad-hoc, Abort()
- Lancia una eccezione (ThreadAbortException) sul thread chiamato
- Il thread deve gestire questa eccezione come qualunque altra (blocchi catch e finally)
- Tale eccezione NON è ignorabile (tramite goto o codice fuori da finally) perchè viene rilanciata automaticamente dal runtime all'uscita del blocco finally (se esiste)
- L'unico modo per “fermare” l'eccezione è utilizzare il metodo ResetAbort()



# Terminazione di un thread

- Abort() è sconsigliabile perchè
  - genera una eccezione “diversa” dalle altre
- Un’eccezione è un evento “eccezionale”, la terminazione di un thread NON lo è
- Termina l’esecuzione del codice in maniera brutale
  - Cosa succede se l’eccezione è lanciata mentre il codice stava aggiornando due contatori in una sezione critica?
  - Cosa succede se si stava gestendo un’altra eccezione?
- Gestire il “recovery” di una eccezione spesso non è banale
  - Chiudere tutte le risorse, riportarsi in uno stato consistente...
- Come terminare correttamente un thread?



# Terminazione di un thread

```
class foo
{
 public volatile bool TerminateThread = false;

 public void ThreadFcn()
 {
 ...
 while (!this.TerminateThread /* && LoopCount < 1234 */)
 {
 //do something
 //update LoopCount
 //...
 }
 ...
 }
 ...
}

//other thread
fooObject.TerminateThread = true;
MyThread.Join();
...
```



# Terminazione di un thread

```
class foo
{
 public AutoResetEvent TerminateEvent = new AutoResetEvent();

 public void ThreadFcn()
 {
 WaitHandle []Handles = new WaitHandle[2];
 Handles[0] = TerminateEvent;
 //Handles[1] = some other event related to IO

 while (WaitHandle.WaitAny(Handles) == 1 /* && other */)
 { //the thread must not terminate
 //do something
 ...
 }
 ...
 }
 ...
}

//other thread
fooObject.TerminateEvent.Set();
MyThread.Join();
...
```

# Considerazioni sui thread

- In quale classe definire il metodo ThreadFcn?
  - Classe dedicata al thread
  - Classe che possiede l'oggetto Thread
- Non registrare più di un metodo sul delegato ThreadStart
- Attenzione alla gestione delle eccezioni nei thread
  - se non sono “catturate” il runtime le segnala, ma il thread principale prosegue l'esecuzione!



# Thread e applicazioni grafiche

- Si usano i thread perché l'interfaccia grafica non risulti bloccata durante una lunga elaborazione ( $>10$  ms)
  - Interrogazioni su un Database
  - Scaricamento della posta
- Tutti i metodi degli oggetti grafici DEVONO essere richiamati dal thread principale (quello che gestisce il message loop)
  - Tutte le classi derivanti da System.Windows.Forms.Control



# Thread e applicazioni grafiche

- La classe `System.Windows.Forms.Control` dispone di una serie di metodi per invocare un metodo.

```
public class Control
{
 public object Invoke(
 Delegate method,
 object []args);

 public IAsyncResult BeginInvoke(
 Delegate method,
 object []args);

 public object EndInvoke(
 IAsyncResult async);
 ...
}
```

Questo metodo riceve un delegato contenente il metodo da eseguire, e i relativi parametri.  
La chiamata è bloccante.

Come `Invoke`, ma la chiamata NON è bloccante.

Attende la terminazione di una chiamata `BeginInvoke`.

# Sincronizzazione

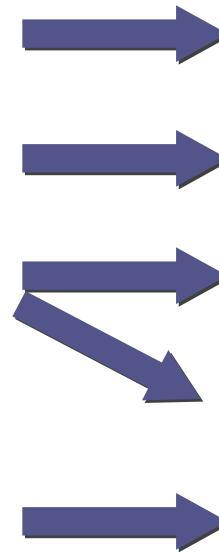
- Motivazioni
- Proteggere risorse condivise da thread differenti
  - Esempio: accesso ad un contatore
- Gestire l'interazione
  - Esempio: Produttore - consumatore



# Oggetti Win32 e .NET

## Win32

Mutex  
CriticalSection  
Event  
  
Semaphore



## .NET

Mutex  
Monitor  
AutoResetEvent  
ManualResetEvent  
Semaphore (2.0)



# Monitor

- Permette di definire una regione critica di codice (“solo un thread alla volta all’interno dello stesso processo”)

```
public sealed class Monitor
{
 private Monitor();

 public static void Enter(object obj);
 public static void Exit(object obj);

 public static void Wait(object obj);
 public static void Pulse(object obj);
 public static void PulseAll(object obj);
}
```

Non si può istanziare

“Entrata” nella regione critica

“Uscita” dalla regione critica

Condition variable associata al monitor

# Monitor

- Ricevono un generico oggetto obj
  - obj è l'entità su cui ci si sincronizza
  - Un solo thread alla volta può entrare in una sezione critica identificata da un certo oggetto obj
  - obj deve essere un reference type!
    - Se viene utilizzato un value type, il codice viene compilato, ma NON è corretto (boxing)

```
void Enter(object obj);
void Exit(object obj);
```



# Monitor - esempio

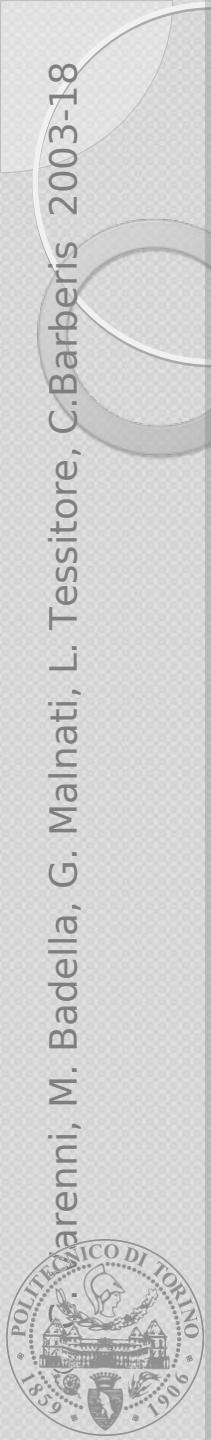
```
class foo
{
 private int a = 0;
 private int b = 0;

 public void MyMethod()
 {
 Monitor.Enter(this);
 a++;
 if (a == 12)
 b++;
 Monitor.Exit(this);
 }
}
```

```
class foo
{
 private int a = 0;
 private int b = 0;

 public void MyMethod()
 {
 try
 {
 Monitor.Enter(this);
 a++;
 if (a == 12)
 b++;
 }
 finally
 {
 Monitor.Exit(this);
 }
 }
}
```





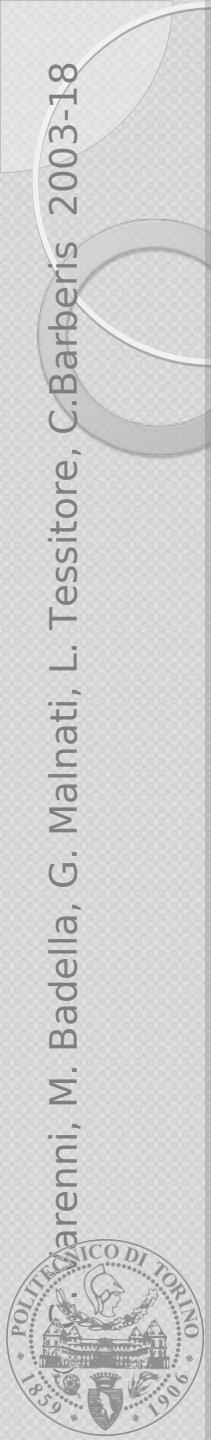
# Costrutto C# lock()

- Si utilizza per creare una sezione critica all'interno del codice C#
- Sintassi

```
lock(object obj)
{
 //code
}
```

```
class foo
{
 private int a = 0;
 private int b = 0;

 public void MyMethod()
 {
 lock(this)
 {
 a++;
 if (a == 12)
 b++;
 }
 }
}
```



# Costrutto C# lock() - 2

- Si tratta di “syntactic sugar”
- Viene sostituito dal compilatore C# con chiamate a Monitor.Enter(obj)/Exit(obj), inglobate in un blocco try/finally
- Valgono le considerazioni sui Monitor

```
lock(obj)
{
 //code
}
```



```
try
{
 Monitor.Enter(obj);
 //code
}
finally
{
 Monitor.Exit(obj);
}
```

# Gerarchia WaitHandle

- Oggetti di sincronizzazione
  - Molto simili agli HANDLE Win32
- Due possibili stati: segnalato e non segnalato

```
public abstract class WaitHandle
{
 public virtual IntPtr Handle{get;set;};

 public virtual bool WaitOne();

 public static bool WaitAll(WaitHandle []handles);
 public static int WaitAny(WaitHandle []handles);
}
```

Proprietà che rappresenta l'handle nativo Win32 (HANDLE) dell'oggetto

Chiamata bloccante, aspetta che l'oggetto sia segnalato

Chiamate bloccanti, aspettano che almeno uno (o tutti) gli oggetti siano segnalati.

I metodi Wait sono analoghi alle chiamate WaitForSingleObject() WaitForMultipleObjects()

# Gerarchia WaitHandle

- **Mutex**
  - Wrapper del Mutex Win32
  - HANDLE hMutex = CreateMutex(...)
- **ManualResetEvent**
  - Wrapper dell'evento Win32 con reset manuale
  - HANDLE hEvent = CreateEvent(..., TRUE, ...)
- **AutoResetEvent**
  - Wrapper dell'evento Win32 con reset automatico
  - HANDLE hEvent = CreateEvent(..., FALSE, ...)



# Mutex

- Si utilizza per creare sezioni critiche di codice (“un thread alla volta”)

```
public sealed class Mutex : WaitHandle
{
 ...
 public override bool WaitOne();
 public void ReleaseMutex();
 ...
}
```

Utilizzato per acquisire il mutex

Utilizzato per rilasciare il mutex

- È consigliabile usare un blocco try/finally
- Un Mutex acquisito NON è segnalato, un Mutex rilasciato è segnalato

# Mutex - Esempio

```
class foo
{
 private int a = 0;
 private int b = 0;
 private Mutex MyMutex = new Mutex();

 public void MyMethod()
 {
 try
 {
 MyMutex.WaitOne();
 a++;
 if (a == 12)
 b++;
 }
 finally
 {
 MyMutex.ReleaseMutex();
 }
 }
}
```

Creo un oggetto Mutex.

NOTA: questo è “syntactic sugar”, questo codice è inserito nel costruttore dal compilatore C#

Acquisisco il Mutex all'interno del blocco try

Rilascio il Mutex all'interno del blocco finally

# Mutex – Note

- All'interno dello stesso thread si può acquisire uno stesso Mutex più volte, a patto di rilasciarlo lo stesso numero di volte
- Non si può rilasciare un Mutex acquisito da un altro thread (ApplicationException)
- Un Mutex NON rilasciato al termine di un thread viene rilasciato automaticamente dal runtime

# Mutex e Monitor

- Analogie
  - Si utilizzano per creare sezioni critiche
- Differenze
  - Un Monitor si sincronizza su un oggetto
    - Consigliabile per sincronizzazione intra-oggetto
  - Un Mutex è un oggetto a sé stante
    - Consigliabile per sincronizzazione tra oggetti
  - Un Mutex può essere utilizzato per la sincronizzazione tra processi



# Eventi

- Sono oggetti che possono essere settati (segnalati) e resettati (non segnalati)

```
public sealed class ManualResetEvent : WaitHandle
public sealed class AutoResetEvent : WaitHandle
{
 ...
 public override bool WaitOne();
 public void Set();
 public void Reset();
 ...
}
```



# ManualResetEvent

- Caratteristiche

- Devono essere resettati (non segnalati) manualmente
- Tutti i thread in attesa (Wait) vengono svegliati non appena l'evento viene segnalato

- Utilizzi

- Si usano per notificare più thread di un certo evento
- NotificationEvent nel kernel di Windows
- Esempio: far partire più thread simili “nello stesso istante”



# AutoResetEvent

- Caratteristiche

- Il primo thread in attesa sulla Waiting Queue viene svegliato, e l'evento viene resettato automaticamente
- Se nessun thread è in attesa sull'evento, questo rimane segnalato
- Molto simile a un semaforo a 1 posizione

- Utilizzi

- Sincronizzazione vera e propria tra thread
- SynchronizationEvent nel kernel di Windows
- Esempio: problema del produttore-consumatore



# Altri classi relative al threading

- **System.Threading.ThreadPool**
  - Permette l'accesso al thread pool nativo attraverso i propri metodi statici
    - QueueUserWorkItem
  - Si ottengono fuzionalità simili utilizzando il costrutto BeginInvoke() su un delegato
- **System.Threading.ReaderWriterLock**
  - Si usa per implementare il pattern dei Readers e Writers
  - Si tratta di un insieme di mutex e/o eventi inglobati in una classe



# Altre classi relative al threading

- System.Threading.Interlocked
  - Contiene una serie di metodi statici per effettuare operazioni matematiche atomiche sugli interi
  - Incremento, decremento, test-and-set, scambio
  - Analoghe ai metodi InterlockedXXX dell'API Win32
- System.Threading.Timer
  - Richiama un metodo (tramite delegato) ad un istante predefinito
  - Utilizza un thread del ThreadPool (e il delegato è quindi chiamato in un thread diverso dal thread principale)

