##################
Domande senza data trovate nei vari pdf
##################

1-Indicare le differenze tra dynamic_cast<T> e static_cast<T> in C++

Lo static_cast<T>(p) è un template C++11 che converte il valore di "p" rendendolo di tipo "T" qualora esista un meccanismo di conversione disponibile noto al compilatore. La conversione avviene in modo statico in fase di compilazione, quindi non vi è nessun tipo di controllo di compatibilità a run-time. In presenza di conversione illecita il risultato è impredicibile. Il vantaggio di static_cast è la sua efficienza visto che ha un impatto in termini di esecuzione molto limitato ed il suo uso principale è per fare movimenti in verticale lungo l'asse ereditario, in particolare dalla classe Figlia a quella Padre (upcast) ossia "Padre * obj = static_cast<Padre*>(objFiglia)". Questa istruzione funziona grazie alla presenza delle V-Table. Nella Fig.1 "b1" e "d" hanno lo stesso indirizzo, solo che il compilatore ha informazioni diverse proprio per la presenza delle V-Table, una per "d", una per "b1" ed una per "b2". Nella Fig.2 "b2" non ha lo stesso indirizzo di "d", ma un indirizzo pari a "d+offset" così che "b2" possa puntare alla sua V-Table. Il compilatore del C avrebbe assegnato a "b2" lo stesso indirizzo di "d", con l'ovvia conseguenza di ottenere la V-Table di "b1" e non quella di "b2" comportando l'esecuzione di codice diverso da quello voluto.

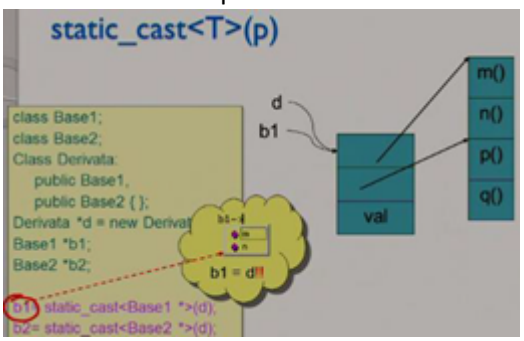


Fig1

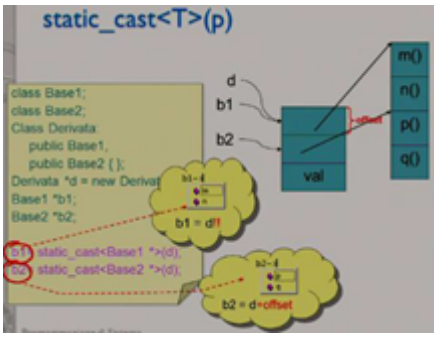


Fig2

Il compilatore permette di utilizzare static_cast anche per scendere nell'ereditarietà (downcast), ma è altamente sconsigliato visto che nella Fig.3 il compilatore penserà di avere tutti i campi da "m() a q()", invece vedrà solo i campi "m() e n()", quindi in fase di esecuzione sorgono molti problemi se si utilizzano i metodi virtuali "p() e q()".

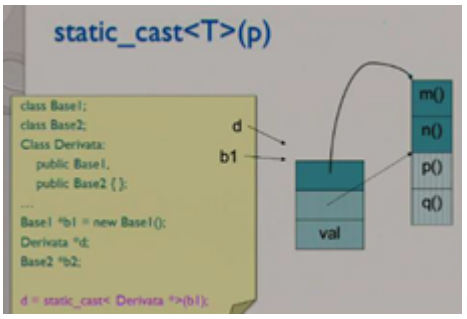


Fig3

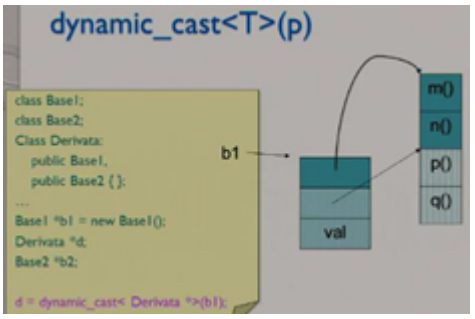


Fig4

Il dynamic_cast<T>(p) è un template C++11 che fa più o meno le stesse cose di static_cast, ma in più effettua il controllo a run-time dei tipi, il che permette di scendere oltre che salire nell'ereditarietà. In caso di discesa in presenza di un puntatore, entra in gioco il RTTI (RunTime Type Identification) il quale verifica se l'oggetto di partenza non è compatibile con quello di arrivo, in tale caso il dynamic_cast restituisce "null" evitando così di accedere alla memori altrui. In presenza di un riferimento, il quale non può mai valere "null" per definizione, se il tentativo di downcast è incompatibile viene sollevata un'eccezione. In Fig.4 il dynamic_cast può effettuare il downcast da una classe base virtuale ad una derivata, anche se l'istruzione "d = dynamic_cast<Derivata*>(b1)" restituisce "null" grazie all'analisi della V-Table, infatti il meccanismo di RTTI verifica se la discesa lungo l'asse verticale porta ad avere una V-Table identica a quella della classe "Derivata". Questo

controllo da parte del RTTI avviene con un semplice "if", quindi rispetto allo static_cast, il dynamic_cast ha un costo di esecuzione maggiore. Identico discorso vale per "d = dynamic_cast<Derivata*>(b2)".

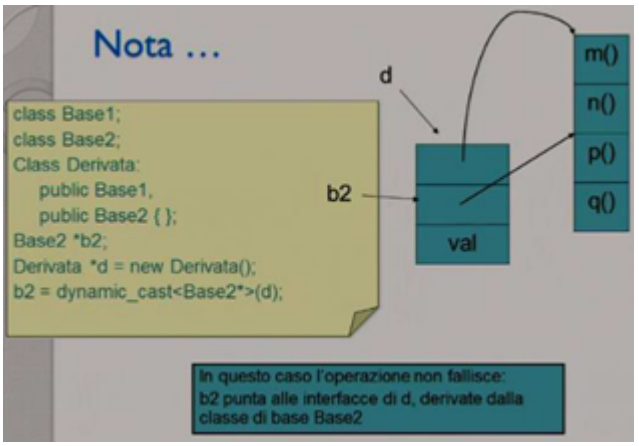


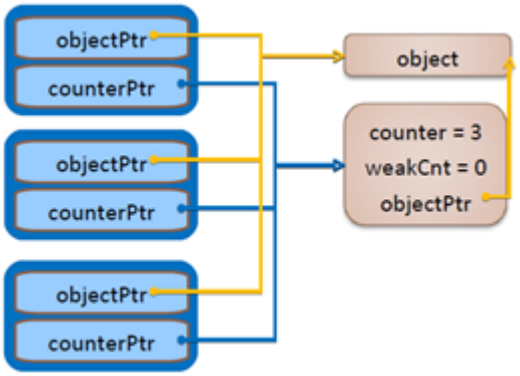
Fig5

Nella Fig.5 si effettua un'operazione di upcast "b2=dynamic_cast<Base2*>(d)", come in static_cast, la quale è semanticamente lecita visto che Derivata <<is_a>> Base2 e nell'indirizzo di "b2" viene messo "d+offset", quindi sarà possibile accedere ai metodi "p() e q()". Qualora si effettua l'operazione di upcast "b1=dynamic_cast<Base1*>(d)" si potrà accedere ai metodi "m() e n()". L'operazione di "upcast" effettuata con dynamic_cast ha un costo di esecuzione maggiore rispetto alla static_cast proprio per la presenza del RTTI, il quale a sua volta rispetto alla Reflection di Java/C# ha costi di esecuzione nettamente minori

Riassumendo in modo sintetico possiamo rispondere alla domanda nel seguente modo:
Lo static_cast<T>(p) è un template C++11 che converte il valore di tipo "p" nel tipo "T" qualora il compilatore conosca un meccanismo di conversione. Il controllo di compatibilità è statico, quindi non vi è nessun meccanismo a run-time per la verifica della compatibilità, ma ci si appoggia esclusivamente al meccanismo di conversione presente. Lo static_cast viene utilizzato per le conversioni lungo l'asse verticale ereditario o quando sono presenti gli operatori di conversione esplicita come ad esempio un costruttore esplicito nella classe destinazione che accetta un parametro della classe sorgente oppure se nella classe sorgente è presente un metodo esplicito che genera un valore della classe di destinazione. Tutti questi meccanismi sono statici, quindi con le sole informazioni note a compile-time. In caso di conversioni illecite il risultato è impredicibile.
Lo static_cast è più idoneo per movimenti lungo l'asse verticale ereditario di tipo "upcast".
Il dynamic_cast<T>(p) si comporta più o meno come static_cast<T>(p), solo che effettua il controllo di compatibilità a run-time sfruttando il RTTI (Runtime Type Information) ed in caso di incompatibilità genera null in presenza di puntatori, oppure solleva un eccezione in presenza di riferimenti.
Il dynamic_cast, appoggiandosi a RTTI, ha un costo di esecuzione maggiore rispetto static_cast, quindi è sconsigliabile utilizzarlo per un movimento verticale di tipo "upcast", ma conviene usarlo esclusivamente nel "downcast"

2-Differenza tra std::shared_ptr, std::unique_ptr, std::weak_ptr e std::auto_ptr
Questi strumenti sono dei template introdotti dal C++11 per la gestione automatica dei puntatori nativi, di conseguenza della memoria.
Uno "shared_ptr" serve per referenziare un blocco di memoria tramite un costruttore, definendo sul blocco di memoria una struttura di controllo, con la quale è possibile sapere quanti riferimenti effettivi puntano direttamente a tale blocco. Il conteggio dei riferimenti permette al distruttore, invocato automaticamente dal compilatore quando l'oggetto di tipo "shared_ptr" esce dal suo scope, di eliminare il blocco di memoria se e solo se il conteggio è pari a 0, ossia il blocco non è più referenziato da nessuno. In questo modo vi è una gestione automatica del rilascio della memoria

evitando "Memory Leakage" ed ambiguità nell'uso del puntatore. Un primo svantaggio di "shared_ptr" è la necessità per ogni blocco di memoria di mantenere anche la struttura di controllo con i conteggi, che ne causa un'occupazione almeno doppia rispetto ad un puntatore nativo del C. Un secondo svantaggio/limite di "shared_ptr" è la gestione di dipendenze cicliche, ossia oggetti che dipendono l'uno dall'altro che causano l'impossibilità di portare il conteggio a zero nel momento in cui gli oggetti escono dal proprio scope, con l'ovvia conseguenza che il blocco di memoria resta allocato.



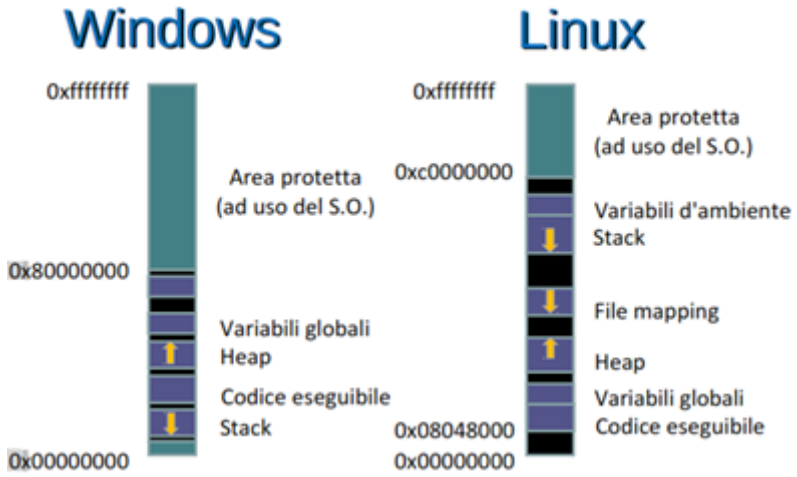
Uno "unique_ptr" serve per referenziare in modo esclusivo un blocco di memoria, per poi rilasciarlo in modo automatico dal parte del distruttore quando viene meno lo validità dell'oggetto nel proprio scope. A differenza di "shared_ptr", qui non vi possono essere più puntatori alla medesima risorsa, di conseguenza non si fa uso di nessuna struttura di controllo per i conteggi. A causa della esclusività del blocco puntato, uno "unique_ptr" non può essere nè copiato nè assegnato (si avrebbe un errore di compilazione), ma solamente mosso con la funzione "std::move()". L'uso di "unique_ptr" garantisce sempre la distruzione di un oggetto.
Uno "weak_ptr" è un oggetto che permette di accedere ad un blocco referenziato da uno "shared_ptr" senza causare l'aumento del conteggio dei riferimenti, quindi questa interazione "debole" non può mai impedire che il conteggio arrivi a 0 con conseguente rilascio del blocco di memoria. Lo "weak_ptr" permette anche di richiedere una nuova copia dello "shared_ptr" sempre che il blocco di memoria sia ancora valido.
L'uso più fine di "weak_ptr" è in presenza di riferimenti ciclici, che con "shared_ptr" causa l'impossibilità di azzerare il conteggio. In tali situazioni si utilizza un "weak_ptr" che permette di impostare un riferimento "debole" tra due o più "shared_ptr". La struttura di controllo dello "shared_ptr" mantiene un conteggio dei "weak_ptr" e tramite il metodo “lock()” è possibile accedere al dato che restituisce uno “shared_ptr”.
Uno “auto_ptr” è un oggetto molto simile a “unique_ptr”, ma cancella la sorgente, quindi è vivamente sconsigliato il suo impiego. Veniva usato prima della versione C++ 2011.

3-Descrivere la sintassi di shared_ptr e i suoi metodi
La sintassi è shared_ptr<T>(T* nat_ptr) e permette la costruzione di uno shared_ptr incapsulando il puntatore nativo "nat_ptr" di tipo T.
I metodi più utilizzati sono:
• "make_shared<BaseType>(params)” che crea e restituisce un oggetto “shared_ptr” tramite una sola chiamata “new”, chiedendo più memoria di quella necessaria col fine di inserire all’interno la struttura di controllo ed ovviamente l’oggetto “BaseType”. Questa strategia fa si che il ciclo di vita dell’oggetto “BaseType” e della struttura di controllo, siano identici così che con una una sola delete si possa eliminare tutto
• "operator=(const share_ptr<T>& other)" permette l'assegnazione del puntatore condiviso aumentando di uno il numero di riferimenti al blocco nella struttura di controllo
• "get(), operator*(), operator->()" sono metodi che permettono l'accesso al puntatore incapsulato

• "reset()" elimina il riferimento al blocco puntato decrementando di uno, nella struttura di controllo, il numero di riferimenti. Qualora il numero di riferimenti è 0, il blocco di memoria viene rimosso dallo heap
• "reset(BaseType* ptr)" come il precedente solo che sostituisce il puntatore con quello passato come parametro
• "~shared_ptr<T>()" distrugge lo shared_ptr e libera la memoria qualora il numero di riferimenti nella struttura di controllo è pari a 0

4-Cosa si intende per ambiguità dei puntatori? Cosa è il Dangling Pointer e il Memory Leakage?
Un puntatore in C non permette di identificare chi lo possiede e quindi la de-allocazione del blocco puntato non è esclusa da errori. Non è certa la dimensione del blocco puntato, se è possibile modificarlo senza problemi e quando il programmatore lo può rilasciare. E' tutto sullo spalle del programmatore la gestione di queste problematiche. Il Memory Leakage è la continua allocazione di memoria dinamica senza nessun rilascio. Può portare alla saturazione dello HEAP. Il Dangling Pointer è un puntatore che dopo aver terminato il suo ciclo di vita può contenere un indirizzo di memoria impiegato da un altro programma. L'impiego di tale puntatore può portare ad effetti impredicibili.

5-Layout della memoria di un'applicazione in Windows e Linux



6-Spiegare la differenza tra variabili locali, globali e dinamiche
Le variabili locali vengono allocate con un indirizzo relativo in cima allo stack e cessano di esistere al termine del blocco di definizione. Le variabili globali hanno un indirizzo fisso determinato dal compilatore e dal linker. Le variabili globali vengono inizializzate all'avvio di un programma. Le variabili dinamiche hanno un indirizzo assoluto definite in fase di esecuzione, accessibili tramite puntatori. Il controllo delle variabili dinamiche è lasciato al programmatore.

7-Spiegare cosa comporta l'esecuzione di un processo
L'esecuzione di un processo comporta la creazione dello spazio di indirizzamento, il caricamento dell'eseguibile in memoria, il caricamento delle librerie e l'esecuzione del processo.

8-Cosa avviene all'esecuzione di un programma?
Avviene la configurazione della piattaforma, ossia l'inizializzazione dello stack, dei registri e delle strutture dati per la gestione delle eccezioni. Si invoca il costruttore degli oggetti globali e viene chiamata la funzione "main". Al termine del "main" si chiama il distruttore ed il processo rilascia lo spazio di indirizzamento e tutte le risorse allocate sfruttando una system call "exit()".

9-Cosa sono le eccezioni? Quale è la strategia di gestione?
Il meccanismo delle eccezioni serve per modificare il normale flusso di esecuzione del programma utilizzando la clausola "throw", dopo di che il chiamante dovrà gestire il tipo di eccezione indicata con la "throw" tramite un costrutto "try-catch". Nel blocco "catch" è necessario specificare l'eccezione che si vuole gestire, ed è possibile elencare più blocchi "catch" relativi ad eccezioni diverse. Se nessuno dei blocchi "catch" definisce il tipo di eccezione che si è verificata, lo stack si contrae e l'eccezione viene passata al chiamante.
Un blocco "catch" ha il compito di reagire, cercando di fare rientrare l'eccezione o, al massimo, terminare in modo ordinato il programma producendo un file di log con i messaggi.
La chiamata di un'eccezione ha un costo di un ordine superiore rispetto alla chiamata di una funzione.

```
int retry=2;
while(retry) {
try {
        <istruzione>
        break;
}
catch(exception& e) {
        retry--;
        if(!retry)
                throw;
        }
}
```

10-Quali sono le azioni fatte dal compilatore C++?
Il compilatore crea le variabili globali prima dell'avvio del programma e le distrugge quando termina. Le variabili locali sono costruite all'ingresso di un blocco e distrutte alla sua uscita. Le variabili dinamiche sono costruite e distrutte in modo esplicito dal programmatore. Il costruttore e distruttore vengono invocati al procedere del ciclo di vita di un oggetto.

11-Descrivere la semantica e la sintassi di una funzione lambda
Il paradigma di "Programmazione Funzionale" permettere di impostare il flusso del codice di un programma con l'ausilio di funzioni, che in ambito C++ trova impiegato con le funzioni Lambda.
La sintassi completa di una funzione lambda è la seguente:
[capture] (params) mutable exception attribute -> return-type { body; }
Le semantica della funzione lambda è la seguente:
return-type = il tipo di ritorno della funzione lambda che è opzionale se il compilatore riesce a dedurne
il tipo
body = il codice che esegue la funzione lambda
params = elenco dei parametri
capture = modalità con cui le variabili locali vengono passate alla funzione lambda, ossia per valore o per riferimento
[ ] non si cattura nulla
[a,&b] si cattura la variabile locale "a" per valore e quella "b" per riferimento
[this] si cattura il puntatore "this"
[&] si cattura tutto per riferimento
[=] si cattura tutto per valore
mutable = permette di modificare i parametri catturati dalla copia
exception = specifica un'eccezione oppure la clausola noexception per operator()
attribute = specifica gli attributi per operator()

12-Spiegare che cosa si intende per RAII?
RAII (Resource Acquisition Is Initialization) è un pattern di programmazione che sfrutta la contrazione dello stack per rilasciare le risorse in precedenza acquisite nello stack. Viene utilizzato in presenza delle eccezioni o anche in presenza del lock_guard() per il quale il mutex viene allocato dal costruttore ed eliminato dal distruttore sfruttando la contrazione dello stack.

13-Che cosa è la regola dei tre?
La regola dei tre stabilisce che quando in una classe si implementa una funzione membro quale l'operatore di assegnazione, il distruttore o il costruttore di copia, allora è necessario implementare anche le restanti due. In mancanza di una definizione esplicita da parte del programmatore, il compilatore C++ utilizzerà una propria implementazione che quasi sempre non è adatta al programma creato.

14-Che cosa è il "Copy&Swap"?
Il "Copy&Swap" è un pattern di programmazione che permette di evitare di sottostare alla "regola dei 5", sfruttando la funzione std::swap, implementata all'interno di una funzione friend, per scambiare il contenuto delle risorse tra due istanze di classe.
Una funzione definita come friend non appartiene alla classe, ma ha diritto di accedere ai campi "private". La"Copy&Swap" è un pattern più sicuro che evita fonti di problemi.

15-Spiegare cosa è la V-Table e a cosa serve
La Virtual Table V-Table è una tabella di lookup che permette di implementare il late binding. Ogni campo della V-Table contiene un puntatore al metodo virtuale, quindi il numero delle entry di tale tabella indica il numero di metodi virtuali. In assenza della V-Table e con classi figlio di tipo classe padre, il compilatore potrebbe invocare il metodo sbagliato. Quando viene creato l'oggetto, viene inserito un campo nascosto "V-Table ptr" che punta alla V-Table, quindi tale puntatore è un overhead. La V-Table è shared per oggetti della stessa classe. Oggetti di classi diverse hanno V-Table differenti.

16-Che cosa sono i metodi dichiarati "virtual"?
I metodi virtuali permettono nel linguaggio C++ l'utilizzo del polimorfismo. Nel momento in cui ai metodi virtuali viene assegnato lo 0, si ottengono metodi virtuali astratti.

17-Spiegare la differenza tra puntatore a funzione e oggetto funzionale
Un puntatore a funzione è una variabile che memorizza l'indirizzo di una funzione e serve per rendere gli algoritmi dinamici, permettendo l'applicazione dall'esterno di funzioni agli elementi.
La sintassi di un puntatore a funzione è la seguente:
<ret> (*var) (<argomenti>)
"var" è una variabile di tipo puntatore a funzione , "ret" il tipo di ritorno, mentre "argomenti" sono i parametri formali. E' compito del compilatore verificare la signature di "var".
Un oggetto funzionale, chiamato anche funtore, è un istanza di classe che implementa la ridefinizione dell'operatore (), tramite "operator()", permettendo di definire una classe funzionale le cui istanze vengono utilizzate come funzioni con stato.

```
class MyClass {
public:
	int operator() (int v) {
		return v*2;
	}
}
...
MyClass obj;
```

int i=obj(5); // uso obj come funzione ed "i" vale 10

18-Spiegare il concetto di "Programmazione Generica" e template

Linguaggi di programmazione fortemente tipizzati permettono la scrittura di codice robusto, basandosi sul concetto di sistema cogente nel quale in fase di compilazione si cercano gli errori sui tipi. Qualora si voglia modificare per una porzione di codice il tipo, è necessario replicarlo apportando le dovute modifiche, con l'ovvia conseguenza che la manutenzione del programma può divenire molto complicata. Nella vita reale i problemi che si presentano o hanno livelli di astrazioni più complessi, di conseguenza anche il dominio dei dati può non essere predicibile in fase di scrittura del codice. Per ovviare a questo problema, il C++ mette a disposizione i "template" che permettono la scrittura di codice generico, ossia non legato a nessun tipo, così che il compilatore non generi nessun codice legato al template stesso, ma solo quando viene passato il tipo verrà creata un'apposita implementazione.
La compilazione avviene quindi in due fase distinte.
1. Analisi sintattica del blocco template
2. Sostituzione dei tipi concreti nei placeholder <T> con conseguente generazione del codice a compile-time

Con la programmazione generica, e quindi i template, è possibile scrivere un'unica funzione che possa venire utilizzata con una varietà di tipi di dati (es: funzione minimo).
Il template può essere applicato alle funzioni generiche o alle classi generiche.

```
template <class T> const T& max(const T& t1, const T& t2) { return (t1<t2 ? t2 : t1); }
```

La classe che realizza questa funzione deve avere implementato l'overloading dell'operatore "<", e sarà compito del compilatore assegnare il giusto tipo a T.

```
int i = max(10,20);
string s = max("pippo","casa");
```

Una classe generica deve prevedere la solita clausola "template <class T>".

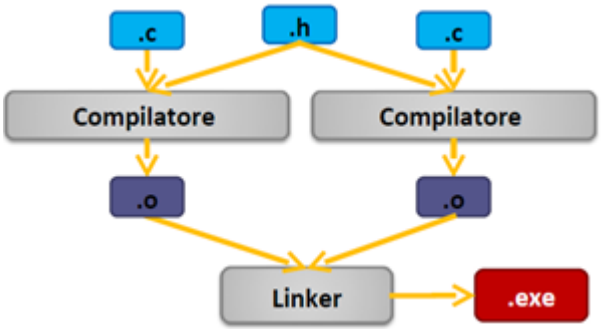
```
template <class T>
class MyClass {
        T size;
public:
        MyClass(T val) : size(val) { }
        T leggiSize() { return size; }
}
```

Il C++ effettua un'ottimizzazione del codice sulla base del tipo di dato che verrà utilizzato al posto di T, mentre i linguaggi Java/C# sfruttano la logica del "Type Erasure" ossia ci si appoggia ad una struttura gerarchica con a capo Object, dove ad una lista di stringhe viene associata una lista di Object, per poi procedere all'operazione di casting. Questo modo di gestione peggiora di molto le prestazioni. Il template C++ non genera nuovamente il codice se questo è già presente, ma si limita solamente a richiamarlo.
I vantaggi del template sono:
risparmio di tempo nello sviluppo del programma
non viene compromesso il sistema dei tipi, visto che viene fatto in 2 fasi
non si pregiudicano i tempi di esecuzione
maggiore flessibilità in caso di miglioramenti del codice

19-Descrivere il processo di compilazione di un compilatore C

Il software realizzato può essere diviso in moduli ".c", ognuno compilato separatamente col fine di produrre un file oggetto contenente il codice macchina relativo all'architettura utilizzata e contenente riferimenti pendenti alle funzioni esterne e alle variabili esterne. I moduli ".c" utilizzano delle librerie header file ".h", che contengono una lista di simboli, questi simboli sono visti come

funzioni esterne. Tutti questi file oggetto vengono poi uniti in un unico file eseguibile da parte del linker, il quale risolve i simboli esterni inserendo gli opportuni indirizzi.



20-Spiegare come avviene il collegamento statico ed il collegamento dinamico
Una libreria può essere caricata all'avvio in 3 modi distinti. Il primo modo è durante la fase di linking, in cui il linker carica staticamente la libreria nell'eseguibile. Si parla quindi di collegamento statico.
Il secondo modo è durante il caricamento del programma in memoria da parte del loader, tipico uso in Windows ed il programma per funzionare ha bisogno di API le quali a sua volta wrappano le system call. Queste API si trovano in DLL le quali sono il più delle volte già presenti in memoria e se non lo sono vengono caricate. E' il loader che risolve gli indirizzi pendenti facendo un mapping tra lo spazio di indirizzamento del processo e gli indirizzi fisici delle librerie dinamiche. Il linux le DLL sono chiamate "shared object" ed hanno estensione ".so".
Il terzo modo per il caricamento delle librerie è durante l'esecuzione del programma stesso. In tale ottica il programma è costruito specificamente sapendo che alcune parti del codice non saranno in suo possesso e quando avrà necessità dovrà mappare delle DLL nel proprio spazio di indirizzamento nel tentativo di risolvere i simboli. Per fare questo si utilizza la "LoadLibrary".
Riassumendo, le librerie possono essere statiche, dinamiche o condivise.
Una libreria statica è un'unità di compilazione dentro un file ".lib" in Windows o ".a" in Unix che il linker inserisce all'interno di un eseguibile. Quest'ultimo sarà velocissimo in fase di startup, ma in caso di esecuzione di più copie dell'eseguibile, vi saranno più copie dei moduli in memoria.
Una libreria collegata dinamicamente è un file ".DLL" o ".so" che il loader carica all'avvio dell'esecuzione dell'applicazione. Quando voglio creare un programma plug-in aggiungo nel codice quello che serve per andare a caricare esplicitamente il codice, in tale caso si parla di caricamento dinamico realizzato tramite la funzione "LoadLibrary".

21-Definizione di programmazione funzionale
La programmazione funzionale è un paradigma che si basa sul cardine fondamentale che per strutturare i programmi è necessario utilizzare delle funzioni, mentre il controllo di esecuzione del programma avviene per mezzo dell'applicazione. Il programmatore dovrà costruire delle funzioni, espresse in termini matematici, con l'ausilio di un certo numero di funzioni ausiliarie, col fine di risolvere un dato problema. Il calcolatore dovrà valutare le espressioni e stampare il risultato.

22-Spiegare la differenza tra le API e ABI
Le API (Application Programming Interface) sono un insieme di funzioni e strutture dati necessarie per accedere ai livelli sottostanti, mentre le ABI (Application Binary Interface) definiscono il formato che deve avere un software per essere compatibile con un O.S.

23-Spiegare come Windows e Linux gestiscono gli errori ad una chiamata API
In Windows è necessario basarsi sul tipo di ritorno adottando una precisa strategia di gestione per poi chiamare il prima possibile la funzione "GetLastError()" che identifica il tipo di errore. Tale funzione restituisce una DWORD su 32bit.

In Linux l'errore di chiamata di una API restituisce quasi sempre il valore "-1" per poi accedere al codice di errore controllando il valore della pseudo-variabile globale "#define errno(*_errno_location)".

24-Spiegare la differenza tra std::future<T> e std::shared_future<T>
Entrambe permettono di recuperare il risultato di un blocco di codice asincrono garantendo il thread-safe, ma mentre la future<T> mette a disposizione la "get" per recuperare il dato una ed una sola volta, la shared_future<T> permette di lavorare con il risultato in più punti del programma costruendo delle vere e proprie catene di elaborazione asincrone che offrono il massimo livello di parallelismo. Per fare questo è necessario utilizzare il metodo "share" in fase di definizione della std::async<T>. La std::future<T> non è mai copiabile, ma solo movibile, mentre la shared_future<T> è sia copiabile che movibile.
Entrambe dispongono di un metodo "wait" con il quale si attende la terminazione del task senza però prelevare il risultato. Qualora il task non sia ancora stato avviato ne forza l'avvio.

25-Spiegare le differenze tra std::future<T> e std::promise<T>
La std::future<T> è usata per operazioni asincrone indipendentemenri di tipo "producer/writer" dove il risultato prodotto all'interno del thread viene salvato in una variabile condivisa nascosta, di conseguenza non va gestita esplicitamente. Questo meccanismo di sincronizzazione è di alto livello, unidirezionale, thread-safe e multipiattaforma.
La std::promise<T> è usata per operazioni asincrone di tipo "consumer/reader" nella quale però il risultato prodotto dal thread va memorizzato in una variabile condivisa passata come riferimento, quindi gestita in modo esplicito tra thread principale e thread secondario.
Un oggetto std::promise<T> ingloba una std::future<T>, quindi anch'essa unidirezionale e thread-safe.
La std::future<T> ritorna l'eccezione al chiamante, mentre la std::promise<T> deve implementare nel thread un blocco try-catch per la sua gestione.

26-Spiegare a cosa serve la std::promise<T>
L'oggetto std::promise<T> permette ad un thread di restituire un valore calcolato all'interno del suo blocco di esecuzione, in caso di impossibilità notificherà un'eccezione. La std::promise<T> permette l'accesso al risultato, inserendo un meccanismo di sincronizzazione unidirezionale, basato sulla conoscenza del future della p romise, infatti finché la "get" non darà un risultato, il flusso logico non proseguirà la sua esecuzione. Possono esserci molti cicli macchina dal momento in cui viene settata la std::promise<T> al momento in cui viene chiamato il distruttore, con ovvi problemi di corse critiche, quindi conviene settare il valore della promise in fase di uscita attraverso il metodo "set_value_at_thread_exit()" o, in caso di eccezione, tramite la "set_exception_at_thread_exit()".

27-Spiegare come funziona un mutex in C++ e quali sono i suoi metodi
Un oggetto std::mutex permette l'accesso ad un blocco di codice critico ad un solo thread alla volta, per tale motivo l'oggetto std::mutex deve essere condiviso tra tutti i thread, ma anche il codice presente all'interno della regione critica. Per delimitare una regione critica, il mutex mette a disposizione due metodi, "lock()" e "unlock()". Il metodo "lock()", quando invocato, effettua un'invalidazione della cache, così che il contenuto della regione critica sia consistente e bloccante, ossia qualora il mutex sia già in possesso di un thread, altri thread che chiamano la "lock()" resteranno bloccati senza poter entrare nella regione critica. Il metodo "unlock()" fa un "refresh cache" rimettendo in cache il dato precedentemente cancellato, inoltre sblocca uno dei thread in attesa. Tale chiamata è obbligatoria altrimenti nessun thread in attesa potrebbe mai entrare nella regione critica.
mutex.lock();
<Regione Critica>

mutex.unlock();
Una corretta implementazione delle chiamate "lock()/unlock()" deve avvenire all'interno di metodi pubblici, mentre i dati condivisi e mutex devono essere private, in questo modo si hanno metodi thread-safe. L'impiego di mutex ha protezione di regioni critiche comporta una riduzione del grado di parallelismo, inoltre l'abbinamento dato da proteggere e mutex è solamente nella testa di chi scrive il codice. I mutex non sono ricorsivi, quindi due chiamate consecutive di "lock()", senza nessun "unlock()", genera un blocco infinito del thread. E' importante evidenziare che un mutex va sempre rilasciato anche se si verifica un'eccezione. Un altro metodo di mutex è "try_lock" il quale restituisce false qualora il thread non possieda il mutex, true in caso contrario.
while(mutex.try_lock() == false) { fai_altro(); }
lock_guard<mutex> lg(mutex, adopt_lock);

28-Spiegare le differenze tra recursive_mutex e timed_mutex
La recursive_mutex permette ad un thread di acquisirla N volte consecutivamente, ciò comporta che il rilascio della recursive_mutex dovrà essere pari a N, prima che un altro thread possa acquisirla. L'occupazione di memoria è maggiore rispetto al semplice mutex.
La timed_mutex permette di porre un limite temporale massimo per l'attesa di acquisizione del mutex tramite il metodo "try_lock_for()", mentre è possibile impostare un tempo di scadenza dopo il quale si restituisce false se il mutex non è acquisito utilizzando il metodo "try_lock_until()".
L'unione della recursive_mutex e timed_mutex da vita alla recursive_timed_mutex che possiede le caratteristiche di entrambe le classi.

29-Spiegare la differenza tra lock_guard() e unique_lock()
Quando si lavora con un oggetto std::mutex<T> è necessario chiamare esplicitamente i metodi "lock()/unlock()" e gestire l'eccezione. Per fare tutto questo in modo automatico, il C++ mette a disposizione una oggetto generico chiamato std::lock_guard<Lockable> il quale sfrutta il proprio costruttore per invocare la "lock()" ed il distruttore per invocare la "unlock()". In caso di eccezione, std::lock_guard<Lockable> sfrutta il paradigma RAII, per cui grazie alla contrazione dello stack sarà sempre chiamato il distruttore che farà una "unlock()".
La std::unique_lock<Lockable> estende il comportamento di std::lock_guard<Lockable> permettendo l'uso dei metodi "lock()/unlock()". Quando viene invocata la std::unique_lock<Lockable> si possiede già il mutex altrimenti lo si acquisisce direttamente. Il processo di creazione di std::unique_lock<Lockable> comporta il possesso del mutex, ma potrebbe capitare che subito dopo sia necessario passare il controllo ad un altro thread per poi riprendere il controllo lasciato. Per fare questo serve quindi invocare una chiamata "unlock()", visto che già si possiede il mutex, per poi invocare la "lock()". La std::unique_lock<Lockable> possiede delle politiche di gestione come "adopt_lock" con la quale si verifica se il thread possiede già il Lockable passato come parametro, ed in tale caso lo adotta. La seconda politica di gestione è "defer_lock" con la quale ci si limita a registrare il riferimento al Lockable senza cercare di acquisirlo.
Sia "unique_lock" che "lock_guard" devono essere dichiarate come variabili locali, così che possano proteggere la regione critica. Non devono mai essere dichiarate come variabili di istanza, altrimenti il thread che crea il mutex come variabile di istanza lo rilascerebbe solo quando viene distrutto. Tanto meno dichiarare "unique_lock" e "lock_guard" come variabili globali, in tale caso solo il thread principale sarebbe il possessore del lock e tutti gli altri thread non potrebbero fare nulla.

30-Spiegare a cosa serve la classe std::atomic<T> e quali metodi offre
Questa classe offre la possibilità di accedere in modo atomico al dato di tipo T garantendone l'accesso in modo concorrente tramite un meccanismo interno di sincronizzazione, quindi sarà ben definito il comportamento nel caso che un thread legga il dato mentre un altro ne modifichi il valore.
I metodi di inizializzazione sono "load()" e "store(T t)" che in modo atomico leggono e modificano il valore della variabile t. Altri metodi atomici sono "fetch_add(val)" che aggiunge alla variabile il valore

val (come +=), "fetch_sub(val)" che sottrae alla variabile il valore val (come -=), "operator++()" che equivale a fetch_add(1), "operator--()" che equivale a fetch_sub(1) ed "exchange(val)" con la quale si assegna alla variabile il nuovo valore "val" restituendo il vecchio valore.

31-Spiegare cosa è e come funziona una std::condition_variable in C++
Una condition variable è un oggetto di sincronizzazione che rilascia temporaneamente un lock in attesa che si verifichi un evento. Il thread resterà in attesa fino al momento in cui un altro thread non invia una notifica per segnalare che qualcosa è successo. Attesa e notifica sono implementate grazie a due metodi "wait()" e "notify()". Il corretto uso di una condition variable prevede l'impiego di una std::unique_lock<Lockable> la quale garantisce che nel momento in cui si esegue il metodo solo un thread sia l'unico proprietario del lock, inoltre garantisce l'assenza di corse critiche nel momento del risveglio. Il metodo "wait()" effettua in sostanza una "unlock()" ed il thread corrente cessa di essere schedulato e viene quindi tolto dalla coda di ready così che possa attendere in modo passivo. Il metodo "notify()" invocato in un altro thread, fa si che il thread precedentemente sospeso venga rimesso nella coda di ready ri-acquisendo il lock. L'identità del thread è scritta all'interno della condition variable. Per evitare notifiche spurie, ossia risvegli errati da parte del thread nonostante non si sia verificato l'evento, è conveniente effettuare il controllo dell'evento direttamente all'interno della condition variable sfruttando le funzioni lambda.

32-Spiegare gli approcci per la programmazione concorrente in C++, (32bis) vantaggi e svantaggi
Sono disponibili due approcci, il primo di alto livello unidirezionale, basato sulla std::async<T> e std::future<T> il secondo bidirezionale di basso livello che prevede l'impiego esplicito di thread e di costrutti di sincronizzazione come "mutex", "condition variabile" e "semaphore".
L'impiego della async<T> permette la scrittura di parti di codice che verranno eseguiti in modo asincrono restituendo subito il controllo al chiamante e rendendo disponibile il risultato di quella porzione di codice in futuro. La async<T> restituisce un oggetto di tipo future<T> e nel momento in cui si procede ad invocare il metodo "get" della future, si chiede di accedere al risultato di quel blocco di codice asincrono. La "get" è bloccante qualora il codice non abbia ancora terminato e prodotto un risultato. In caso di eccezione questa viene propagata nel chiamante. Nel caso peggiore che il codice asincrono non sia ancora stato avviato, la "get" ne forza l'avvio. La "get" fornisce il risultato una ed una sola volta, quindi qualora serva più volte il risultato conviene impiegare la shared_future<T>. Sia la async<T> che future<T> sono thread-safe perché sono state preparate per funzionare in modo corretto su tutte le piattaforme e sono da utilizzare nelle situazioni in cui i compiti da eseguire siano indipendenti tra di loro, in caso contrario di accesso alle medesime risorse servono costrutti più sofisticati per la gestione delle regione critica che sono però di più basso livello. Tutto questo va accompagnato da una creazione esplicita dei thread.
(bis)
La programmazione concorrente all'atto dell'esecuzione di un programma dispone di un unico flusso logico di esecuzione, quello che viene chiamato "Main thread". Quest'ultimo può richiedere la creazione di molti altri thread col fine di creare più flussi logici di esecuzione i quali condividono tutti lo stesso spazio di indirizzamento. Ogni thread ha un proprio stack e condivide tutta la memoria, quindi variabili globali, costanti, heap e spazio di indirizzamento, se sono thread dello stesso processo. I vantaggi nell'uso della programmazione concorrente è che la comunicazione tra thread è banale, visto che la memoria è condivisa, quindi non è necessario utilizzare tecniche complesse come nell'IPC. Nelle architetture multi core ogni thread può essere assegnato dallo scheduler ad un dato core, introducendo un vero grado di parallelismo, con la possibilità di ridurre notevolmente l'intero processo di elaborazione. Altro grande vantaggio è la possibilità che operazioni di I/O non rendono l'intero processo bloccante, infatti il thread che realizza il meccanismo di I/O può essere messo in attesa ed altri thread eseguiti, migliorando nettamente il tempo di esecuzione del processo.
Gli svantaggi della programmazione concorrente è l'elevata complessità del programma ed il suo debug che può divenire non deterministico proprio perché lo schedulatore non è pilotabile dal

programmatore, e diversi avvi del programma potrebbe causare comportamenti diversi difficilmente verificabili. Infine lo stesso programma eseguito su piattaforme diverse potrebbe comportarsi in modo diverso.

33-Spiegare la differenza tra std::thread<T> e std::async<T>

La std::async<T> permette l'esecuzione di codice asincrono in modo automatico, senza che il programmatore debba preoccuparsi di implementare qualche meccanismo di sincronizzazione. E' tutto gestito ad alto livello. La std::async<T> restituisce un oggetto di tipo std::future<T> il quale tramite il metodo "get" permette di recuperare una ed una sola volta il risultato. In caso di usi del risultato in più punti del programma conviene optare per uno std::shared_future<T> accompagnato dal metodo "share" sulla std::async<T>. Con la std::async<T> è possibile impostare una politica di attivazione del thread secondario, utilizzando le opzioni "std::launch:async" e "std::launch:deferred". La prima attiva un thread secondario (con diverso ID) lanciando un'eccezione in caso di mancanza di risorse o multithreading non supportato. La seconda non genera nessun thread secondario, quindi è il solo thread principale che esegue il codice specificato nel metodo del thread solo quando viene chiamato il metodo get() oppure wait() sul future associato alla async. La "std::async<T>" è thread-safe.
La std::thread<T> è di basso livello nel senso che non c'è nulla di automatico, il programmatore deve gestire tutto, dalla creazione del thread, alla gestione dell'accesso concorrente nella regione critica, fino alle gestione delle eccezioni. Eccezioni non gestite causano l'interruzione del programma. La std::thread<T> non ha l'equivalente della "get" della std::async<T> per accedere al risultato, ma si limita ad una "get_id" che restituisce un ID associato al thread stesso, quindi non esiste un metodo standard che permette di accedere al risultato. La std::thread<T> non è thread-safe ed il programmatore deve, per evitare turbe, implementare meccanismi di Race Condition come mutex o condition variable. La st
d::thread<T> non ha una politica di attivazione del thread, quindi si cercherà di creare un thread nativo dell'O.S. il prima possibile, in caso di mancanza di risorse verrà sollevata un'eccezione.

34-Descrivere il ciclo di vita di un thread

Qualsiasi oggetto std:thread<T> non è copiabile, ma solo movibile. Quando si definisce la std::thread<T>, è necessario attendere la terminazione del codice associato al thread tramite una "join()", così che questo thread secondario possa finire. Qualora non si è interessati all'esito della computazione svolta dal thread, si può informare il sistema operativo tramite una chiamata "detach()". E' possibile trasferire le informazioni di un oggetto thread ad un altro tramite movimento, inoltre il distruttore deve attendere che il thread termini, perché se si distrugge il thread principale senza che i thread secondari siano terminati il programma va in crash.

35-Descrivere la sintassi e la semantica di async<T> e i suoi metodi

La sintassi è "async(launch policy, Fn&& fn, ArgTypes&&... args);" dove "Fn&& fn" è il puntatore alla funzione che contiene il codice asincrono da eseguire, mentre " ArgTypes&&... args" i parametri da passare alla funzione. Infine "policy" può valere "launch::async" ad indicare che la funzione viene eseguita da un thread secondario in modo asincrono, oppure "launch::deferred" ad indicare che è il thread primario (chiamante) ad eseguire il codice senza che venga generato nessun thread secondario, ma solo in presenza di una wait/get.
I metodi più importanti della async<T> sono:
"get()" metodo bloccante con il quale si legge il risultato prodotto dal codice asincrono e nel caso il thread non sia ancora partito ne viene forzato l'avvio. Il risultato viene prelevato dal future. La get() può essere chiamata una sola volta.
"wait()" metodo bloccante con il quale si attende il termine dell'esecuzione del thread senza prelevare il risultato

"wait_for()/wait_until()" metodi bloccanti che attendono per un lasso di tempo il completamento del thread
"share()" è un metodo che restituisce uno "shared_future" con il quale si può costruire delle catene asincrone, in tale contesto la get() può venire richiamata più volte nel codice.
In caso di eccezione questa viene propagata al chiamante.

36-Spiegare a cosa serve la "pthread_atfork()"
In presenza di codice sequenziale la "fork()" non da problemi, ma se prima di invocarla si creano dei thread, questo potrebbe creare problemi di congruenza sugli oggetti di sincronizzazione, visto che il nuovo processo si aspetta magari di avere delle cose che invece non possiede o avere gli oggetti di sincronizzazione in stati incongruenti. Per risolvere questa problematica conviene utilizzare la "pthread_atfork()", la quale verifica lo stato di consistenza delle risorse di sincronizzazione rilasciando eventualmente i mutex. Con questa funzione è possibile registrare le callback in presenza di altri thread.

37-Che cosa è la PThread?
La PThread è una libreria di gestione dei thread in ambiente Linux standard POSIX definita nel header file "pthread.h". In fase di compilazione e collegamento, è necessario segnalare il flag "-pthread" altrimenti si avranno una serie di messaggi di errore. Nella PThread sono presenti strutture dati e funzioni per la gestione del ciclo di vita di un thread.

38-Che cosa si intende per interferenza? Come si risolve?
L'interferenza è il fenomeno che si verifica quando due o più thread accedono contemporaneamente al medesimo dato, causando malfunzionamenti casuali molto difficili da identificare. Un esempio è l'incremento di una variabile "a++" che non essendo un'operazione atomica, ma una cascata di due operazioni semplici, può causare problemi difficilmente rilevabili. Per evitare problemi di interferenza è necessario utilizzare dei costrutti di sincronizzazione che disciplinino l'accesso alla risorsa condivisa.

39-Che cosa è la "Lazy Evaluation"?
E' una tecnica con la quale si cerca di rimandare la creazione e l'inizializzazione di strutture dati fino al loro reale impiego. Con programmi sequenziali è facilmente realizzabile sfruttando un puntatore nullo, mentre nei programmi concorrenti, essendo tale puntatore condiviso da più thread, è obbligatorio utilizzare un mutex come protezione elementare anche se il C++ 2011 offre le classi std::once_flag e std::call_once per creare un thread safe.
Il primo thread che chiama la "call_once" inizializza l'oggetto, gli altri restano in attesa su tale chiamata per poi essere fatti uscire se il primo thread inizializza l'oggetto.

40-Spiegare a cosa serve la "CreateProcess()" in C++
La "CreateProcess" in ambiente windows serve per creare un nuovo spazio di indirizzamento dentro il quale copiare l'immagine dell'eseguibile attivando un thread principale da assegnare alla cpu. Eventuali processi figli potranno condividere le variabili d'ambiente, le handle a file, pipe e semafori, ma non handle a thread, librerie dinamiche e regioni di memoria. In Unix le cose sono leggermente diverse, visto che la "fork()" o "vfork()" effettuano un clone dell'area di memoria. La "vfork()" inizialmente non duplica la pagina, ma solo quando si effettuano scritture (CopyOnWrite), inoltre quando si chiama la "exec*()" nelle sue varianti, questa resetta lo spazio di indirizzamento corrente sostituendolo con un nuovo eseguibile, quindi l'istruzione successiva alla "exec*()" non verrà mai richiamata se non in caso di errore della stessa "exec*()"

41-Spiegare la differenza tra rappresentazione interna ed esterna. Cosa significa serializzazione?
La rappresentazione interna è relativa a come un processo rappresenta le informazioni in memoria sfruttando tipi di dati elementari, come numeri, caratteri, oppure dati strutturati, come classi, struct

o strutture dati più complesse come alberi, grafi grazie all'impiego di puntatori. Questa rappresentazione non è adatta ad essere esportata verso altre macchine, visto che un puntatore ha senso solo all'interno del suo spazio di indirizzamento ed altre informazioni non sono nemmeno esportabili, come ad esempio gli handle sotto windows.
La rappresentazione esterna basata su formati testuali utilizza molta memoria, ma grazie alla sua flessibilità è possibile esportare i dati tramite il protocollo HTTP. I formati di testo più in uso sono XML e JSON. E' possibile utilizzare anche formati binari come XDR, anche se stanno andando in disuso.
La serializzazione è il meccanismo che traduce da rappresentazione interna ad esterna indipendentemente dalla macchina, come sequenza seriale di bit. La serializzazione è anche chiamata marshalling. L'operazione inversa è chiamata de-serializzazione o unmarshalling

42-Descrivere i metodi IPC in Windows
Le IPC più utilizzate sotto Windows sono le mailslot, le pipe, file mapping, socket e RPC.
Una mailslot è una coda dei messaggi asincrona, la quale gestisce sequenze di blocchi di informazioni. La coda è a priorità di messaggio.
Agli estremi della coda vi sono due processi che possono essere contemporaneamente mailslot client e mailslot server, quindi il canale è bidirezionale. Le mailslot sono create in locale tramite l'istruzione "CreateMailslot()", mentre i messaggi sono letti con della API "ReadFile()" e "ReadFileEx()". L'inserimento di messaggi avviene tramite la "WriteFile()" e "WriteFileEx()". Per verificare la presenza di messaggi si utilizza la "GetMailslotInfo()". Infine la "CloseHandle()" chiude tutti gli handle rilasciando tutte le risorse.
La pipe è un oggetto che rappresenta uno stream di dati e può essere di tipo anonimo "anonymous pipe" oppure dotata di nome "named pipe".
La "anonymous pipe" sono monodirezionali e si impiegano in presenza di processi padre-figlio dove lo standard output del padre deve venire rediretto sullo standard input del processo figlio.
Le funzioni "ReadFile()" e "WriteFile()" servono a leggere e scrivere dalla pipe senza che vi sia nessun vincolo di atomicità come nelle rispettive "ReadFile()" e "WriteFile()" della mailslot. Non è supportato I/O asincrono. La "named pipe" permette la comunicazione bidirezionale tra processi appartenenti a macchine diverse. La pipe viene identificata con un nome "\\ServerName\pipe\PipeName". Le funzioni "ReadFile()" e "WriteFile()" permettono di leggere e scrivere su file.
Il file mapping è un insieme di due o più pagine fisiche mappate su più spazi di indirizzamento che però richiede un meccanismo di sincronizzazione per essere utilizzata. Il contenuto di un file è visto come se fosse un blocco di memoria.
Il socket è un astrazione software che permette la comunicazione tra macchine con sistemi operativi differenti trasferendo array di byte. RPC è una tecnica basata sui socket che permette l'invocazione remota di metodi tramite meccanismo di alto livello.

43-Descrivere le differenze e le similitudini in Win32 tra mutex e semafori
Sia mutex che semafori sono oggetti kernel di sincronizzazione Win32, così come gli eventi. Un semaforo binario può essere considerato come un mutex. Tutti questi oggetti di sincronizzazione utilizzano due funzioni di attesa Win32 che sono la WaitForSingleObject() e la WaitForMultiObject(). La WaitForSingleObject attende che un oggetto kernel entri nello stato segnalato o che trascorra un timeout, mentre la WaitForMultiObject attende che uno o tutti gli oggetti kernel entrino nello stato segnalato.
Un mutex serve per proteggere il codice condiviso della regione critica tramite l'ausilio delle chiamate "lock()/unlock()" così che un solo thread alla volta può avere accesso in modo esclusivo alla regione critica. Il mutex può venire utilizzato anche per thread di processi diversi. Il mutex ha un ID per conoscere il thread che lo ha acquisito ed un contatore per conoscere il numero di volte il thread ha acquisito la risorsa. Le due funzioni di attesa WaitForSingleObject() e la WaitForMultiObject() attendono se il mutex è non segnalato, altrimenti lo acquisiscono incrementando il contatore.

Un semaforo in generale possiede un contatore >0 per indicare lo stato segnalato, mentre ad ogni chiamata della WaitForSingleObject() questo viene decrementato e nel momento in cui arriva a 0 lo stato dell'oggetto diviene non segnalato ed il thread bloccato in attesa che venga incrementato il contatore tramite la chiamata ReleaseSemaphore() . I semafori sono molto comodi a basso livello per introdurre meccanismi di sincronizzazione tra thread di processi diversi. In Windows i semafori hanno un limite massimo per il contatore, quindi se si tenta di incrementarlo ulteriormente, il semaforo va in uno stato di errore e cessa di funzionare.

44-Si illustrino i meccanismi Win32 di sincronizzazione

Ad ogni handle del kernel object è associato un meccanismo di segnalazione che consente di realizzare meccanismi di sincronizzazione. Ogni handle possiede due stati: segnalato (signaled) e non segnalato (non-signaled).
Questo meccanismo viene fatto da due funzioni di attesa dell'API Win32: WaitForSingleObject() e WaitForMultiObject() che prendono come parametro un HANDLE di kernel object e sospendono il thread chiamante fino a quando questo non diviene segnalato. Le modalità di passaggio sono diverse per ogni kernel object, in ogni caso lo stato di un thread è non segnalato finché il thread è in esecuzione e diviene segnalato solo quando termina.
Tutti gli oggetti kernel per la sincronizzazione si appoggiano su queste due funzioni di attesa, come ad esempio gli eventi, i mutex o i semafori. Un evento segnala il verificarsi di una situazione sulla base di quello che vuole fare il programmatore. Gli eventi possono essere di tipo "manual reset" ossia tutti i thread vengono risvegliati, come se fosse una specie di "notify_all", oppure possono essere di tipo "auto reset" con il quale un solo thread si risveglia, come nel caso della "notify_one".
E' possibile utilizzare i semafori per la sincronizzazione, i quali mantengono un contatore >0 per lo stato segnalato, mentre quando arrivano a 0 lo stato è non segnalato. La "WaitForSingleObject" decrementa di uno il contatore e blocca il thread se il contatore è uguale a 0, in attesa che altri thread incrementino di uno il valore. Il semaforo ha un limite superiore oltre il quale non può essere incrementato, ogni tentativo provoca un errore del semaforo il quale cessa di funzionare.

##################
05/07/2019
##################

1. Connect di QT

2. Eccezioni in C++

3. La uniquelock/uniqueptr è solo movibile o anche assegnabile?

##################
24/07/2019
##################

1. Quali sono i vantaggi degli iteratori negli algoritmi della Standard Template Library (1pt)
2. Descrivere il caricamento dinamico delle librerie sul sistema operativo Windows/Linux (1pt)
3a. lambda: cos'è una chiusura, tutte le opzioni per catturare parametri pro e contro
3b. Descrivere le funzioni lambda nell'ambito del linguaggio C++, e suggerire una strategia adottabile dal compilatore per implementare questa astrazione (2pt)
4. Un sistema embedded riceve su due porte seriali sequenze di dati provenienti da due diversi sensori. Ciascun sensore genera i dati con cadenze variabili nel tempo e non predicibili, in quanto il processo di digitalizzazione al suo interno può richiedere più o meno tempo in funzione del dato letto. Ogni volta che il sistema riceve un nuovo valore su una delle due porte seriali, deve accoppiarlo con il dato più recente ricevuto sull'altra (se già presente) e inviarlo ad una fase successiva di computazione. Il sistema al proprio interno utilizza due thread differenti per leggere dalle due porte seriali e richiede l'uso di un oggetto di sincronizzazione in grado di implementare la logica descritta sopra. Tale oggetto offre la seguente interfaccia pubblica:

```
class Synchronizer {
public:
 Synchronizer(std::function<void(float d1, float d2)> process);
 void dataFromFirstPort(float d1);
 void dataFromSecondPort(float d2);
}
```

All'atto della costruzione, viene fornita la funzione process(...) che rappresenta la fase successiva della computazione. Quando vengono invocati i metodi dataFromFirstPort(...) o dataFromSecondPort(...), se non è ancora presente il dato dalla porta opposta, questi si bloccano al proprio interno senza consumare CPU, in attesa del valore corrispondente. Al suo arrivo, viene invocata una sola volta la funzione process(...). Si implementi tale classe utilizzando le funzionalità offerte dallo standard C++.

##################
06/09/2019
##################
1. Tutto sui metodi virtual
2. primitive IPC in Windows
3. Meccanismo di gestione degli eventi nel framework .NET/QT
4. Gioco online tra due giocatori, gestione di più thread
Implementare classe Challenge che aveva, oltre al costruttore, i seguenti metodi:
-void accept(Boolean response)
Indica il responso dello sfidato alla richiesta di giocare
-void result1(int score)
Indica lo score dello sfidante
-void result2(int score)
Indica lo score dello sfidato
-int winner()
Ritorna:
-0 se è pari
-1 se ha vinto lo sfidante
-2 se ha vinto lo sfidato
- -1 se la richiesta è stata rifiutata
Accept deve precedere result1 e result2, che invece non hanno ordine di chiamata tra loro, e tutti e 3 possono essere chiamati una sola volta, la violazione dei vincoli genera eccezione.
Winner si mette in attesa senza consumare CPU se il gioco è in corso o non è ancora cominciato

##################
20/01/2020
##################

1. IPC Windows
2. Unique ptr
3. Connect in QT
4. È presente una classe Executor che ha il compito di rimanere in ascolto su una coda di funzioni da svolgere e si appoggia su una classe esterna Context

```
class Executor {
public:
Executor(Context *ctx);
~Executor();
std::future<void> submit(std::function<void(Context*)> f);
void shutdown();
};
```

Quando viene costruita la classe, viene generato un unico thread che ha il compito di estrarre le funzioni dalla coda, immesse tramite il metodo submit(), ed eseguirle. Esso termina quando viene chiamato il metodo shutdown() o il distruttore della classe.
Le funzioni da svolgere vengono immesse tramite un metodo submit, che torna un oggetto future col risultato dell'elaborazione. Se il metodo viene chiamato quando è già stato chiamato il metodo shutdown, viene lanciata un'eccezione di tipo std::logic_error.
Il metodo shutdown viene chiamato per terminare l'esecuzione, e se ci sono ancora delle funzioni in coda, esse vengono comunque svolte.
Implementare la classe blabla

##################
27/06/2020
##################

1- Si confrontino i puntatori nativi con gli smart pointer offerti dalla libreria standard del linguaggio C++, evidenziandone pregi e limiti. Si scriva un frammento di codice che mostri l'utilizzo ed il rilascio di un blocco allocato dinamicamente da parte di uno smart pointer (3.0)

[Commenti dal prof:
-non si spiega il problema che risolvono i weak_ptr
-nessuna menzione alle performance
- non è solo in problema di rilascio automatico - che problemi introducono, a che servino i weak ptr? le prestazioni?
]

2- Si descriva il funzionamento del meccanismo di gestione delle eccezioni nel linguaggio C++ e si scriva un esempio concreto che mostri come sia possibile implementare una strategia di gestione degli errori. (3.0)

[Commenti dal prof:
- non solo la chiusura del processo crea problemi di rilascio
- abbastanza pasticciata la descrizione dei problemi da risolvere
- come sono definite le eccezioni e qual è il meccanismo di catch?
- Non lancia nessuna eccezione per div per 0, il risultato è NaN (devi controllare tu)
- sbrigativa la discussione della contrazione dello stack: perché è importante nel flusso di gestione?
]

3-Si metta a confronto il modello di esecuzione concorrente basato su più processi con quello basato su più thread all'interno dello stesso processo e si evidenzino pregi e difetti in termini di robustezza, prestazioni, semplicità di sviluppo di ciascuno di essi. (3.0)
(100%)
Il modello di esecuzione concorrente basato su più processi differisce molto da quello basato su più thread all'interno dello stesso processo. La prima differenza sta nel fatto che quando viene eseguita la fork() (consente la creazione di un processo figlio) viene generato un nuovo address space che non riesce a vedere nulla di ciò che riguarda il processo padre e per questo motivo bisogna cercare di far comunicare i processi tra di loro in modo da poter scambiare informazioni e sfruttare al massimo il nuovo processo creato. La comunicazione tra processi avviene mediante una pipe che può essere half/full duplex e consente la scrittura/lettura di soli dati binari e che quindi bisogna gestire convertendo opportunamente le strutture dati in un formato binario (es. JSON) prima di scrivere la pipe. Nella creazione di un nuovo thread, invece, poiché viene condiviso lo spazio di indirizzamento è possibile fare accesso alle stesse strutture dati (senza dover generare una copia in binario) ma ciò implica di dover gestire le eventuali sezioni critiche attraverso lock e/o condition variables. Sicuramente sviluppare un processo risulta più semplice perché non c'è bisogno di gestire la sincronizzazione che è uno degli aspetti più critici della programmazione concorrente, ma d'altra parte bisogna lavorare di più per poter consentire la comunicazione. In termini di efficienza è sicuramente più vantaggioso utilizzare i thread (se il codice è ben scritto) perché viene assegnata una cpu ad ogni thread e quindi, nel caso di multicore, si ottiene si ottengono le massime prestazioni.

[Commenti dal prof:
]

4-Un'azienda ha sviluppato un software multipiattaforma (Windows/MacOSX/Linux) che purtroppo soffre di alcune criticità: al suo interno sono presenti alcune funzioni ben note che quando vengono invocate determinano l'allocazione di una grande quantità di memoria, che viene poi correttamente liberata quando esse ritornano; poiché il programma è concorrente, se più thread iniziano ad eseguire contemporaneamente tali funzioni, determinano una allocazione di memoria complessivamente superiore a quanto il sistema permetta, provocando così la morte del processo. Siete stati incaricati di sviluppare un frammento di codice conforme allo standard C++11 che consenta di limitare il numero massimo di invocazioni concorrenti di tali funzioni provocando un'attesa iniziale che non consumi CPU fino a che il numero di invocazioni in corso è sceso sotto una data soglia. Tale codice deve essere soggetto ai seguenti vincoli:

-Essere compatibile con l'ambiente multi-threaded e con tutti i sistemi operativi citati.

-Permettere di impostare staticamente il numero massimo di invocazioni concorrenti accettate.

-Garantire che, qualunque sia la motivazione che porta ad uscire dalle funzioni sotto osservazione, venga correttamente aggiornato il conteggio, così da consentire ad altri thread di procedere.

Si scriva il codice necessario e si indichi come devono essere modificate le funzioni "sospette" per poterlo utilizzare (5.0)

(risoluzione del prof)

```
#include <iostream>

#include <thread>

#include <mutex>

class semaphore {

  int count;

  std::mutex m;

  std::condition_variable cv;

public:

  explicit semaphore(int count): count(count) {}

  void acquire() {

     std::unique_lock ul(m);

     cv.wait(ul,[this]() {return this->count>0;});

              count--;

           }

           void release() {

              std::unique_lock<std::mutex> ul(m);

              count++;

              cv.notify_one();

           }
```

```
};

class raii_guard {

  semaphore& s;

public:

  explicit raii_guard(semaphore& s): s(s) {

     s.acquire();

  }

  ~raii_guard() {

     s.release();

  }

};
```

[Commenti dal prof:
-non hai capito la domanda. Il codice deve prevenire l'errore di out_of_memory che si verifica nel programma esistente, non riconoscerlo a posteriori né tantomeno sostituire il main()
-Il codice indicato deve essere replicato in tutte le funzioni. Se una funzione ha più punti di uscita, il codice della seconda parte deve essere replicato per ciascuno Gestisci solo eccezioni che derivano da std::exception, e introduci un errore se viene lanciato un tipo non correlato ad ess
]

##################
10/07/2020
##################

1-Si illustrino le differenze nel linguaggio C++ tra costruttore di copia e di movimento. Si implementi il costruttore di movimento di una classe che possiede una risorsa di tipo FILE* e si indichi quali vantaggi/responsabilità tale costrutto introduce. (3.0)

[Commenti dal prof:
- è precidibile quale venga usato
- i vantaggi?
- sulle resposabilità c'è un po' di confusione 1) è costruttore, non assegnazione 2) NON FAI LA DELETE! 3) non esiste perché non fai la new, hai confuso con il costruttore di copia
]

2- Si descriva concetto di specializzazione di un template nel contesto della programmazione generica, indicando le ragioni che possono portare un programmatore ad utilizzare tale costrutto - Si fornisca un esempio di specializzazione per la classe std::vector (3.0)
(100% ma è stato dato comunque il primo commento riportato)
I template permettono di generalizzare gran parte delle cose ma non tutto. In alcuni casi ho bisogno di specializzare il mio template in modo da definire un particolare tipo di comportamento nel caso di una classe specifica.
Se per esempio ho un std::vector<Oggetto> dove la classe Oggetto è definita in questo modo:

```
class Oggetto{
        int nome;
        std::string descrizione;
public: Oggetto(nome, descrizione):nome(nome), descrizione(descrizione){}
}
```

una possibile specializzazione potrebbe essere

```
template<typename T>
class vector{
.........
//Relativi costruttori e metodi generici basati sul tipo T
........
//Questo è un esempio di specializzazione per il metodo clear che normalmente svuoterebbe il vettore di tutti i suoi elementi, mentre nel caso particolare di std::vector<Oggetto> poiché l'ho espressamente ridefinito non farà altro che modificare le descrizioni di tutti gli elementi del vettore
        std::vector<Oggetto>& clear (std::vector<Oggetto>& oggetti){
                for (int i = 0; i < oggetti.size(); i++){
                        oggetti[i].descrizione=' ';
                }
                return oggetti;
                }
        }
```

[Commenti dal prof:
-In realtà esiste anche il caso in cui una classe potrebbe non essere utilizzabile in un template (es manca operatore). Anziché derivare classe si può specializzare template.
]

3-Un server di rete gestisce connessioni TCP di lunga durata su ciascuna delle quali vengono scambiate molteplici coppie di richieste/risposte, conformi ad un protocollo applicativo dotato di stato (ovvero, nel quale la risposta ad una richiesta dipende dalle richieste/risposte precedenti nell'ambito della stessa connessione come, ad esempio, il protocollo FTP).
Si descrivano, ad alto livello, i passi necessari per gestire le connessioni entranti e servire le risposte, evidenziando una possibile
strategia di gestione in termini di concorrenza e/o asincronia. (3.0)

[Commenti dal prof:
Generico: non sei entrato nel contesto del problema posto
]

4-In una macchina utensile, sono in esecuzione N thread concorrenti, ciascuno dei quali rileva continuamente una sequenza di valori, risultato dell'elaborazione delle misurazioni di un sensore. I valori devono essere raggruppati N a N in una struttura dati per essere ulteriormente trattati dal sistema. A questo scopo è definita la seguente classe thread-safe:
class Joiner {
public:
Joiner(int N); // N is the number of values that must be conferred
std::map<int, double> supply(int key, double value);
};
Il metodo bloccante supply(...) riceve una coppia chiave/valore generata da un singolo thread e si blocca senza consumare CPU fino a che gli altri N-1 thread hanno inviato le loro misurazioni. Quando sono arrivate N misurazioni (corrispondenti ad altrettante invocazioni concorrenti), si sblocca e ciascuna invocazione precedentemente bloccata restituisce una mappa che contiene N elementi (uno per ciascun fornitore). Dopodiché, l'oggetto Joiner pulisce il proprio stato e si prepara ad accettare un nuovo gruppo
di N misurazioni, in modo ciclico.
Si implementi tale classe, facendo attenzione a non mescolare nuovi conferimenti con quelli della tornata precedente (un thread appena uscito potrebbe essere molto veloce a rientrare, ripresentandosi con un nuovo valore quando lo stato non è ancora stato ripulito). (5.0)
(risolto dal prof)

```
#include <iostream>

#include <mutex>

#include <condition_variable>

#include <stdexcept>

#include <vector>

#include <thread>

#include <map>

class Joiner {

   std::mutex m;

   std::condition_variable cv;

   std::map<int,double> values;

   int inside;

   bool entering;

   int max;
```

```
public:
   Joiner(int max): max(max), inside(0), entering(true) {
       if (max<=1) throw std::logic_error("A Joiner requires at least 2 elements");
   }
   std::map<int,double> supply(int key,double val) {
       std::unique_lock<std::mutex> ul(m);
       cv.wait(ul,[this]() {return entering;});
       values.insert(std::pair(key,val));
       inside++;
       if (inside==max) {
           entering = false;
           cv.notify_all();
       } else {
           cv.wait(ul, [this]() {return !entering;});
       }
       inside--;
       if (inside==0) {
           entering=true;
           cv.notify_all();
           auto res{values};
           values.clear();
           return res;
       } else return values;
   }
};
```

[Commenti dal prof:
Mescoli i nuovi conferimenti con quelli della tornata precedente.
Modifichi la variabile N perdendo il conto di quanti thread debbano essere attesi
]

##################
12/09/2020
##################

1-Come viene gestito dal compilatore C++ il polimorfismo nell'ereditarietà tra classi e mediante l'uso dei template? Argomentare la risposta con un esempio. (4.0)
(100%)
In C++ il polimorfismo nell'ereditarietà tra le classi viene gestito mediante la V-Table. Ogni volta che un metodo di una classe viene dichiarato virtual viene inserita una entry nella V-Table con il puntatore alla funzione corretta da chiamare. Per esempio:

```
class Base{
public:
	virtual std::string function (void) { return "Base" }
}
class Der: public Base{
public:
	std::string function (void){ return "Der"; }
}
int main(){
	Base *b = new Der();
	std::cout<<b->function()<<std::endl;
	return 0;
}
```

In questo caso poiché function è definita come virtual nella classe Base b->function() restituirà la stringa "Der" poiché nella V-Table ci sarà un puntatore alla funzione di Der. Se il metodo non fosse stato dichiarato come virtual, b->function() avrebbe restituito "Base".

I template consentono di gestire il polimorfismo statico senza ricorrere all'ausilio di una struttura dati aggiuntiva. Se per esempio consideriamo il seguente pezzo di codice:

```
template <typename T>
T add (T t1, T t2){
	return t1 + t2;
}
```

La funzione add assumerà un diverso significato in base a come viene richiamata:

```
add <int> (1, 2); // Qui effettuerà la normale somma aritmetica
add <std::string> ("stringa1", "stringa2"); // Qui effettuerà la concatenazione tra stringhe
```

E' possibile anche effettuare una sorta di polimorfismo dinamico, simile a quello ottenuto con l'ereditarietà ma senza V-Table e sfruttando i template, attraverso il paradigma CRTP che funziona nel seguente modo:

```
template <typename T>
class C1{
public:
	getValue(){
		return std::reinterpret_cast(*this)->getValue();
	}
}
class C2: public C1 <C2>{
public:
```

```
        int getValue(){
                return 2;
        }
}
class C3: public C1 <C3>{
public:
        int getValue(){
                return 3;
        }
}
```

[Commenti dal prof:
- anche la velocità di esecuzione migliora con i template
- l'esempio dei template è di programmazione generica, non crtp
]

2-Illustrare il pattern RAII, spiegando in dettaglio il modello di memoria su cui si basa, e indicare tre diversi casi d'uso in cui può essere usato. (3.0)
(Risposta con valutazione 90% e commentato con il secondo)
Il pattern RAII (resource acquisition is initialization) è un metodo di programmazione in uso nel linguaggio c++ che prevede l'acquisizione della memoria durante l'invocazione del costruttore di un oggetto e il suo rilascio durante l'invocazione del suo distruttore, non essendo il c++ fornito di garbage collector: è particolarmente vantaggioso utilizzarlo perchè permette di avere delle garanzie sul corretto rilascio della memoria, soprattutto in caso di sollevamento di eccezioni, in quanto, venendo chiamato il distruttore una volta effettuato lo stack unwinding (procedura che si innesca dopo un'eccezione), il distruttore di una classe conforme al pattern RAII garantirà che questa venga correttamente rilasciata, evitando memory leakage o dangling pointers e, conseguentemente, togliendo dalle spalle del programmatore qualche responsabilità. Dei casi in cui è particolarmente utile usare il pattern RAII sono i seguenti: 1. gestione di puntatori -> è utile wrappare i puntatori nativi in delle classi che seguono il paradigma RAII, per avere garanzie sul loro rilascio una volta invocato il distruttore della classe che li wrappa (in c++: smart pointers) 2. gestione dei mutex -> si wrappano i mutex nelle classi che seguono il paradigma RAII in modo che questi vengano "unlockati" durante l'invocazione del distruttore (in cpp: lock/lock_guard/unique_lock/...) 3. getsione delle strutture dati -> se una struttura dati dovesse chiamare una realloc in modo da aumentare la propria dimensione e non ci dovesse esssere più memoria dipsonibile nell'elaboratore, verrebbe sollevata un'eccezione, per questo è bene wrappare le strutture dati dentro classi conformi a RAII, in modo che le risorse attualmente acquisite vengano rilasciate correttamente (in cpp: container della STL)

[Commenti dal prof:
- veniva chiesto anche di spiegare il modello di memoria, ovvero come è legato allo stack
- il RAII è legato sempre alla gestione dello stack, non solo per le eccezioni. Quando si esce da un una funzione o uno scope locale tutte le varibili vengono distrutte
]

3-Spiegare le ragioni per cui può convenire usare un modello di concorrenza basato su processi invece che su thread.
Quali accorgimenti occorre prendere in questo caso?
Quali risorse sono disponibili per i figli dopo una fork(), come può avvenire la comunicazione tra padre e figli e come avviene la corretta terminazione dei processi in un sistema operativo Unix-like? (4.0)
(100%)

Il processo, essendo un'entità che possiede un proprio spazio di indirizzamento e un suo stack di memoria, è un'entità a se' stante, ovvero è completamente agnostica sull'esistenza di altri processi, per questo garantisce un maggior grado di sicurezza per quanto riguarda l'accesso a porzioni di memoria altrui (eventualità più rara rispetto ai thread, i quali condividono lo stesso spazio di indirizzamento e che necessitano di strutture di sincronizzazione per scongiurare l'accesso contemporaneo alla stessa zona di memoria, che potrebbe portare a comportamenti impredicibili). I processi infatti, al contrario dei thread, non hanno una maniera "nativa" di comunicare o di condividere risorse, ma è il sistema operativo che deve implementare dei metodi per permettere questo scambio,i cosiddetti metodi di "inter process comunication" (come le pipe). Il formato interno di ogni processo è però unico per ogni processo; per questo, per far comunicare più processi, occorre adottare un accorgimento, ossia bisogna tradurre il formato interno dei dati di un processo, in un formato esterno (XML, JSON, ecc) comprensibile da entrambi i processi, e poi ritradurlo in un formato interno da parte dell'altro processo. Questo metodo si chiama marshaling (traduzione da formato interno a esterno) e unmarshaling (traduzione da formato esterno a formato interno). Inoltre, è molto importante sottolineare la maggiore facilità di debug: per un sistema concorrente basato su processi, è molto più semplice il debug da parte del programmatore, piuttosto che in un sistema basato su thread, essendo il comportamento di questi ultimi potenzialmente impredicibile. Dopo una fork il figlio ha un nuovo spazio di indirizzamento, in cui viene copiato in maniera 1 ad 1 quello del padre. La comunicazione tra un processo padre e uno figlio può avvenire, per esempio, per mezzo di una pipe: il padre crea la pipe e fa la fork, successivamente il padre chiude un'estremità della pipe, il figlio chiude l'altra e la comunicazione può iniziare tramite l'utlizzo di read e write. es. in pseudo-codice

```
int main(){
        int fd[2];
        pipe p(fd);
        if(fork()) {
                closePipeFromOneSide(p);
        }
        closePipeFromOtherSide(p);
         //communicating...
}
```

La corretta terminazione di un processo in un sistema unix-like avviene per morte "naturale" del processo, ossia quando il main thread di quel processo arriva alla funzione return e alla fine dello scope, così che possa invocare il distruttore di tutte le variabili allocate. Il processo padre attende un figlio chiamando la funzione "waitpid()", la quale aspetta la terminazione del figlio. Se il processo padre termina prima che abbia terminato il processo figlio, questo diventa orfano. Se invece il processo figlio termina prima che il padre abbia chiamato la waitpid() questo diventa uno zombie.

[Commenti dal prof:
-Cosa succede dopo la fork ai descrittori di file? Sono o meno visibili ad entrambi i processi?
-Che responsabilità ha il processo padre nei confronti della terminazione del processo figlio?]

4-La classe CountDownLatch permette di sincronizzare uno o più thread che devono attendere, senza consumare CPU, il completamento di operazioni in corso in altri thread.
All'atto della costruzione, gli oggetti di questa classe contengono un contatore inizializzato con un valore intero strettamente positivo. Oltre al costruttore, questa classe offre due soli metodi pubblici: void await() e void countDown(). Quando un thread invoca await(), rimane bloccato fino a che il contatore non raggiunge il valore 0, dopodiché ritorna; se, viceversa, all'atto della chiamata il contatore vale già 0, il metodo ritorna immediatamente.

Quando viene invocato countDown(), se il contatore è maggiore di zero, viene decrementato e se, come conseguenza del decremento, diventa nullo libera i thread bloccati all'interno di await(). Se, viceversa, il contatore valeva già a zero, l'invocazione di countDown() non ha effetti.
Si implementi tale classe utilizzando le librerie standard C++11.(3.0)

[Commenti dal prof:
Costruttore e metodi await() e countDown() devono essere pubblici.
Non hai verificato che il parametro passato al costruttore sia non negativo
]