Ottimizzare C++

Indice

Capitolo 1 – Introduzione

Il principale motivo per cui si sceglie si sviluppare del software in C++, invece che in linguaggi di programmazione di livello più alto, è il fatto che questo linguaggio consente di produrre software complesso più efficiente di quello prodotto usando altri linguaggi. Si noti che il linguaggio non garantisce automaticamente di produrre software efficiente, ma lo consente soltanto. Infatti, scrivendo di getto lo stesso programma in C++ e in linguaggi di livello più alto, tipicamente la versione sviluppata in C++ è altrettanto efficiente quanto quella sviluppata negli altri linguaggi.

Tuttavia, un buon programmatore C++, seguendo delle linee-guida apprese da programmatori esperti, o dalla propria esperienza, o da questo libro, è in grado, in primo luogo, di scrivere del software discretamente efficiente fin dalla prima stesura, e poi di *ottimizzare* il programma risultante, cioè incrementare sensibilmente le prestazioni del programma sostituendo alcuni costrutti con altri equivalenti ma più efficienti. Tale ottimizzazione richiede in primo luogo che il codice sorgente sia stato scritto in modo sufficientemente modulare da isolare le parti più critiche per le prestazioni, e poi di impiegare strumenti, librerie, conoscenze, e tempo, per applicare le modifiche necessarie ad ottimizzare le prestazioni del software prodotto.

Oggigiorno, molte delle sostituzioni che ottimizzano il software sono già effettuate dai compilatori, e quindi non sono più compito del programmatore. Tuttavia, molte altre ottimizzazioni non sono ancora ottenibili dagli attuali compilatori. Questo libro tratta proprio delle tecniche di ottimizzazione che spettano al programmatore, in quanto non sono applicate da tutti gli attuali compilatori.

Questo libro è rivolto ai programmatori che sanno già usare il linguaggio C++, e che lo vogliano impiegare per sviluppare software applicativo o librerie di alta qualità.

Quasi tutte le tecniche di ottimizzazione qui trattate sono indipendenti dalla piattaforma, e pertanto raramente si farà riferimento a particolari sistemi operativi, a particolari architetture di processore, o a particolari compilatori. Tuttavia, si deve tenere presente che alcune delle tecniche proposte risultano non convenienti o persino inapplicabili in alcune combinazioni di sistema operativo/processore/compilatore.

Capitolo 2 – Ciclo di vita dell'ottimizzazione

La costruzione di un'applicazione efficiente dovrebbe procedere secondo il seguente processo di sviluppo:

1. **Progettazione** (*design*). Dapprima, si progettano gli algoritmi e le strutture dati in modo tale che abbiano senso per la logica applicativa, e che siano ragionevolmente efficienti, ma senza occuparsi di ottimizzarle. Dove si deve definire una struttura dati di ampio utilizzo e per la quale non è ovvio quale sia l'implementazione ottimale (per esempio, non si sa scegliere tra un array e una lista collegata), si definisce una struttura astratta, la cui implementazione possa essere cambiata in fase di ottimizzazione.
2. **Codifica** (*coding*). Poi si scrive il codice che implementa gli algoritmi progettati, seguendo linee-guida che permettano di evitare alcune operazioni inefficienti e di incapsulare le operazioni che probabilmente richiederanno ottimizzazioni.
3. **Collaudo funzionale** (*functional testing*). Poi si collauda il software prodotto, in modo da aumentare la probabilità che non abbia difetti rilevanti.
4. **Ottimizzazione** (*tuning*). Dopo aver completato lo sviluppo di un'applicazione o libreria funzionante correttamente, si passa alla fase di ottimizzazione, costituita dalle seguenti sotto-fasi:
    1. **Collaudo prestazionale** (*performance testing*). Si valuta quali comandi hanno prestazioni inadeguate, cioè si identificano i comandi che, elaborando dei dati tipici, richiedono più memoria o più tempo di quelli massimi specificati nei requisiti.
    2. **Analisi delle prestazioni**. Per ogni comando avente prestazioni inadeguate, si determina, usando un profiler, quali porzioni di codice costituiscono i cosiddetti **colli di bottiglia** per tale comando. Cioè si identificano le porzioni di codice nelle quali, tra l'inizio del comando e il suo completamento, viene trascorso più tempo e viene allocata più memoria.
    3. **Ottimizzazione algoritmica**. Nei colli di bottiglia, si applicano tecniche di ottimizzazione sostanzialmente indipendenti dal linguaggio di programmazione, e totalmente indipendenti dalla piattaforma. Sono le tecniche che si trovano nei testi di teoria degli algoritmi. In pratica, si cerca di ridurre il numero di istruzioni eseguite, e in particolare il numero delle chiamate a routine costose, oppure a trasformare le chiamate costose in chiamate equivalenti ma meno costose. Per esempio, si sceglie di implementare l'algoritmo di ordinamento *quick sort* invece dell'algoritmo *selection sort*. Se questa ottimizzazione rende il programma sufficientemente veloce, si termina l'ottimizzazione.
    4. **Ottimizzazione indipendente dalla piattaforma**. Nei colli di bottiglia, si adottano tecniche di ottimizzazione dipendenti dal linguaggio di programmazione e dalla sua libreria standard, ma indipendenti sia dalla piattaforma software che dalla piattaforma hardware. Per esempio, si usano operazioni intere invece di operazioni a virgola mobile, o si sceglie il tipo di contenitore più appropriato tra quelli disponibili nella libreria standard. Se questo rende il programma sufficientemente veloce, si termina l'ottimizzazione.
    5. **Ottimizzazione dipendente dalla piattaforma software**. Nei colli di bottiglia, si adottano tecniche di ottimizzazione dipendenti sia dal linguaggio di programmazione che dalla piattaforma software, ma indipendenti dalla piattaforma hardware. Per esempio, si sfruttano le opzioni di compilazione, le direttive *pragma* di compilazione, le estensioni al linguaggio offerte da un particolare compilatore, si

usano librerie non-standard, o si chiama direttamente il sistema operativo. Se questo rende il programma sufficientemente veloce, si termina l'ottimizzazione.

6. **Ottimizzazione dipendente dalla piattaforma hardware**. Nei colli di bottiglia si adottano tecniche di ottimizzazione dipendenti dalla piattaforma hardware, cioè o istruzioni macchina che esistono solo su una particolare famiglia di processori, o costrutti ad alto livello che, pur essendo eseguibili su qualunque processore, risultano vantaggiose solo su alcuni tipi di processore.

Questo processo di sviluppo segue due criteri:

- **Principio delle rese calanti**. Le ottimizzazioni che danno grandi risultati con poco sforzo devono essere applicate per prime, in quanto così si minimizza il tempo necessario a raggiungere gli obiettivi prestazionali.
- **Principio della portabilità calante**. È meglio applicare prima le ottimizzazioni applicabili su più piattaforme, in quanto rimangono applicabili anche cambiando piattaforma, e in quanto sono di più facile comprensione per altri programmatori.

Nei rari casi di software che dovrà funzionare con più compilatori e su più sistemi operativi ma su un solo tipo di processore, le fasi 4.5 e 4.6 dovrebbero essere invertite.

Questa sequenza di fasi non va affatto interpretata come una sequenza a senso unico, cioè tale per cui una volta raggiunta una fase non si torna più alla fase precedente. In realtà ogni fase può avere successo o fallire. Se ha successo, si passa alla fase successiva, se fallisce si torna alla fase precedente.

Inoltre, un collaudo parziale delle prestazioni deve essere eseguito dopo ogni tentativo di ottimizzazione, per verificare se il tentativo risulta utile, e, in caso affermativo, per verificare se risulta risolutivo, cioè se sono necessarie altre ottimizzazioni.

Infine, dopo aver completato la fase di ottimizzazione, si devono ripetere sia il collaudo funzionale che il collaudo prestazionale completo, per garantire che la nuova versione ottimizzata del software non sia peggiorata né per la correttezza né per le prestazioni complessive.

Questo testo approfondisce solo tre delle fasi citate:

- La fase 2, limitatamente all'uso del linguaggio C++, nel capitolo "*Scrivere codice efficiente*".
- Alcune tecniche generali relative alla fase 4.3, con esempi in C++, nel capitolo "*Tecniche generali di ottimizzazione*".
- La fase 4.4, limitatamente all'uso del linguaggio C++, nel capitolo "*Ottimizzazione del codice*".

2.1 Notazioni terminologiche

Per **oggetto** si intende una regione allocata di memoria. In particolare, un dato associato a una variabile di un tipo fondamentale (come *bool*, *double*, *unsigned long*, o un puntatore) è un oggetto, così come lo è la struttura dati associata a un'istanza di una classe. A ogni variabile è associato un oggetto, la cui lunghezza si ottiene con l'operatore del C++ `sizeof`, ma un oggetto potrebbe non avere nessuna variabile associata a esso, oppure più variabili associate a esso. Per esempio, un puntatore è un oggetto, ma può puntare a un altro oggetto; tale oggetto puntato non è associato a nessuna variabile. D'altra parte, nel seguente codice, sia la variabile `a` che la variabile `b` sono associate allo stesso oggetto:

```
int a;
int& b = a;
```

Gli array, le strutture, e le istanze di classi sono oggetti che, se non sono vuoti, contengono sotto-oggetti. Perciò, tali oggetti verranno chiamati **oggetti composti** (sinonimo di *oggetti compositi* o *oggetti aggregati*).

Diciamo che un oggetto **possiede** un altro oggetto se la deallocazione del primo oggetto comporta la deallocazione del secondo. Per esempio, un oggetto `vector` non vuoto tipicamente contiene un puntatore a un buffer contenente gli elementi; la distruzione del `vector` comporta la distruzione di tale buffer, e quindi diciamo che questo buffer è *posseduto* dall'oggetto `vector`.

Alcune ottimizzazioni risultano utili solo per brevi sequenze di dati, altre per sequenze più lunghe. In seguito, si userà la seguente classificazione per le dimensioni degli oggetti:

- **Piccolissimo**: Non oltre 8 byte. Sta in uno o due registri a 32 bit o in un registro a 64 bit.
- **Piccolo**: Oltre 8 byte, ma non oltre 64 byte. Non sta in un registro del processore, ma sta in una linea della cache dei dati del processore, e può essere interamente indirizzato da istruzioni macchina molto compatte tramite uno scostamento rispetto all'indirizzo iniziale.
- **Medio**: Oltre 64 byte, ma non oltre 4096 byte. Non sta in una linea della cache dei dati del processore, e non può essere interamente indirizzato da istruzioni macchina compatte, ma sta nella cache dei dati di primo livello del processore, sta in una pagina di memoria virtuale, e sta in un cluster della memoria di massa.
- **Grande**: Oltre 4096 byte. Non sta nella cache dei dati di primo livello del processore, non sta in una sola pagina di memoria virtuale, e non sta in un solo cluster della memoria di massa.

Per esempio, un array di `double` è considerato *piccolissimo* solo se contiene esattamente un elemento, *piccolo* se ha da 2 a 8 elementi, *medio* se ne ha da 9 a 512, *grande* se ne ha più di 512.

Dato che ci sono architetture hardware molto variabili, i numeri suddetti sono solo indicativi. Tuttavia, tali numeri sono abbastanza realistici, e possono essere considerati seriamente come criteri per sviluppare del software che copra le principali architetture in modo piuttosto efficiente.

Capitolo 3 – Scrivere codice C++ efficiente

In questa sezione vengono proposte linee-guida per la programmazione in C++ finalizzate a evitare operazioni inefficienti e a preparare il codice sorgente a un'eventuale fase successiva di ottimizzazione, senza con questo rendere il codice meno sicuro né manutenibile.

Tali linee-guida potrebbero non dare alcun vantaggio prestazionale, ma molto probabilmente non danno neanche svantaggi, e quindi le si può applicare senza preoccuparsi del loro impatto sulle prestazioni. Si consiglia di abituarsi ad adottare sempre tali linee-guida, anche nelle porzioni di codice che non hanno particolari requisiti di efficienza.

1. Costrutti che migliorano le prestazioni
2. Costrutti che peggiorano le prestazioni
3. Costruzioni e distruzioni
4. Allocazioni e deallocazioni
5. Accesso alla memoria
6. Uso dei thread

3.1 Costrutti che migliorano le prestazioni

Alcuni costrutti del linguaggio C++, se usati opportunamente, permettono di aumentare la velocità del software risultante.

In questa sezione si presentano le linee-guida per approfittare di tali costrutti.

3.1.1 I tipi più efficienti

Quando definisci un oggetto per memorizzare un numero intero, usa il tipo `int` o il tipo `unsigned int`, a meno che sia necessario un tipo più lungo; quando definisci un oggetto per memorizzare un carattere, usa il tipo `char`, a meno che serva il tipo `wchar_t`; e quando definisci un oggetto per memorizzare un numero a virgola mobile, usa il tipo `double`, a meno che sia necessario il tipo `long double`. Se l'oggetto composto risultante è medio o grande, sostituisci tutti i tipi interi con il tipo intero più piccolo in grado di contenerlo, ma senza usare i *bit-field*, e sostituisci tutti i tipi a virgola mobile con il tipo `float`, a meno che sia necessaria maggiore precisione.

I tipi `int` e `unsigned int` sono per definizione quelli più efficienti su qualunque piattaforma.

Il tipo `double` è efficiente quanto il tipo `float`, ma è più preciso.

Alcuni tipi di processore elaborano più velocemente gli oggetti di tipo `signed char`, mentre altri elaborano più velocemente gli oggetti di tipo `unsigned char`. Pertanto, sia in C che in C++ è stato introdotto il tipo `char`, diverso dal tipo `signed char`, che corrisponde al tipo di carattere elaborato più velocemente dal processore target.

Il tipo `char` può contenere solo piccoli insiemi di caratteri; tipicamente fino a un massimo di 255 caratteri distinti. Per memorizzare set di caratteri più grandi, si deve ricorrere al tipo `wchar_t`, che ovviamente è meno efficiente.

Nel caso di numeri contenuti in un oggetto composto di media o grande dimensione, o in una collezione che si presume sarà di media o grande dimensione, è meglio minimizzare la dimensione in byte dell'oggetto composto o della collezione. Questo si può fare sostituendo gli `int` con `short` o con `signed char`, sostituendo gli `unsigned int` con `unsigned short` o con `unsigned char`, e sostituendo i `double` con `float`. Per esempio, per memorizzare un numero intero che può essere compreso tra 0 e 1000, si può usare un `unsigned short`, mentre per memorizzare un numero intero che può essere compreso tra -100 e 100, si può usare un `signed char`.

I bit-field contribuirebbero a minimizzare la dimensione in byte dell'oggetto o della collezione, ma la loro elaborazione introduce un rallentamento che potrebbe essere eccessivo; pertanto, posponi la loro introduzione alla fase di ottimizzazione.

3.1.2 Gli oggetti-funzione

Invece di passare come argomento di funzione un puntatore a funzione, passa un oggetto-funzione, o, se stai usando lo standard C++0x, un'espressione lambda.

Per esempio, se hai il seguente array di strutture:

```
struct S {
    int a, b;
};
S arr[n_voci];
```

e vuoi ordinarlo in base al campo `b`, potresti definire la seguente funzione di confronto:

```
bool confronta(const S& s1, const S& s2) {
    return s1.b < s2.b;
}
```

e passarla all'algoritmo `std::sort`:

```
std::sort(arr, arr + n_voci, confronta);
```

Ma probabilmente è più efficiente definire la seguente classe di oggetto-funzione (nota anche come *funtore*):

```
struct Comparatore {
    bool operator()(const S& s1, const S& s2) const {
        return s1.b < s2.b;
    }
};
```

e passarne un'istanza temporanea all'algoritmo `std::sort`:

```
std::sort(arr, arr + n_voci, Comparatore());
```

Le funzioni degli oggetti-funzione di solito sono espansi *inline*, e perciò sono altrettanto efficienti quanto il codice scritto sul posto, mentre le funzioni passate per puntatore vengono raramente espanse *inline*.

Le espressioni lambda sono implementate come oggetti-funzione, e quindi hanno le loro stesse prestazioni.

3.1.3 Funzioni `qsort` e `bsearch`

Invece delle funzioni della libreria standard del C `qsort` e `bsearch`, usa le funzioni della libreria standard del C++ `std:sort` e `std:lower_bound`.

Le prime due funzioni richiedono necessariamente un puntatore a funzione, mentre le seconde due possono usare un oggetto-funzione (o, usando C++0x, un'espressione lambda). I puntatori a funzione spesso non sono espansi *inline* e sono quindi meno efficienti degli oggetti-funzione, che sono quasi sempre espansi *inline*.

3.1.4 Collezioni incapsulate

Incapsula in apposite classi le collezioni accessibili da più unità di compilazione, in modo da garantire l'intercambiabilità di implementazione.

In fase di progettazione, è difficile decidere quale struttura dati avrà prestazioni ottimali nell'uso effettivo dell'applicazione. In fase di ottimizzazione, si può misurare se cambiando il tipo di un contenitore, per esempio passando da `std::vector` a `std::list`, si ottengono prestazioni migliori. Tuttavia, tale cambio di implementazione comporta la modifica di gran parte del codice sorgente che utilizza direttamente il contenitore che ha cambiato di tipo.

Se una collezione è privata a una sola unità di compilazione, tale modifica avrà impatto solamente sul codice sorgente contenuto in questa unità, e quindi non è necessario incapsulare tale collezione. Se invece quella collezione non è privata, cioè è accessibile direttamente da altre unità di compilazione, in futuro ci potrà essere una quantità enorme di codice che dovrà essere modificata a fronte di tale cambio di tipo. Quindi, per rendere fattibile in tempi ragionevoli tale ottimizzazione, si deve incapsulare la collezione in una classe la cui interfaccia non cambia al cambiare dell'implementazione del contenitore.

I contenitori STL perseguono già questo principio, ma alcune operazioni sono disponibili solo per alcuni contenitori (come l'`operator[]`, che esiste per `std::vector` ma non per `std::list`).

3.1.5 Uso dei contenitori STL

Nell'uso dei contenitori STL, se più espressioni equivalenti hanno le stesse prestazioni, scegli l'espressione più generale.

Per esempio, chiama `a.empty()` invece di `a.size() == 0`, chiama `iter != a.end()` invece di `iter < a.end()`, e chiama `distance(iter1, iter2)` invece di `iter2 - iter1`. Le prime espressioni sono valide per ogni tipo di contenitore, mentre le seconde solamente per alcuni tipi, e le prime non sono meno efficienti delle seconde.

Purtroppo, non è sempre possibile scrivere del codice egualmente valido ed efficiente per ogni tipo di contenitore. Tuttavia, riducendo il numero di istruzioni dipendenti dal tipo di contenitore, si riduce anche il numero di istruzioni da modificare qualora, in fase di ottimizzazione, il tipo del contenitore venisse cambiato.

3.1.6 Scelta del contenitore di default

Nella scelta di un contenitore a lunghezza variabile, in caso di dubbio, scegli un `vector`.

Fino a 8 elementi, il `vector` è il contenitore a lunghezza variabile più efficiente per qualunque operazione.

Per insiemi più grandi, altri contenitori possono diventare sempre più efficienti per alcune operazioni, ma il `vector` rimane quello che ha minore occupazione di memoria (purché non ci sia capacità in eccesso), e maggiore località di riferimento dei dati.

3.1.7 Funzioni espanse *inline*

Se usi compilatori che consentono l'ottimizzazione dell'intero programma e l'espansione *inline* automatica delle funzioni, usa tali opzioni, e non dichiarare `inline` nessuna funzione. Se tali funzionalità del compilatore non fossero disponibili, dichiara `inline` nei file di intestazione solo le funzioni che contengono non più di tre righe di codice, tra le quali non ci siano cicli.

Le funzioni espanse *inline* non hanno il costo della chiamata di funzione, che è tanto più grande quanti più sono gli argomenti della funzione. Inoltre, dato che il codice è vicino a quello del chiamante, hanno migliore località di riferimento del codice. Infine, dato che il codice intermedio delle funzioni espanse *inline* si fonde con quello del chiamante, tale codice può essere facilmente ottimizzato dal compilatore.

Espandere *inline* una funzione estremamente piccola, cioè un semplice assegnamento o una semplice istruzione `return`, comporta addirittura una riduzione del codice macchina generato.

Tuttavia, ogni volta che una routine che contiene una quantità notevole di codice viene espansa *inline*, il suo codice macchina viene duplicato, e quindi la dimensione complessiva del programma aumenta, peggiorando la località di riferimento del codice.

Inoltre, il codice espanso *inline* è più difficile da analizzare con un profiler. Se una funzione non espansa *inline* è un collo di bottiglia, viene individuata dal profiler. Ma se quella funzione viene espansa *inline* in tutti i punti in cui è chiamata, il suo tempo di esecuzione viene sparpagliato fra tutte le funzioni che la chiamano.

In fase di ottimizzazione, tra le funzioni non piccolissime, solo quelle critiche per la velocità dovranno essere dichiarate `inline`.

3.1.8 Rappresentazione di simboli

Per rappresentare dei simboli interni, usa degli enumerati, invece di stringhe.

Per esempio, invece del seguente codice:

```
const char* direzioni[] = { "Nord", "Sud", "Est", "Ovest" };
```

scrivi il seguente codice:

```
enum direzioni { Nord, Sud, Est, Ovest };
```

Un enumerato è implementato come un intero. Tipicamente, rispetto ad un intero, una stringa occupa più spazio, ed è più lenta da copiare e da confrontare.

3.1.9 Istruzioni `if` e `switch`

Se devi confrontare un valore intero con una serie di valori costanti, invece di una serie di istruzioni `if`, usa un'istruzione `switch`.

Per esempio, invece del seguente codice:

```
if (a[i] == 1) f();
else if (a[i] == 2) g();
else if (a[i] == 5) h();
```

scrivi il seguente codice:

```
switch (a[i]) {
case 1: f(); break;
case 2: g(); break;
case 5: h(); break;
}
```

I compilatori possono sfruttare la regolarità delle istruzioni `switch` per applicare alcune ottimizzazioni, in particolare se viene applicata la linea-guida "Valori dei casi di istruzioni `switch`" di questa sezione.

3.1.10 Valori dei casi di istruzioni `switch`

Come costanti per i casi delle istruzioni `switch`, usa sequenze compatte di valori, cioè senza lacune o con poche piccole lacune.

I compilatori ottimizzanti, quando compilano un'istruzione `switch` i cui valori dei casi costituiscono la maggior parte dei valori interi compresi in un intervallo, invece di generare una sequenza di istruzioni `if`, generano una *jump-table*, ossia un array degli indirizzi a cui inizia il codice di ogni caso, e per eseguire l'istruzione `switch`, usano tale tabella per saltare al codice associato al numero del caso.

Per esempio, il seguente codice C++:

```
switch (i) {
case 10:
case 13:
    funz_a();
    break;
case 11:
    funz_b();
    break;
}
```

probabilmente genera del codice macchina corrispondente al seguente pseudo-codice:

```
// N.B.: Questo non è codice C++
static indirizzo a[] = { caso_a, caso_b, fine, caso_a };
```

```
unsigned int indice = i - 10;
if (indice > 3) goto fine;
goto a[indice];
caso_a: funz_a(); goto fine;
caso_b: funz_b();
fine:
```

Invece, il seguente codice C++:

```
switch (i) {
case 100:
case 130:
    funz_a();
    break;
case 110:
    funz_b();
    break;
}
```

probabilmente genera del codice macchina corrispondente al seguente codice:

```
if (i == 100) goto caso_a;
if (i == 130) goto caso_a;
if (i == 110) goto caso_b;
goto fine;
caso_a: funz_a(); goto fine;
caso_b: funz_b();
fine:
```

Per così pochi casi, probabilmente non ci sono molte differenze tra le due situazioni, ma con l'aumentare del numero di casi, il primo codice diventa più efficiente, in quanto esegue un solo *goto calcolato* invece di una serie di `if`.

3.1.11 Ordine dei casi dell'istruzione `switch`

Nelle istruzioni `switch`, poni prima i casi più tipici.

Se il compilatore non generasse la *jump-table*, i casi verrebbero confrontati in ordine di comparizione; pertanto, nei casi più tipici verrebbero fatti meno confronti.

3.1.12 Raggruppamento di più array in un solo array di strutture

Invece di elaborare in parallelo due o più array della stessa lunghezza, elabora un solo array di oggetti composti.

Esempio: Invece del seguente codice:

```
const int n = 10000;
double a[n], b[n], c[n];
for (int i = 0; i < n; ++i) {
    a[i] = b[i] + c[i];
}
```

scrivi il seguente codice:

```
const int n = 10000;
struct { double a, b, c; } s[n];
for (int i = 0; i < n; ++i) {
    s[i].a = s[i].b + s[i].c;
}
```

In tal modo, i dati da elaborare insieme sono più vicini tra di loro in memoria, e questo permette di ottimizzare l'uso della cache dei dati, e di indirizzare tali dati con istruzioni più compatte, che quindi ottimizzano anche l'uso della cache del codice.

3.1.13 Raggruppamento di argomenti di funzione

Se una funzione riceve almeno sei argomenti, e viene chiamata in un ciclo con gli stessi valori tranne al più due, prima del ciclo crea una struttura che contiene tutti gli argomenti, e passa alla funzione tale struttura.

Per esempio, invece del seguente codice:

```
for (int i = 0; i < 1000; ++i) {
    f(i, a1, a2, a3, a4, a5, a6, a7, a8);
}
```

scrivi il seguente codice:

```
struct {
    int i;
    tipo a1, a2, a3, a4, a5, a6, a7, a8;
} s;
s.a1 = a1; s.a2 = a2; s.a3 = a3; s.a4 = a4;
s.a5 = a5; s.a6 = a6; s.a7 = a7; s.a8 = a8;
for (int i = 0; i < 1000; ++i) {
    s.i = i;
    f(s);
}
```

Se una funzione riceve pochi argomenti, questi vengono posti direttamente nei registri, e quindi il loro passaggio è molto veloce; ma se gli argomenti non sono pochi, devono essere posti nello stack ad ogni chiamata, anche se non sono cambiati dall'iterazione precedente.

Se invece si passa alla funzione solo l'indirizzo di una struttura, questo indirizzo viene probabilmente posto in un registro, e, dopo l'inizializzazione della struttura, solamente i campi della struttura che sono modificati tra chiamate successive devono essere assegnati.

3.1.14 Uso di funzioni membro di contenitori

Per cercare un elemento in un contenitore, usa una funzione membro del contenitore, invece di un algoritmo STL.

Se è stata creata una tale funzione membro specifica, quando esisteva già un algoritmo STL generico, è solo perché tale funzione membro è più efficiente.

Per esempio, per cercare in un oggetto `std::set`, si può usare l'algoritmo generico `std::find`, o la funzione membro `std::set::find`; ma il primo ha complessità lineare (O(n)), mentre la seconda ha complessità logaritmica (O(log(n))).

3.1.15 Ricerca in sequenze ordinate

Per cercare un elemento in una sequenza ordinata, usa gli algoritmi `lower_bound`, `upper_bound`, `equal_range`, o `binary_search`.

Tutti i citati algoritmi, dato che usano la ricerca binaria, avente complessità logaritmica (O(log(n))), sono più veloci dell'algoritmo `std::find`, che usa la ricerca sequenziale, avente complessità lineare (O(n)).

3.2 Costrutti che peggiorano le prestazioni

Rispetto al linguaggio C, il linguaggio C++ aggiunge alcuni costrutti, il cui utilizzo peggiora l'efficienza.

Alcuni di essi sono piuttosto efficienti, e quindi li si può usare tranquillamente quando servono, ma si dovrebbe evitare di pagarne il costo quando non li si usa.

Altri costrutti sono invece alquanto inefficienti, e devono quindi essere usati con parsimonia.

In questa sezione si presentano le linee-guida per evitare i costi dei costrutti C++ che peggiorano le prestazioni.

3.2.1 L'operatore `throw`

Chiama l'operatore `throw` solamente quando vuoi avvisare un utente del fallimento del comando corrente.

Il sollevamento di una eccezione ha un costo molto elevato, rispetto a quello di una chiamata di funzione. Sono migliaia di cicli di processore. Se tale operazione viene effettuata solamente ogni volta che un messaggio viene mostrato all'utente o scritto in un file di log, si ha la garanzia che non verrà eseguita troppo spesso senza che ce ne si accorga. Invece, se si sollevano eccezioni per scopi algoritmici, anche se tali operazioni sono state pensate inizialmente per essere eseguite raramente, potrebbero finire per essere eseguite troppo frequentemente.

3.2.2 Funzioni membro `static`

In ogni classe, dichiara `static` ogni funzione membro che non accede ai membri non-`static` di tale classe.

In altre parole, dichiara `static` tutte le funzioni membro che puoi.

In questo modo, non viene passato l'argomento implicito `this`.

3.2.3 Funzioni membro `virtual`

In ogni classe, definisci `virtual` il distruttore se e solo se la classe contiene almeno un'altra funzione membro `virtual`, e, a parte i costruttori e il distruttore, definisci `virtual` solamente le funzioni membro che ritieni possa essere utile ridefinire.

Classi che contengono almeno una funzione membro `virtual` occupano un po' più di spazio delle classi che non ne contengono. Le istanze delle classi che contengono almeno una funzione membro

`virtual` occupano un po' più di spazio (tipicamente, un puntatore ed eventualmente dello spazio di allineamento) e la loro costruzione richiede un po' più di tempo (tipicamente, per impostare tale puntatore) rispetto alle istanze di classi prive di funzioni membro `virtual`.

Inoltre, ogni funzione membro `virtual` è più lenta da chiamare di un'identica funzione membro non-`virtual`.

3.2.4 Derivazione `virtual`

Usa la derivazione `virtual` solo quando due o più classi devono condividere la rappresentazione di una classe base comune.

Per esempio, considera le seguenti definizioni di classe:

```
class A { ... };
class B1: public A { ... };
class B2: public A { ... };
class C: public B1, public B2 { ... };
```

Con tali definizioni, ogni oggetto di classe C contiene due oggetti distinti di classe A, uno ereditato dalla classe base B1, e l'altro ereditato dalla classe base B2.

Questo non costituisce un problema se la classe A non ha nessuna variabile membro non-`static`.

Se invece tale classe A contiene qualche variabile membro, e si intende che tale variabile membro debba essere unica per ogni istanza di classe C, si deve usare la derivazione `virtual`, nel seguente modo:

```
class A { ... };
class B1: virtual public A { ... };
class B2: virtual public A { ... };
class C: public B1, public B2 { ... };
```

Questa situazione è l'unica in cui è necessaria la derivazione `virtual`.

Se si usa la derivazione `virtual`, le funzioni membro della classe A sono un po' più lente da chiamare su un oggetto di classe C.

3.2.5 Template di classi polimorfiche

Non definire template di classi polimorfiche.

In altre parole, non usare le parole-chiave "`template`" e "`virtual`" nella stessa definizione di classe.

I template di classe, ogni volta che vengono istanziati, producono una copia di tutte le funzioni membro utilizzate, e se tali classi contengono funzioni virtuali, vengono replicate anche le *vtable* e le informazioni *RTTI*. Questi dati ingrandiscono eccessivamente il programma.

3.2.6 Uso di deallocatori automatici

Usa un gestore della memoria basato su garbage-collection o un tipo di smart-pointer dotato di reference-count (come lo `shared_ptr` della libreria Boost) solamente se ne dimostri l'opportunità per il caso specifico.

La garbage collection, cioè il recupero automatico della memoria non più referenziata, fornisce la comodità di non doversi occupare della deallocazione della memoria, e previene i *memory leak*. Tale funzionalità non viene fornita dalla libreria standard, ma viene fornita da librerie non-standard. Tuttavia, tale tecnica di gestione della memoria potrebbe produrre prestazioni peggiori rispetto alla deallocazione esplicita (cioè chiamando l'operatore `delete`).

La libreria standard del C++98 contiene un solo smart-pointer, l'`auto_ptr`, che è efficiente. Altri smart-pointer sono forniti da librerie non-standard, come Boost, o verranno forniti dal C++0x. Tra di essi, gli smart-pointer basati su reference-count, come lo `shared_ptr` di Boost, sono meno efficienti dei puntatori semplici, e quindi devono essere usati solo nei casi in cui se ne dimostra la necessità. In particolare, compilando con l'opzione di gestione del multithreading, tali puntatori hanno pessime prestazioni nella creazione, distruzione e copia dei puntatori, in quanto devono garantire la mutua esclusione delle operazioni.

Normalmente bisognerebbe, in fase di progettazione, cercare di assegnare ogni oggetto allocato dinamicamente ad un proprietario, cioè a un altro oggetto che lo possiede. Solo quando tale assegnazione risulta difficile, in quanto più oggetti tendono a rimpallarsi la responsabilità di distruggere l'oggetto, risulta opportuno usare uno smart-pointer con reference-count per gestire tale oggetto.

3.2.7 Il modificatore `volatile`

Definisci `volatile` solamente quelle variabili che vengono modificate in modo asincrono da dispositivi hardware o da più thread.

L'uso del modificatore `volatile` impedisce al compilatore di allocare una variabile in un registro, anche per un breve periodo. Questo garantisce che tutti i dispositivi e tutti i thread *vedano* la stessa variabile, ma rende molto più lente le operazioni che manipolano tale variabile.

3.3 Costruzioni e distruzioni

La costruzione e la distruzione di un oggetto richiedono tempo, particolarmente se tale oggetto, direttamente o indirettamente, possiede altri oggetti.

In questa sezione vengono proposte linee-guida per ridurre il numero di creazioni di oggetti, e quindi delle loro corrispondenti distruzioni.

3.3.1 Ambito delle variabili

Dichiara le variabili il più tardi possibile.

Dichiarare una variabile il più tardi possibile significa sia dichiararla nell'ambito (in inglese, *scope*) più stretto possibile, sia dichiararla il più avanti possibile entro quell'ambito. Essere nell'ambito più stretto possibile comporta che se tale ambito non viene mai raggiunto, l'oggetto associato alla variabile non viene mai costruito né distrutto. Dichiarare una variabile il più avanti possibile

all'interno di un ambito comporta che se prima di tale dichiarazione c'è un'uscita prematura, tramite `return` o `break` o `continue`, l'oggetto associato alla variabile non viene mai costruito né distrutto.

Inoltre, spesso all'inizio di una routine non si ha un valore appropriato per inizializzare l'oggetto associato alla variabile, e quindi si è costretti a inizializzarla con un valore di default, e poi assegnarle il valore appropriato. Se invece la si dichiara quando si ha a disposizione il valore appropriato, la si può inizializzare con tale valore senza bisogno di fare un successivo assegnamento, come suggerito dalla linea-guida "Inizializzazioni" in questa sezione.

3.3.2 Inizializzazioni

Usa inizializzazioni invece di assegnamenti. In particolare, nei costruttori usa le liste di inizializzazione.

Per esempio, invece di scrivere:

```
string s;
...
s = "abc"
```

scrivi:

```
string s("abc");
```

Anche se un'istanza di una classe non viene inizializzata esplicitamente, viene comunque inizializzata automaticamente dal costruttore di default.

Chiamare il costruttore di default seguito da un assegnamento di un valore può essere meno efficiente che chiamare solo un costruttore con tale valore.

3.3.3 Operatori di incremento/decremento

Usa gli operatori prefissi di incremento (`++`) o decremento (`--`) invece dei corrispondenti operatori postfissi, se il valore dell'espressione non viene usato.

Se l'oggetto incrementato è di un tipo fondamentale, non ci sono differenze tra le due forme, ma se si tratta di un tipo composto, l'operatore postfisso comporta la creazione di un oggetto temporaneo, mentre l'operatore prefisso no.

Siccome ogni oggetto che è attualmente di un tipo fondamentale potrebbe diventare in futuro di una classe, è bene usare sempre l'operatore che è comunque più efficiente.

Tuttavia, se il valore dell'espressione formata dall'operatore di incremento o decremento viene usata in un'espressione più grande, potrebbe essere opportuno usare l'operatore postfisso.

3.3.4 Operatori compositi di assegnamento

Usa gli operatori compositi di assegnamento (come in `a += b`) invece degli operatori semplici combinati con operatori di assegnamento (come in `a = a + b`).

Per esempio, invece del seguente codice:

```
string s1("abc");
string s2 = s1 + " " + s1;
```

scrivi il seguente codice:

```
string s1("abc");
string s2 = s1;
s2 += " ";
s2 += s1;
```

Tipicamente un operatore semplice, crea un oggetto temporaneo. Nell'esempio, gli operatori + creano stringhe temporanee, la cui creazione e distruzione richiede tempo.

Al contrario, il codice equivalente che usa l'operatore += non crea oggetti temporanei.

3.3.5 Passaggio di argomenti alle funzioni

Quando devi passare un argomento `x` di tipo `T` a una funzione, usa il seguente criterio:

- **Se `x` è un argomento di solo input,**
  - **se `x` può essere nullo,**
    - **passalo per puntatore a costante (`const T* x`),**
  - **altrimenti, se `T` è un tipo fondamentale o un iteratore o un oggetto-funzione,**
    - **passalo per valore (`T x`) o per valore costante (`const T x`),**
  - **altrimenti,**
    - **passalo per riferimento a costante (`const T& x`),**
- **altrimenti, cioè se `x` è un argomento di solo output o di input/output,**
  - **se `x` può essere nullo,**
    - **passalo per puntatore a non-costante (`T* x`),**
  - **altrimenti,**
    - **passalo per riferimento a non-costante (`T& x`).**

Il passaggio per riferimento è più efficiente del passaggio per puntatore in quanto facilita al compilatore l'eliminazione della variabile, e in quanto il chiamato non deve verificare se il riferimento è valido o nullo; tuttavia, il puntatore ha il pregio di poter rappresentare un valore nullo, ed è più efficiente passare solo un puntatore, che un riferimento a un oggetto insieme a un booleano che indica se tale riferimento è valido.

Per oggetti che possono essere contenuti in uno o due registri, il passaggio per valore è più efficiente o ugualmente efficiente del passaggio per riferimento, in quanto tali oggetti possono essere contenuti in registri e non hanno livelli di indirettezza, pertanto questo è il modo più efficiente di passare oggetti sicuramente piccoli, come i tipi fondamentali, gli iteratori e gli oggetti-funzione. Per oggetti più grandi di due registri, il passaggio per riferimento è più efficiente del passaggio per valore, in quanto il passaggio per valore comporta la copia di tali oggetti nello stack.

Un oggetto composito veloce da copiare potrebbe essere efficientemente passato per valore, ma, a meno che si tratti di un iteratore o di un oggetto-funzione, per i quali si assume l'efficienza della copia, tale tecnica è rischiosa, in quanto l'oggetto potrebbe diventare in futuro più lento da copiare. Per esempio, se un oggetto di classe `Point` contiene solo due `float`, potrebbe essere efficientemente passato per valore; ma se in futuro si aggiungesse un terzo `float`, o se i due `float` diventassero due `double`, potrebbe diventare più efficiente il passaggio per riferimento.

3.3.6 Dichiarazione `explicit`

Dichiara `explicit` tutti i costruttori che possono ricevere un solo argomento, eccetto i costruttori di copia delle classi concrete.

I costruttori non-`explicit` possono essere chiamati automaticamente dal compilatore che esegue una conversione automatica. L'esecuzione di tale costruttore può richiedere molto tempo.

Se tale conversione è resa obbligatoriamente esplicita, e il nome della classe destinazione non viene specificato nel codice, il compilatore potrebbe scegliere un'altra funzione in overload, evitando così di chiamare il costoso costruttore, oppure segnalare l'errore e costringere il programmatore a scegliere un'altra strada per evitare la chiamata al costruttore.

Per i costruttori di copia delle classi concrete si deve fare eccezione, per consentirne il passaggio per valore. Per le classi astratte, anche i costruttori di copia possono essere dichiarati `explicit`, in quanto, per definizione, le classi astratte non si possono istanziare, e quindi gli oggetti di tale tipo non dovrebbero mai essere passati per valore.

3.3.7 Operatori di conversione

Dichiara operatori di conversione solamente per mantenere la compatibilità con una libreria obsoleta (in C++0x, dichiarali `explicit`).

Gli operatori di conversione consentono conversioni implicite, e quindi incorrono nello stesso problema dei costruttori impliciti, descritto nella linea-guida "Dichiarazione `explicit`" di questa sezione.

Se tali conversioni sono necessarie, fornisci invece una funzione membro equivalente, che può essere chiamata solo esplicitamente.

L'unico utilizzo che rimane accettabile per gli operatori di conversione si ha quando si vuole far convivere una nuova libreria con un'altra vecchia libreria simile. In tal caso, può essere comodo avere operatori che convertono automaticamente gli oggetti dai tipi della vecchia libreria ai tipi della nuova libreria e viceversa.

3.3.8 Idioma *Pimpl*

Usa l'idioma *Pimpl* solamente quando vuoi rendere il resto del programma indipendente dall'implementazione di una classe.

L'idioma *Pimpl* (che significa Puntatore a IMPLementazione) consiste nel memorizzare nell'oggetto solamente un puntatore alla struttura che contiene tutte le informazioni utili di tale oggetto.

Il vantaggio principale di tale idioma è che velocizza la compilazione incrementale del codice, cioè rende meno probabile che una piccola modifica ai sorgenti comporti la necessità di ricompilare grandi quantità di codice.

Tale idioma consente anche di velocizzare alcune operazioni, come lo `swap` tra due oggetti, ma in generale rallenta gli accessi ai dati dell'oggetto a causa del livello di indirettezza, e provoca un'allocazione aggiuntiva per ogni creazione e copia di tale oggetto. Quindi non dovrebbe essere usato per classi le cui funzioni membro pubbliche sono chiamate frequentemente.

3.3.9 Iteratori e oggetti-funzione

Fa' in modo che gli oggetti iteratori o oggetti-funzione siano piccolissimi e che non allochino memoria dinamica.

Gli algoritmi di STL passano tali oggetti per valore. Pertanto, se la loro copia non è estremamente efficiente, gli algoritmi STL vengono rallentati.

3.4 Allocazioni e deallocazioni

L'allocazione e la deallocazione dinamiche di memoria sono operazioni molto lente, a confronto dell'allocazione e della deallocazione automatiche di memoria. In altre parole, lo *heap* è molto più lento dello *stack*.

Inoltre, tale tipo di allocazione comporta uno spreco di spazio per ogni allocazione, genera frammentazione della memoria virtuale, e produce una scarsa località dei dati, con conseguente scadente utilizzo sia delle cache dei dati del processore che dello spazio di memoria virtuale.

L'allocazione/deallocazione dinamica di memoria veniva fatta in linguaggio C usando le funzioni `malloc` e `free` della libreria standard. In C++, pur essendo ancora disponibili tali funzioni, normalmente a tale scopo si usano gli operatori `new`, `new[]`, `delete`, e `delete[]`.

Ovviamente, un modo di ridurre le allocazioni è ridurre il numero di oggetti costruiti, e quindi la sezione "Costruzioni e distruzioni" di questo capitolo serve indirettamente anche allo scopo di questa sezione.

Tuttavia, qui si presenteranno linee-guida per ridurre il numero di allocazioni di memoria per un dato numero di chiamate all'operatore `new`.

3.4.1 Array di lunghezza fissa

Se un array statico o non grande ha lunghezza costante, invece di usare un oggetto `vector`, usa un array del C, o un oggetto `array` della libreria Boost.

I `vector` memorizzano i dati in un buffer allocato dinamicamente, mentre le altre soluzioni proposte allocano i dati nell'oggetto stesso. Questo consente di evitare allocazioni/deallocazioni di memoria dinamica e di favorire la località dei dati.

Se l'array è grande, tali vantaggi diminuiscono, e invece risulta più importante evitare di usare troppo spazio sullo stack.

3.4.2 Allocatore a blocchi

Se devi allocare numerosi blocchi di memoria della stessa dimensione, assicurati di usare un allocatore a blocchi.

Un *allocatore a blocchi* (detto anche *allocatore a pool*) alloca blocchi di memoria medi o grandi, e fornisce servizi di allocazione/deallocazione di blocchi più piccoli di dimensione costante. Offre alta velocità di allocazione/deallocazione, bassa frammentazione della memoria, uso efficiente delle cache dei dati e della memoria virtuale.

In particolare, un allocatore di questo tipo migliora notevolmente le prestazioni dei contenitori `std::list`, `std::set`, `std::multi_set`, `std::map`, e `std::multi_map`.

Se la tua implementazione della libreria standard non usa già un allocatore a blocchi per questi contenitori, dovresti procurartene uno (per esempio, questo: http://www.codeproject.com/KB/stl/blockallocator.aspx), e specificarlo come parametro di template per le istanze di tali template di contenitori.

3.4.3 Aggiunta di elementi a collezione

Quando aggiungi elementi in fondo a una collezione, usa `push_back` per aggiungere un singolo elemento, usa `insert` per aggiungere una sequenza, e usa `back_inserter` per fare in modo che un algoritmo STL aggiunga elementi a una sequenza.

La funzione `push_back` garantisce un tempo lineare ammortizzato, in quanto, nel caso dei `vector`, ingrandisce esponenzialmente la capacità.

La classe `back_inserter` chiama internamente la funzione `push_back`.

La funzione `insert` permette di inserire in modo ottimizzato un'intera sequenza, e quindi una sola chiamata di questo tipo è più veloce di numerose chiamate a `push_back`.

3.5 Accesso alla memoria

Questa sezione presenta le linee-guida per migliorare le prestazioni di accesso alla memoria principale, facendo buon uso delle memorie cache del processore, e dello swapping su hard disk del gestore di memoria virtuale del sistema operativo.

3.5.1 Ordine di accesso alla memoria

Accedi alla memoria in ordine crescente. In particolare:

- **scandisci gli array in ordine crescente;**
- **scandisci gli array multidimensionali usando gli indice più a destra per i cicli più interni;**
- **nei costruttori delle classi e negli operatori di assegnamento (`operator=`) accedi alle variabili membro nell'ordine in cui sono dichiarate nella classe.**

La cache dei dati ottimizza gli accessi alla memoria in ordine sequenziale crescente.

Quando si itera su un array multidimensionale, il ciclo più interno dovrebbe iterare sull'ultimo indice, il ciclo appena più esterno dovrebbe iterare sul penultimo indice, e così via. In tal modo, è garantito che le celle vengono elaborate nello stesso ordine in cui si trovano in memoria. Per esempio, il seguente codice è ottimizzato:

```
float a[num_livelli][num_righe][num_colonne];
for (int liv = 0; liv < num_livelli; ++liv) {
    for (int r = 0; r < num_righe; ++r) {
         for (int c = 0; c < num_colonne; ++c) {
            a[liv][r][c] += 1;
        }
    }
}
```

3.5.2 Allineamento di memoria

Lascia l'allineamento di memoria suggerito dal compilatore.

I compilatori attivano di default un criterio di allineamento dei tipi fondamentali, per cui gli oggetti possono avere solamente indirizzi di memoria che sono un multiplo di fattori particolari. Tale criterio garantisce le massime prestazioni, ma può introdurre degli spazi inutilizzati (in inglese, *padding*) tra oggetti consecutivi.

Se per alcune strutture è necessario eliminare tali spazi, usa le direttiva *pragma* solamente intorno alle definizioni di tali strutture.

3.5.3 Raggruppamento di funzioni in unità di compilazione

Definisci nella stessa unità di compilazione tutte le funzioni membro di una classe, tutte le funzioni `friend` di tale classe, e tutte le funzioni delle classi `friend` di tale classe, a meno che il file risultante diventi scomodo da gestire per la sua dimensione eccessiva.

In tal modo, sia il codice macchina prodotto compilando tali funzioni sia i dati statici definiti in tali classi e funzioni avranno indirizzi vicini tra loro; inoltre, così si consente anche ai compilatori che non effettuano ottimizzazioni sull'intero programma di ottimizzare le chiamate tra tali funzioni.

3.5.4 Raggruppamento di variabili in unità di compilazione

Definisci ogni variabile globale nell'unità di compilazione in cui è usata più spesso.

In tal modo, tali variabili avranno indirizzi vicini tra loro e vicini a quelli delle variabili statiche definite in tale unità di compilazione; inoltre, così si consente anche ai compilatori che non effettuano ottimizzazioni sull'intero programma di ottimizzare l'accesso a tali variabili da parte delle funzioni che le usano maggiormente.

3.5.5 Funzioni e variabili private in unità di compilazione

Dichiara in un namespace anonimo le variabili e le funzioni globali a un'unità di compilazione, ma non usate da altre unità di compilazione.

In linguaggio C e anche in C++, tali variabili e funzioni possono essere dichiarate `static`.

Tuttavia, nel C++ moderno, l'uso di variabili e funzioni globali `static` è deprecato, e dovrebbe essere sostituito da variabili e funzioni dichiarate in un namespace anonimo.

In entrambi i casi, si dichiara al compilatore che tali identificatori non verranno usati da altre unità di compilazione. Questo permette anche ai compilatori che non effettuano ottimizzazioni sull'intero programma di ottimizzare l'utilizzo di tali variabili e funzioni.

3.6 Uso dei thread

3.6.1 Thread di lavoro

Ogni volta che in un'applicazione interattiva devi eseguire un compito che può richiedere più di una manciata di secondi, assegna tale compito a un apposito thread di calcolo di priorità più bassa del normale.

In tal modo, il thread principale si occupa solo di gestire l'interfaccia utente, ed è pronto a rispondere ad altri comandi. Assegnando al thread di calcolo priorità più bassa del normale, l'interfaccia utente rimane veloce quasi come se non ci fosse un'elaborazione in corso.

Questa linea-guida in realtà non aiuta a migliorare la velocità dell'applicazione, ma solo la sua responsività. Tuttavia, questo è percepito dagli utenti come un aumento di velocità.

3.6.2 Thread di lavoro multipli

In un sistema multicore, se riesci a suddividere un'elaborazione in più thread, usa tanti thread di calcolo quanti sono i core di processore.

In tal modo ogni core può elaborare un thread. Se i thread di calcolo fossero più dei processori, ci sarebbe contesa tra i thread, e questo rallenterebbe l'elaborazione. Il thread di interfaccia utente non rallenta, in quanto è pressoché inattivo.

3.6.3 Uso di librerie multi-threaded

Se sviluppi un'applicazione single-threaded, non usare librerie progettate per applicazioni multi-threaded.

Le tecniche per rendere thread-safe una libreria possono dover usare memoria e tempo. Se non usi i thread, evita di pagarne il costo.

3.6.4 Creazione di librerie multi-threaded

Se sviluppi una libreria, gestisci correttamente il caso in cui sia usata da applicazioni multi-threaded, ma ottimizza anche il caso in cui sia usata da applicazioni single-threaded.

Le tecniche per rendere thread-safe una libreria possono dover usare memoria e tempo. Se gli utenti della tua libreria non usano i thread, evita di fargliene pagare il costo.

3.6.5 Mutua esclusione

Usa primitive di mutua esclusione solo quando più thread accedono contemporaneamente agli stessi dati, e almeno uno degli accessi è in scrittura.

Le primitive di mutua esclusione richiedono tempo.

Se sei sicuro che in un dato intervallo di tempo nessun thread scrive in un'area di memoria, non c'è bisogno di sincronizzare gli accessi in lettura a tale area.

Capitolo 4 – Tecniche generali di ottimizzazione

In questa sezione vengono proposte alcune tecniche di ottimizzazione algoritmica di ampia utilizzabilità, e sostanzialmente indipendenti sia dal linguaggio di programmazione, che dalla piattaforma software e hardware.
Per alcune delle tecniche proposte viene mostrata un'implementazione in linguaggio C++.

1. Input/Output
2. Caching
3. Ordinamento
4. Altre tecniche

4.1 Input/Output

4.1.1 Formato binario

Invece di memorizzare i dati su file in formato testuale, memorizzali in formato binario.

In media, i numeri in formato binario occupano meno spazio dei numeri formattati, e quindi richiedono meno tempo per essere trasferiti dalla memoria al disco o viceversa, ma, soprattutto, se i dati vengono trasferiti nello stesso formato usato dal processore, non c'è bisogno di nessuna costosa conversione dal formato testuale al formato binario o viceversa.

Gli svantaggi del formato binario sono che i dati non sono facilmente leggibili e che tale formato può dipendere dall'architettura del processore.

4.1.2 File aperti

Invece di aprire e chiudere un file di utilizzo frequente ogni volta che ci accedi, aprilo solamente la prima volta che ci accedi, e chiudilo quando hai finito di usarlo.

Chiudere e riaprire un file di disco richiede un tempo variabile, ma approssimativamente lo stesso che ci vuole per leggere dai 15 ai 20 KB di dati dalla cache del disco.

Perciò, se devi accedere spesso a un file, puoi evitare questa inefficienza aprendo il file solamente una volta prima di accedervi, mantenerlo aperto spostando il gestore del file a un ambito più esterno, e chiudendo il file quando hai finito.

4.1.3 Buffer di I/O

Invece di fare molte operazioni di I/O su singoli oggetti piccoli o piccolissimi, fai operazioni di I/O su un buffer di 4 KB contenente molti oggetti.

Anche se le operazioni di I/O del supporto run-time sono bufferizzate, l'inefficienza di molte chiamate alle funzioni di I/O costa di più che copiare gli oggetti in un buffer.

I buffer grandi non hanno una buona località di riferimento dei dati.

4.1.4 Memory-mapped-file

Eccetto che in una sezione critica di un sistema real-time, se devi accedere a gran parte di un file binario in modo non-sequenziale, invece di accedervi ripetutamente con operazioni di *seek*, oppure di caricarlo tutto in un buffer dell'applicazione, usa un memory-mapped-file, se il tuo sistema operativo fornisce tale strumento.

Quando si deve accedere a gran parte di un file binario in modo non-sequenziale, ci sono due tecniche alternative standard:

- Aprire il file senza leggerne il contenuto; e ogni volta che si deve leggere un dato, saltare al punto di interesse usando una operazione di posizionamento nel file (*seek*), e leggere il dato usando un'operazione di lettura.
- Allocare un buffer grande quanto tutto il file, aprire il file, leggere tutto il contenuto del file nel buffer, chiudere il file; e ogni volta che si deve leggere un dato, cercarlo nel buffer.

Rispetto alla prima tecnica, usando i memory-mapped-file ogni operazione di posizionamento viene sostituita da una semplice assegnazione a un puntatore, e ogni operazione di lettura da file viene sostituita da una semplice copia da memoria a memoria. Anche supponendo che i dati siano già nella disk cache, entrambe le operazioni effettuate con i memory-mapped-files sono notevolmente più veloci delle operazioni effettuate sui file, in quanto queste ultime comportano altrettante chiamate di libreria, le quali a loro volta effettuano chiamate di sistema.

Rispetto alla tecnica di precaricare in memoria l'intero file, usando i memory-mapped-file si hanno i seguenti vantaggi:

- Usando le primitive di lettura di file, i dati vengono normalmente letti prima nella cache del disco e poi nella memoria del processo, mentre con i memory-mapped-file si accede direttamente al buffer caricato dal disco, risparmiando così sia un'operazione di copia che lo spazio di memoria per la cache del disco. Analoga situazione si ha per la scrittura su disco.
- Leggendo tutto il file, il programma si blocca per un tempo significativo per leggere il file, mentre usando un memory-mapped-file tale tempo viene distribuito nel corso dell'elaborazione, man mano che si accede alle varie parti del file.
- Se in alcune esecuzioni serve solo una piccola parte del file, il memory-mapped-file carica in memoria solo quelle parti.
- Se più processi devono caricare in memoria lo stesso file, lo spazio di memoria viene allocato per ogni processo, mentre usando i memory-mapped-file il sistema operativo tiene in memoria una sola copia dei dati, condivisa da tutti i processi.
- In condizioni di scarsità di memoria, il sistema operativo scrive nell'area di swap del disco anche la memoria del processo che non è stata modificata, mentre si limita a scartare le pagine non modificate del memory-mapped-file, senza scriverle sul disco.

Tuttavia, l'uso di memory mapped file non è appropriato in una porzione critica di un sistema real-time, in quanto l'accesso a tali dati ha una latenza fortemente variabile a seconda che il dato acceduto sia già stato caricato nella memoria di sistema o sia ancora solamente su disco.

A rigore, questa è una tecnica dipendente dalla piattaforma, in quanto la funzionalità dei memory-mapped-file non esiste in tutti i sistemi operativi. Tuttavia, dato che tale funzionalità esiste in tutti i principali sistemi operativi dotati di memoria virtuale, questa tecnica è di ampia applicabilità.

Ecco una classe che incapsula le primitive di accesso in sola lettura a un file tramite memory-mapped-file, seguita da un piccolo programma che dimostra l'uso di tale classe. È utilizzabile sia nei sistemi operativi di tipo Posix (come Unix, Linux, e Mac OS X), sia in ambiente Windows.

File "memory_file.hpp":

```
#ifndef MEMORY_FILE_HPP
#define MEMORY_FILE_HPP

/*
  Wrapper di memory-mapped file per sola lettura.
  Gestisce solo file che possono essere interamente caricati
  nello spazio di indirizzamento del processo.
  Il costruttore apre il file, il distruttore lo chiude.
  La funzione "data" rende un puntatore all'inizio del file,
  se il file e' stato aperto con successo, altrimenti rende 0.
  La funzione "length" rende la lunghezza in byte del file,
  se il file e' stato aperto con successo,
  altrimenti rende 0.
*/

class InputMemoryFile {
public:
    InputMemoryFile(const char *pathname);
    ~InputMemoryFile();
    const void* data() const { return data_; }
    unsigned long length() const { return length_; }
private:
    void* data_;
    unsigned long length_;
#if defined(__unix__)
    int file_handle_;
#elif defined(_WIN32)
    typedef void * HANDLE;
    HANDLE file_handle_;
    HANDLE file_mapping_handle_;
#else
    #error Solo i sistemi Posix o Windows possono usare i memory-mapped
files.
#endif
};
#endif
```

File "memory_file.cpp":

```
#include "memory_file.hpp"
#if defined(__unix__)
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#elif defined(_WIN32)
#include <windows.h>
#endif

InputMemoryFile::InputMemoryFile(const char *pathname):
    data_(0),
    length_(0),
#if defined(__unix__)
    file_handle_(-1)
{
    file_handle_ = open(pathname, O_RDONLY);
    if (file_handle_ == -1) return;
    struct stat sbuf;
    if (fstat(file_handle_, &sbuf) == -1) return;
```

```
    data_ = mmap(0, sbuf.st_size, PROT_READ, MAP_SHARED, file_handle_, 0);
    if (data_ == MAP_FAILED) data_ = 0;
    else length_ = sbuf.st_size;
#elif defined(_WIN32)
    file_handle_(INVALID_HANDLE_VALUE),
    file_mapping_handle_(INVALID_HANDLE_VALUE)
{
    file_handle_ = ::CreateFile(pathname, GENERIC_READ,
        FILE_SHARE_READ, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (file_handle_ == INVALID_HANDLE_VALUE) return;
    file_mapping_handle_ = ::CreateFileMapping(
        file_handle_, 0, PAGE_READONLY, 0, 0, 0);
    if (file_mapping_handle_ == INVALID_HANDLE_VALUE) return;
    data_ = ::MapViewOfFile(
        file_mapping_handle_, FILE_MAP_READ, 0, 0, 0);
    if (data_) length_ = ::GetFileSize(file_handle_, 0);
#endif
}

InputMemoryFile::~InputMemoryFile() {
#if defined(__unix__)
    munmap(data_, length_);
    close(file_handle_);
#elif defined(_WIN32)
    ::UnmapViewOfFile(data_);
    ::CloseHandle(file_mapping_handle_);
    ::CloseHandle(file_handle_);
#endif
}
```

File "memory_file_test.cpp":

```
include "memory_file.hpp"
#include <iostream>
#include <iterator>

int main() {
    // Scrive su console il contenuto del file sorgente.
    InputMemoryFile imf("memory_file_test.cpp");
    if (imf.data())
        copy((const char*)imf.data(),
            (const char*)imf.data() + imf.length(),
            std::ostream_iterator<char>(std::cout));
    else std::cerr << "Impossibile aprire il file";
}
```

4.2 Caching

Le tecniche di *caching* (chiamate anche tecniche di *memoizzazione*) si basano sul principio che se una funzione pura ossia una funzione *referenzialmente trasparente* (cioè una funzione matematica) deve essere calcolata più volte per lo stesso argomento, e se tale calcolo richiede parecchio tempo, si risparmia tempo salvando il risultato la prima volta che lo si calcola, e recuperando il valore salvato le volte successive.

4.2.1 La *look-up table*

Se devi calcolare spesso una funzione pura avente come dominio un piccolo intervallo di numeri interi, pre-calcola (in fase di sviluppo del software o in fase di inizializzazione dell'applicazione) tutti i valori della funzione per ogni valore del dominio, e ponili in un array statico, detto *lookup-table*. Quando devi calcolare il valore della funzione per un dato valore del dominio, preleva l'elemento corrispondente di tale array.

Per esempio, dovendo calcolare la radice quadrata di un numero intero compreso tra 0 e 9, conviene usare la seguente funzione:

```
double sqrt10(int i) {
    static double lookup_table[] = {
        0, 1, sqrt(2.), sqrt(3.), 2,
        sqrt(5.), sqrt(6.), sqrt(7.), sqrt(8.), 3,
    };
    assert(0 <= i && i < 10);
    return lookup_table[i];
}
```

L'accesso a un array è molto veloce, soprattutto se la cella a cui si accede si trova nella cache dei dati del processore. Pertanto, se la look-up table non è grande, quasi sicuramente il suo accesso è più veloce della funzione che si deve calcolare.

Se la *look-up table* risulta piuttosto grande, può non essere più conveniente, sia per l'impiego di memoria, sia per il tempo necessario a pre-calcolare tutti i valori, ma soprattutto per il fatto che non può essere contenuta nella cache dei dati del processore.

Se però la funzione da calcolare è molto lenta, è chiamata molte volte, e le si può dedicare molta memoria, può essere opportuno usare una *look-up table* grande decine o centinaia di KB. Tuttavia, raramente è opportuno superare un megabyte.

4.2.2 Caching a un posto

Se devi chiamare spesso una funzione pura con gli stessi argomenti, la prima volta che tale funzione viene chiamata salva gli argomenti e il risultato in variabili statiche; quando la funzione viene chiamata ancora, confronta i nuovi argomenti con quelli vecchi; se coincidono, rendi il risultato memorizzato, altrimenti calcola il risultato e memorizza i nuovi argomenti e il nuovo risultato.

Per esempio, invece della seguente funzione:

```
double f(double x, double y) {
    return sqrt(x * x + y * y);
}
```

puoi scrivere la seguente funzione:

```
double f(double x, double y) {
    static double prev_x = 0;
    static double prev_y = 0;
    static double result = 0;
    if (x == prev_x && y == prev_y) {
        return result;
    }
```

```
    prev_x = x;
    prev_y = y;
    result = sqrt(x * x + y * y);
    return result;
}
```

Nota che, per avere un vantaggio prestazionale, non è necessario che la funzione sia chiamata con gli stessi argomenti per tutta l'esecuzione del programma. È invece sufficiente che sia chiamata alcune volte con gli stessi argomenti, poi altre volte con altri argomenti, e così via. In tale caso, il calcolo viene effettuato solo quando gli argomenti cambiano.

Ovviamente, invece di migliorare le prestazioni, questa soluzione le peggiora nel caso in cui la funzione è chiamata con argomenti variabili quasi sempre, oppure se il confronto tra gli argomenti nuovi e quelli vecchi richiede più tempo del calcolo della funzione stessa.

Nota che a causa dell'uso di variabili statiche la routine non è più thread-safe, e non può essere ricorsiva. Se questa routine deve poter essere chiamata contemporaneamente da più thread, è necessario sostituire le variabili statiche con variabili distinte per ogni thread.

Nota anche che nell'esempio si assume che la funzione valga zero quando gli argomenti sono entrambi zero. In caso contrario, si dovrebbe adottare un'altra soluzione, come una delle seguenti:

- Inizializzare la variabile *result* al valore corrispondente ad argomenti tutti a valore zero.
- Inizializzare le variabili *prev_x* e *prev_y* a valori che non saranno mai passati come argomenti.
- Aggiungere un flag statico che indica se le variabili statiche hanno valori validi, e controllare tale flag ad ogni chiamata.

4.2.3 Caching a N posti

Se devi chiamare molte volte una funzione pura passando argomenti che nella maggior parte dei casi appartengono a un piccolo dominio, usa una *mappa* (detta anche *dizionario*) statica, inizialmente vuota. Quando la funzione viene chiamata, cerca l'argomento nella mappa. Se lo trovi, rendi il valore associato, altrimenti calcola il risultato e inserisci nella mappa la coppia argomento-risultato.

Ecco un esempio, in cui la mappa è implementata con un array, riferito alla stessa funzione della linea-guida "Caching a un posto" in questa sezione:

```
double f(double x, double y) {
    static const int n_buckets = 8; // Dovrebbe essere una potenza di 2
    static struct {
        double x; double y; double result;
    } cache[n_buckets];
    static int last_read_i = 0;
    static int last_written_i = 0;
    int i = last_read_i;
    do {
        if (cache[i].x == x && cache[i].y == y) {
            return cache[i].result;
        }
        i = (i + 1) % n_buckets;
    } while (i != last_read_i);
    last_read_i = last_written_i = (last_written_i + 1) % n_buckets;
    cache[last_written_i].x = x;
    cache[last_written_i].y = y;
```

```
    cache[last_written_i].result = sqrt(x * x + y * y);
    return cache[last_written_i].result;
}
```

Alcune funzioni, pur avendo un dominio teoricamente grande, sono chiamate frequentemente con pochi valori distinti.

Per esempio, un word processor può avere installato un grande numero di tipi di caratteri (detti anche *font*), ma in un tipico documento vengono usati solo pochissimi font. Una funzione che deve manipolare il font di ogni carattere del documento verrà chiamata tipicamente con pochissimi valori diversi.

In tali casi, invece di una cache a un solo posto, risulta preferibile una cache a più posti, come quella dell'esempio.

Anche per questo caso valgono le considerazioni riguardanti le variabili statiche, fatte nella linea-guida "Caching a un posto" in questa sezione.

Per cache piccole (come quella di soli 8 posti dell'esempio), l'algoritmo più efficiente è la scansione sequenziale su un array. Tuttavia, se si volesse fare una cache di dimensioni maggiori, un albero di ricerca o una hashtable sarebbero più efficienti.

Inoltre, nell'esempio la cache ha dimensione fissa, ma potrebbe essere opportuno avere invece una cache di dimensioni variabili.

Solitamente, l'ultimo elemento letto è il più probabile per la chiamata successiva. Quindi, come nell'esempio, può essere opportuno salvare la sua posizione e far partire la ricerca da tale posizione.

Se la cache non si espande indefinitamente, c'è il problema di quale elemento rimpiazzare. Ovviamente è meglio rimpiazzare l'elemento che è meno probabile che sia richiesto dalla chiamata successiva.

Nell'esempio si assume che tra gli elementi presenti nella cache, l'elemento inserito per primo sia il meno probabile. Pertanto, le scritture scorrono ciclicamente l'array.

Spesso un criterio migliore consiste nel sostituire l'elemento non letto da più tempo. Per attuare questo criterio, si deve usare un algoritmo un po' più complesso.

4.3 Ordinamento

4.3.1 Countingsort

Per ordinare un insieme di dati in base a una chiave intera avente un range limitato, usa l'algoritmo *counting sort*.

L'algoritmo *counting sort* ha complessità O(N+M), dove N è il numero di elementi da ordinare e M è il range delle chiavi di ordinamento, cioè la differenza tra la chiave massima e la chiave minima. Nel caso in cui si vogliano ordinare N elementi la cui chiave è un numero intero appartenente a un intervallo contenente un numero di valori non superiore al doppio di N (cioè quando vale `M <= 2 * N`), questo algoritmo può essere notevolmente più veloce di *quick sort*. In alcuni casi è più veloce anche per range più grandi.

Questo algoritmo può essere usato anche per un ordinamento parziale; per esempio, se la chiave è un intero compreso tra zero e un miliardo, si può ordinare solamente in base al byte più significativo

della chiave, così da ottenere un ordine tale per cui per ogni n vale la formula $a_n < a_{n+1} + 256 * 256 * 256$.

4.3.2 Partizionamento

Se devi solo dividere una sequenza in due gruppi in base a un criterio, usa un algoritmo di partizionamento, invece di uno di ordinamento.

In STL c'è l'algoritmo `std::partition`, che è più veloce dell'algoritmo `std::sort`, in quanto ha complessità O(N) invece di O(N log(N)).

4.3.3 Partizionamento e ordinamento stabili

Se devi partizionare oppure ordinare una sequenza per cui non è richiesto di mantenere l'ordine delle entità equivalenti, non usare un algoritmo stabile.

In STL c'è l'algoritmo di partizionamento `std::stable_partition`, che è leggermente più lento dell'algoritmo `std::partition`; e c'è l'algoritmo di ordinamento `std::stable_sort`, che è leggermente più lento dell'algoritmo `std::sort`.

4.3.4 Partizionamento d'ordine

Se devi solo individuare i primi N elementi di una sequenza, o l'N-esimo elemento di una sequenza, usa un algoritmo di partizionamento d'ordine, invece di uno di ordinamento.

In STL c'è l'algoritmo `std::nth_element`, che, pur essendo leggermente più lento dell'algoritmo `std::stable_partition`, è notevolmente più veloce dell'algoritmo `std::sort`, in quanto ha complessità O(N) invece di O(N log(N)).

4.3.5 Statistica d'ordine

Se devi solo ordinare i primi N elementi di una sequenza, usa un algoritmo di statistica d'ordine, invece di un algoritmo di ordinamento.

In STL ci sono gli algoritmi `std::partial_sort` e `std::partial_sort_copy`, che, pur essendo più lenti dell'algoritmo `std::nth_element`, sono tanto più veloci dell'algoritmo `std::sort` quanto più è breve la sequenza parziale da ordinare rispetto a quella totale.

4.4 Altre tecniche

4.4.1 Query con cursore

Invece di definire una funzione che restituisce al chiamante una collezione di dati (detta anche *snapshot*), definisci una funzione che restituisce un iteratore (detto anche *cursore* o *dynaset*), con il quale si possono generare ed eventualmente modificare i dati richiesti.

Questa tecnica è particolarmente utile per interrogazioni su database (dette anche *query*), ma è applicabile anche a strutture dati gestite internamente dall'applicazione.

Supponiamo di avere una collezione (o un insieme di collezioni) incapsulati in una classe. Tale classe espone uno o più metodi per estrarre (o filtrare) un sottoinsieme da tale collezione.

Un modo di ottenere ciò è costruire una nuova collezione, copiarci i dati estratti, e restituire tale collezione al chiamante. Nel mondo dei database, tale collezione si chiama *snapshot*. Questa tecnica è però inefficiente, perché ognuna delle suddette tre operazioni richiede molto tempo, e potrebbe richiede molta memoria. Inoltre, ha il difetto che, finché non sono stati estratti tutti i dati, non si può procedere a elaborare quelli già estratti.

Ecco una tecnica equivalente ma più efficiente.

La funzione di interrogazione rende un iteratore. Nel mondo dei database, tale iteratore si chiama *cursore* o *dynaset*. Il chiamante usa tale iteratore per estrarre, uno alla volta i dati filtrati, ed eventualmente per modificare tali dati.

Nota che questa soluzione non è del tutto equivalente, in quanto se durante l'uso dell'iteratore la collezione viene modificata da un'altra chiamata di funzione, eventualmente proveniente da un altro thread, può succedere che l'iteratore sia invalidato, o anche solo che l'insieme filtrato non corrisponda ai criteri impostati. Pertanto, questa tecnica è da usare solo se si ha la certezza che la collezione sottostante non viene modificata da nessuno, se non tramite l'iteratore stesso, durante tutta la vita dell'iteratore.

Questa tecnica è indipendente dal linguaggio di programmazione, in quanto il concetto di iteratore è un design pattern, implementabile in qualsiasi linguaggio. Per esempio in Python esistono i generatori, navigabili con appositi iteratori.

4.4.2 Ricerca binaria

Se devi fare molte ricerche in una collezione che viene variata raramente o mai, invece di usare un albero di ricerca o una hashtable, puoi migliorare la velocità se poni i dati in un array, ordini l'array, ed effettui le ricerche con il metodo dicotomico (noto anche come *ricerca binaria*).

La ricerca binaria ha complessità logaritmica, come gli alberi di ricerca, ma ha il pregio della compattezza e della località dei riferimenti propria degli array.

Se l'array viene modificato, questo algoritmo può essere ancora competitivo, purché le modifiche siano molte meno delle ricerche.

Se la modifica consiste in pochissimi inserimenti o modifiche o eliminazioni di elementi, conviene effettuare uno scorrimento dell'array ad ogni operazione. Se invece la modifica è più massiccia, conviene ricreare tutto l'array.

In C++, se la dimensione dell'array non è una costante, usa un `vector`.

4.4.3 Lista a collegamento singolo

Se per una lista non hai bisogno di iteratori bidirezionali, non devi inserire gli elementi solo alla fine né prima dell'elemento corrente, e non ti serve sapere quanti elementi ci sono nella lista, usa una lista a concatenamento singolo, invece di una lista a concatenamento doppio.

Tale contenitore, pur avendo molte limitazioni, occupa meno memoria ed è più veloce.

Infatti, tipicamente, l'intestazione di una lista a concatenamento doppio contiene un puntatore alla cima, un puntatore al fondo della lista, e il contatore del numero di elementi, mentre l'intestazione di

una lista a concatenamento singolo contiene solo un puntatore alla cima della lista. Inoltre, tipicamente, ogni nodo di una lista a concatenamento doppio contiene un puntatore all'elemento precedente e un puntatore all'elemento successivo, mentre ogni nodo di una lista a concatenamento singolo contiene solo un puntatore all'elemento successivo. Infine, ogni inserimento di un elemento in una lista a concatenamento doppio deve aggiornare quattro puntatori e incrementare un contatore, mentre ogni inserimento in una lista a concatenamento singolo deve solo aggiornare due puntatori.

Nella libreria standard del C++, il contenitore `std::list` è implementato da una lista a concatenamento doppio. Il contenitore `slist`, non standard ma disponibile in varie librerie, e il contenitore `forward_list`, che farà parte della libreria standard C++0x, sono implementati da una lista a concatenamento singolo.

Capitolo 5 – Ottimizzazione del codice C++

In questa sezione si suggeriscono dei trucchi, specifici del linguaggio C++, da adottare solamente nei colli di bottiglia, in quanto, pur rendendo il codice più veloce, ne rendono più complessa la stesura e lo rendono meno manutenibile.

Inoltre, tali linee-guida in alcuni casi potrebbero sortire l'effetto indesiderato di peggiorare le prestazioni invece che migliorarle, per cui bisognerebbe sempre misurarne l'effetto prima di rilasciarle.

Le tecniche di ottimizzazione sono raggruppate in base all'obiettivo che si propongono di raggiungere.

1. Allocazione e deallocazione
2. Supporto run-time
3. Numero di istruzioni
4. Costruzioni e distruzioni
5. Pipeline
6. Accesso alla memoria
7. Operazioni veloci

5.1 Allocazione e deallocazione

Per quanto sia efficiente l'allocatore, le operazioni di allocazione e deallocazione, richiedono parecchio tempo, e spesso l'allocatore non è molto efficiente.

In questa sezione si descrivono alcune tecniche per ridurre il numero complessivo di allocazioni di memoria, e quindi delle corrispondenti deallocazioni. Sono da adottare solamente nei colli di bottiglia, cioè dopo aver constatato che il grande numero di allocazioni ha un impatto significativo sulle prestazioni.

5.1.1 La funzione `alloca`

In funzioni non-ricorsive, per allocare spazio di dimensione variabile ma non grande, usa la funzione `alloca`.

È molto efficiente, in quanto alloca spazio sullo stack.

È una funzione non-standard, ma presente in molti compilatori per vari sistemi operativi.

Può essere usata anche per allocare un array di oggetti che hanno un costruttore, purché si chiami l'operatore `new` di piazzamento sullo spazio ottenuto, ma non dovrebbe essere usata per array di oggetti che hanno un distruttore o che, direttamente o indirettamente, contengono membri che hanno un distruttore, in quanto tali distruttori non verrebbero mai chiamati.

Tuttavia, è piuttosto pericolosa, in quanto, se chiamata troppe volte o con un valore troppo grande, esaurisce lo stack, e, se usata per oggetti aventi un distruttore, provoca resource leak. Pertanto si consiglia di usare con grande moderazione questa funzione.

5.1.2 Spostare le allocazioni e le deallocazioni

Sposta prima dei colli di bottiglia le allocazioni di memoria, e dopo i colli di bottiglia le corrispondenti deallocazioni.

La gestione di memoria dinamica di dimensione variabile è molto più lenta della gestione della memoria sullo stack.

Analoga ottimizzazione va fatta per le operazioni che provocano allocazioni indirettamente, come la copia di oggetti che, direttamente o indirettamente, possiedono memoria dinamica.

5.1.3 La funzione `reserve`

Prima di aggiungere elementi a un oggetto `vector` o `string`, chiama la sua funzione membro `reserve` con una dimensione sufficiente per la maggior parte dei casi.

Se si aggiungono ripetutamente elementi a oggetti `vector` o `string`, ogni tanto viene eseguita una costosa operazione di riallocazione del contenuto. Per evitare tali riallocazioni, basta allocare inizialmente lo spazio che probabilmente sarà sufficiente.

5.1.4 Mantenere la capacità dei `vector`

Per svuotare un oggetto `x` di tipo `vector<T>` senza deallocarne la memoria, usa l'istruzione `x.resize(0);`; per svuotarlo deallocandone la memoria, usa l'istruzione `vector<T>().swap(x);`.

Per svuotare un oggetto `vector`, esiste anche la funzione membro `clear()`; tuttavia lo standard C++ non specifica se tale istruzione conserva la capacità allocata del `vector` oppure no.

Se riempi e svuoti ripetutamente un oggetto `vector`, e quindi vuoi essere sicuro di evitare frequenti riallocazioni, esegui lo svuotamento chiamando la funzione `resize`, che, secondo lo standard, conserva sicuramente la capacità per il successivo riempimento.

Se invece hai finito di usare un oggetto `vector` di grandi dimensioni, e per un po' di tempo non lo userai più, oppure lo userai con un numero molto più piccolo di elementi, e quindi vuoi essere sicuro di liberare la memoria usata da tale collezione, chiama la funzione `swap` su un nuovo oggetto temporaneo `vector` vuoto.

5.1.5 Ridefinire la funzione `swap`

Per ogni classe concreta copiabile `T` che, direttamente o indirettamente, possiede della memoria dinamica, ridefinisci le appropriate funzioni `swap`.

In particolare, aggiungi alla classe una funzione membro `public` con la seguente firma:

```
void swap(T&) throw();
```

e aggiungi la seguente funzione non-membro nello stesso namespace che contiene la classe `T`:

```
void swap(T& lhs, T& rhs) { lhs.swap(rhs); }
```

e, se la classe non è un template di classe, aggiungi la seguente funzione non-membro nello stesso file che contiene la definizione della classe `T`:

```
namespace std { template<> swap(T& lhs, T& rhs) { lhs.swap(rhs); } }
```

Nella libreria standard la funzione `std::swap` viene richiamata frequentemente da molti algoritmi. Tale funzione ha una implementazione generica e ha implementazioni specializzate per vari tipi della libreria standard.

Se gli oggetti di una classe non-standard vengono usati in algoritmi della libreria standard, e non viene fornito un overload della funzione `swap`, viene usata l'implementazione generica.

L'implementazione generica di `swap` comporta la creazione e la distruzione di un oggetto temporaneo e l'esecuzione di due assegnamenti di oggetti. Tali operazioni richiedono molto tempo se applicate ad oggetti che possiedono della memoria dinamica, in quanto tale memoria viene riallocata per tre volte.

Il possesso di memoria dinamica citato nella linea-guida può essere anche solo indiretto. Per esempio, se una variabile membro è un oggetto string o vector, o è un oggetto che contiene un oggetto string o vector, la memoria posseduta da tali oggetti viene riallocata ogni volta che si copia l'oggetto che li contiene. Quindi anche in tali casi si devono ridefinire le funzioni `swap`.

Se l'oggetto non possiede memoria dinamica, la copia dell'oggetto è molto più veloce, e comunque non sensibilmente più lenta che usando altre tecniche, e quindi non si devono ridefinire funzioni `swap`.

Se la classe non è copiabile o è astratta, la funzione `swap` non dovrebbe mai essere chiamata sugli oggetti di tale tipo, e quindi anche in questo caso non si devono ridefinire funzioni `swap`.

Per velocizzare la funzione `swap`, la si deve specializzare per la propria classe. Ci sono due modi possibili per farlo: nel namespace della stessa classe (che può essere quello globale) come overload, oppure nel namespace `std` come specializzazione del template standard. È meglio definirla in entrambi i modi, in quanto, in primo luogo, se si tratta di un template di classe solo il primo modo è possibile, e poi alcuni compilatori non accettano o segnalano un avvertimento se è definita solo nel primo modo.

L'implementazione di tali funzioni devono accedere a tutti i membri dell'oggetto, e quindi hanno bisogno di richiamare una funzione membro, che per convenzione si chiama ancora `swap`, che effettua il lavoro.

Tale lavoro deve consistere nello scambiare tutti i membri non-static dei due oggetti, tipicamente chiamando la funzione `swap` su di essi, senza qualificarne il namespace.

Per consentire di trovare la funzione `std::swap`, la funzione membro deve iniziare con la seguente istruzione:

```
using std::swap;
```

5.2 Supporto run-time

Le routine del supporto run-time del linguaggio C++ hanno ovviamente un costo significativo, in quanto altrimenti tali funzionalità sarebbero state espanse *inline*.

In questa sezione si presentano tecniche per evitare costrutti che comportano la chiamata implicita di costose routine del supporto run-time.

5.2.1 L'operatore `typeid`

Invece di usare l'operatore `typeid`, usa una funzione `virtual`.

Tale operatore può richiedere un tempo superiore a quello richiesto da una semplice chiamata a funzione.

5.2.2 L'operatore `dynamic_cast`

Invece dell'operatore `dynamic_cast`, usa l'operatore `typeid`, o, ancora meglio, una funzione `virtual`.

Tale operatore può richiedere un tempo notevolmente superiore a quello richiesto da una semplice chiamata a funzione, e maggiore perfino di quello richiesto dall'operatore `typeid`.

5.2.3 La specifica di eccezioni vuota

Usa la specifica di eccezioni vuota (cioè aggiungi `throw()` dopo la dichiarazione) alle funzioni di cui sei certo che non solleveranno eccezioni.

Alcuni compilatori utilizzano tale informazione per ottimizzare la contabilità necessaria per gestire le eventuali eccezioni.

5.2.4 Il costrutto `try/catch`

Sposta prima dei colli di bottiglia le istruzioni `try`, e dopo i colli di bottiglia le corrispondenti istruzioni `catch`.

In altre parole, estrai dai colli di bottiglia i blocchi `try/catch`.

L'esecuzione di un blocco `try/catch` talvolta ha costo zero, ma altre volte comporta un rallentamento. Evita di eseguire tale blocco all'interno dei colli di bottiglia.

5.2.5 Operazioni a virgola mobile o intere

Se il processore target non contiene un'unità a virgola mobile, sostituisci le funzioni, costanti e variabili a virgola mobile con le corrispondenti funzioni, costanti e variabili intere; mentre se contiene un'unità a virgola mobile a precisione solamente singola, sostituisci funzioni, costanti e variabili di tipo `double` con le corrispondenti di tipo `float`.

Gli odierni processori per sistemi desktop e server contengono hardware dedicato all'aritmetica a virgola mobile, sia a precisione singola che doppia, per cui tali operazioni, sono pressoché altrettanto veloci quanto quelle su numeri interi.

Alcuni processori dedicati per sistemi embedded, invece, non contengono hardware dedicato all'aritmetica a virgola mobile, o contengono hardware in grado di gestire solo la precisione singola. Pertanto, in tali sistemi, le operazioni non disponibili in hardware vengono solamente emulate con

lentissime funzioni di libreria. In tal caso, risulta molto più veloce utilizzare l'aritmetica intera, oppure, se disponibile in hardware, l'aritmetica a precisione singola.

Per gestire numeri decimali usando l'aritmetica intera si devono usare gli interi intendendoli moltiplicati per un fattore di scala. A tale scopo, ogni numero viene moltiplicato per tale fattore in fase di input e viene diviso per lo stesso fattore in fase di output, o viceversa.

5.2.6 Conversione da numero a stringa

Usa funzioni ottimizzate per convertire numeri in stringa.

Le funzioni standard di conversione da numero intero a stringa e da numero a virgola mobile a stringa sono poco efficienti. Per svolgere velocemente tali operazioni, usa funzioni ottimizzate non-standard, eventualmente scritte da te.

5.2.7 Uso delle funzioni di `cstdio`

Per eseguire operazioni di input/output, invece di usare gli stream del C++, usa le vecchie funzioni del C, dichiarate nel file di intestazione `cstdio`.

Le primitive di I/O del C++ sono state progettate principalmente per effettuare il controllo statico dei tipi (o *type safety*) e per consentire la personalizzazione per le proprie classi invece che per le prestazioni, e molte loro implementazioni risultano essere piuttosto inefficienti. In particolare, le funzioni di I/O del linguaggio C `fread` e `fwrite` sono più efficienti delle funzioni membro della classe `fstream` `read` e `write`.

5.3 Numero di istruzioni

Anche i costrutti che generano del codice espanso *inline* possono avere un costo significativo, in quanto tali istruzioni devono pur essere eseguite.

In questa sezione si descrivono le tecniche per ridurre il numero complessivo di istruzioni che il processore dovrà eseguire per compiere una data operazione.

5.3.1 Ordine dei casi in istruzioni `switch`

Nelle istruzioni `switch`, poni i casi in ordine di probabilità decrescente.

Nella linea-guida "Ordine dei casi dell'istruzione `switch`" del capitolo 3.1, si consigliava già di porre prima di casi più tipici, cioè quelli che si presume siano più probabili. Come ulteriore ottimizzazione, si potrebbe contare, in esecuzioni tipiche, il numero di volte in cui viene eseguito ognuno dei singoli casi, e porre i casi in ordine da quello eseguito più volte a quello eseguito meno volte.

5.3.2 Parametri interi di template

Se un certo valore intero è una costante nel codice applicativo, ma è una variabile nel codice di libreria, rendilo un parametro di template.

Supponi che stai scrivendo la seguente funzione di libreria, in cui sia `x` che `y` non hanno un valore definito in fase di sviluppo della libreria:

```
int f1(int x, int y) { return x * y; }
```

Tale funzione può essere chiamata dal seguente codice applicativo, nel quale `x` non ha un valore costante, ma `y` è la costante 4:

```
int a = f1(b, 4);
```

Se, mentre scrivi la libreria, sai che il chiamante ti passerà sicuramente una costante intera come argomento `y`, puoi trasformare la tua funzione nel seguente template di funzione:

```
template <int Y> int f2(int x) { return x * Y; }
```

Tale funzione può essere chiamata dal seguente codice applicativo:

```
int a = f2<4>(b);
```

Tale chiamata istanzia automaticamente la seguente funzione:

```
int f2(int x) { return x * 4; }
```

Quest'ultima funzione è più veloce della precedente funzione `f1`, per i seguenti motivi:

- Viene passato un solo parametro alla funzione (`x`) invece di due (`x` e `y`).
- La moltiplicazione per una costante intera (4) è più veloce della moltiplicazione per una variabile intera (`y`).
- Dato che il valore costante (4) è una potenza di due, il compilatore, invece di eseguire una moltiplicazione intera, esegue uno scorrimento di bit.

In generale, i parametri di template interi sono delle costanti per chi istanzia il template e quindi per il compilatore, e le costanti sono gestite in modo più efficiente delle variabili. Inoltre, alcune operazioni su costanti vengono pre-calcolate in fase di compilazione.

Se invece di avere una funzione avevi già un template di funzione, basta aggiungere un ulteriore parametro a tale template.

5.3.3 Il *Curiously Recurring Template Pattern*

Se devi scrivere una classe base astratta di libreria tale che in ogni algoritmo nel codice applicativo si userà una sola classe derivata da tale classe base, usa il *Curiously Recurring Template Pattern*.

Supponi che stai scrivendo la seguente classe base di libreria:

```
class Base {
public:
    void g() { f(); }
private:
    virtual void f() = 0;
};
```

In questa classe, la funzione `g` esegue un algoritmo che chiama la funzione `f` come operazione astratta per l'algoritmo. Nella terminologia dei *design pattern*, `g` è un template method, ossia un algoritmo astratto con uno o più punti di personalizzazione. Lo scopo di questa classe è consentire di scrivere il seguente codice applicativo:

```
class Derivata1: public Base {
private:
    virtual void f() { ... }
};
...
Base* p1 = new Derivata1;
p1->g();
```

In tal caso, è possibile convertire il precedente codice di libreria nel seguente:

```
template <class Derivata> class Base {
public:
    void g() { f(); }
private:
    void f() { static_cast<Derivata*>(this)->f(); }
};
```

Il codice applicativo, di conseguenza, diventerà il seguente:

```
class Derivata1: public Base<Derivata1> {
public:
    void f() { ... }
};
...
Derivata1* p1 = new Derivata1;
p1->g();
```

In tal modo, si ha binding statico alla funzione membro `Derivata1::f` della chiamata a `f` fatta all'interno della funzione membro `Base<Derivata1>::g`, cioè la chiamata a tale funzione non è più di tipo `virtual`, e può essere espansa *inline*.

Tuttavia, supponiamo che si volesse aggiungere la seguente definizione:

```
class Derivata2: public Base<Derivata2> {
public:
    void f() { ... }
};
```

Con questa soluzione non sarebbe più possibile definire un puntatore o riferimento a una classe base comune sia a `Derivata1` che a `Derivata2`, in quanto tali classi risultano due tipi senza alcuna relazione; di conseguenza, questa soluzione non è applicabile se si vuole permettere al codice applicativo di definire un contenitore di oggetti arbitrari derivati dalla classe Base.

Altre limitazioni sono le seguenti:

- `Base` è necessariamente un tipo astratto;
- un oggetto di tipo Derivata1 non può essere convertito in un oggetto di tipo `Derivata2` o viceversa;
- per ogni derivazione di `Base`, tutto il codice macchina di `Base` viene duplicato.

5.3.4 Il pattern *Strategy*

Se un oggetto che implementa il *design pattern Strategy* (noto anche come design pattern *Policy*) è una costante in ogni algoritmo nel codice applicativo, elimina tale oggetto, rendi `static` tutti i suoi membri, e aggiungi tale classe come parametro di template.

Supponi che stai scrivendo il seguente codice di libreria, che implementa il design pattern *Strategy*:

```
class C;
class Strategy {
public:
    virtual bool is_valid(const C&) const = 0;
    virtual void run(C&) const = 0;
};

class C {
public:
    void set_strategy(const Strategy& s): s_(s) { }
    void f() { if (s.is_valid(*this)) s.run(*this); }
private:
    Strategy s_;
};
```

Questo codice di libreria ha lo scopo di consentire il seguente codice applicativo:

```
class MyStrategy: public Strategy {
public:
    virtual bool is_valid(const C& c) const { ... }
    virtual void run(C& c) const { ... }
};
...
MyStrategy s; // Oggetto che rappresenta la mia strategia.
C c; // Oggetto contenente un algoritmo con strategia personalizzabile.
c.set_strategy(s); // Assegnazione della strategia personalizzata.
c.f(); // Esecuzione dell'algoritmo con la strategia assegnata.
```

In tal caso, è possibile convertire il precedente codice di libreria nel seguente:

```
template <class Strategy>
class C {
public:
    void f() {
        if (Strategy::is_valid(*this)) Strategy::run(*this);
    }
};
```

Il codice applicativo, di conseguenza, diventerà il seguente:

```
class MyStrategy {
public:
    static bool is_valid(const C<MyStrategy>& c) { ... }
    static void run(C<MyStrategy>& c) { ... }
};
...

C<MyStrategy> c; // Oggetto con strategia assegnata staticamente.
c.f(); // Esecuzione con strategia assegnata staticamente.
```

In tal modo, si evita l'oggetto-strategia, e si ha il binding statico delle funzioni membro `MyStrategy::is_valid` e `MyStrategy::run`, cioè si evitano chiamate a funzioni `virtual`.

Tuttavia, tale soluzione non consente di decidere la strategia in fase di esecuzione, e tanto meno di cambiarla durante la vita dell'oggetto. Inoltre, il codice dell'algoritmo astratto viene duplicato ogni volta che viene personalizzato.

5.3.5 Operatori bit-a-bit

Dovendo eseguire operazioni booleane su un insieme di singoli bit, affianca tali bit in un oggetto di tipo `unsigned int`, e usa gli operatori bit-a-bit su tale oggetto.

Gli operatori bit-a-bit (`&`, `|`, `^`, <<, e >>) sono tradotti in singole istruzioni veloci, e operano su tutti i bit di un registro in una sola istruzione.

5.4 Costruzioni e distruzioni

Spesso capita che per elaborare un'espressione venga creato un oggetto temporaneo, che viene distrutto alla fine della stessa espressione in cui viene creato. Se tale oggetto è di un tipo fondamentale, il compilatore quasi sempre riesce a evitarne la creazione, e comunque la creazione e la distruzione di un oggetto di un tipo fondamentale sono abbastanza veloci. Invece, se l'oggetto è invece di un tipo composito, la sua creazione e la sua distruzione hanno un costo illimitato, in quanto comportano la chiamata di un costruttore e di un distruttore, che possono avere qualunque durata.
In questa sezione si descrivono alcune tecniche per evitare che siano creati oggetti temporanei di tipo composito, e quindi che siano chiamati i relativi costruttori e distruttori.

5.4.1 Valore di ritorno di funzioni

Per le funzioni che non siano espanse *inline*, cerca di dichiarare un tipo di ritorno tale che la copia di oggetti di tale tipo non sposta più di 8 byte. Se non fosse fattibile, almeno costruisci l'oggetto da ritornare nelle stesse istruzioni `return`.

Nella compilazione di una funzione non espansa *inline*, il compilatore non può sapere se il valore di ritorno verrà usato, e quindi lo deve comunque generare. Generare un oggetto la cui copia non sposta più di 8 byte costa poco o niente, ma generare oggetti più complessi richiede tempo. Se l'oggetto temporaneo possiede delle risorse, il tempo richiesto è enormemente maggiore, ma anche senza allocazioni, il tempo richiesto cresce al crescere del numero delle word che vengono copiate quando si copia un oggetto di tale tipo.

Comunque, se si costruisce l'oggetto da ritornare nelle stesse istruzioni `return`, senza quindi assegnare tale valore a una variabile, si sfrutta l'ottimizzazione garantita dallo standard detta *Return Value Optimization*, che previene la creazione di oggetti temporanei.

Alcuni compilatori riescono a evitare la creazione di oggetti temporanei, anche se questi sono legati a variabili locali (con la cosiddetta *Named Return Value Optimization*), ma in generale questo non è garantito e ha comunque alcune limitazioni.

Per verificare se viene attuata una di tali ottimizzazioni, incrementa un contatore statico nei costruttori, nel distruttore, e nell'operatore di assegnamento della classe dell'oggetto ritornato. Nel caso non risultassero applicate ottimizzazioni, ricorri a una delle seguenti tecniche alternative:

- Rendi la funzione `void`, e aggiungile un argomento passato per riferimento, che funge da valore di ritorno.
- Trasforma la funzione in un costruttore del tipo ritornato, che riceve gli stessi parametri della funzione.
- Fai in modo che la funzione restituisca un oggetto di un tipo ausiliario che ruba le risorse e le cede all'oggetto destinazione, senza copiarle.
- Usa un *expression template*, che è una tecnica avanzata, facente parte del paradigma di programmazione detto *Template metaprogramming*.
- Se usi lo standard C++0x, usa un *rvalue reference*.

5.4.2 Spostamento di variabili all'esterno di cicli

Se una variabile è dichiarata all'interno di un ciclo, e l'assegnamento ad essa costa di meno di una costruzione più una distruzione, sposta tale dichiarazione prima del ciclo.

Se la variabile è dichiarata all'interno del ciclo, l'oggetto associato ad essa viene costruito e distrutto a ogni iterazione, mentre se è esterna al ciclo, tale oggetto viene costruito e distrutto una volta sola, ma presumibilmente viene assegnato una volta in più nel corpo del ciclo.

Tuttavia, in molti casi un assegnamento costa esattamente quanto una coppia costruzione+distruzione, per cui in tali casi non ci sono vantaggi a spostare la dichiarazione all'esterno e aggiungere un assegnamento all'interno.

5.4.3 Operatore di assegnamento

In un overload dell'operatore di assegnamento (operator=), se sei sicuro che non solleverà eccezioni, copia ogni variabile membro, invece di usare l'idioma *copy&swap*.

La tecnica più efficiente per copiare un oggetto è imitare una corretta lista di inizializzazione di un costruttore di copia, cioè, prima, si chiama l'analoga funzione membro delle classi base, e poi si copiano tutte le variabili membro, in ordine di dichiarazione.

Purtroppo, tale tecnica non è *exception-safe*, cioè se durante questa operazione viene sollevata un'eccezione, i distruttori di alcuni sotto-oggetti già costruiti potrebbero non venire mai chiamati. Pertanto, se c'è la possibilità che durante la copia venga sollevata un'eccezione, si deve usare una tecnica *exception-safe*, che tuttavia non avrà prestazioni ottimali.

La tecnica di assegnamento *exception-safe* più elegante è l'idioma *copy&swap*. Viene mostrata dal seguente codice, nel quale `C` rappresenta il nome della classe, e `swap` una funzione membro che dovrà essere definita:

```
C& C::operator=(C new_value) {
    swap(new_value);
    return *this;
}
```

5.4.4 Overload per evitare conversioni

Per evitare costose conversioni di tipo, definisci delle funzioni in overload per i tipi di argomento più comune.

Supponiamo di aver scritto la seguente funzione:

```
int f(const std::string& s) { return s[0]; }
```

il cui scopo è consentire di scrivere il segueente codice:

```
std::string s("abc");
int n = f(s);
```

Tale funzione può però essere usata anche dal seguente codice:

```
int n = f(string("abc"));
```

E, grazie alla conversione implicita da `char*` a `std::string`, può essere usata anche dal seguente codice:

```
int n = f("abc");
```

Entrambe le due ultime chiamate alla funzione `f` sono inefficienti, perché creano un oggetto temporaneo `std::string` non vuoto.

Per mantenere l'efficienza della prima chiamata dell'esempio, si dovrebbe definire anche la seguente funzione in overload:

```
int f(const char* s) { return s[0]; }
```

In generale, se una funzione è chiamata passandole un argomento di un tipo non consentito ma che può venire implicitamente convertito a un tipo consentito, viene creato un oggetto temporaneo del tipo consentito.

Per evitare tale oggetto temporaneo, si deve definire una funzione in overload rispetto alla funzione originale, che prenda un argomento del tipo dell'effettivo oggetto passato, evitando così la necessità di una conversione.

5.5 Pipeline

Le istruzioni in linguaggio macchina di salto condizionato (chiamate anche *branch*), possono essere generate dalla compilazione di vari costrutti C++, tra cui le istruzioni `if-else`, `for`, `while`, `do-while`, `switch-case`, e dagli operatori booleani e dall'operatore di espressione condizionale (`?:`).
I moderni processori elaborano efficientemente i salti condizionati solo se riescono a prevederli. In caso di errore di previsione, il lavoro che hanno già incominciato a fare, caricando nella pipeline le istruzioni successive, risulta inutile e devono ripartire dall'istruzione di destinazione del salto.
La predizione del salto si basa sulle iterazioni precedenti. Se queste sono regolari, il salto viene predetto correttamente.

I casi migliori sono quelli in cui un'istruzione di salto ha sempre lo stesso effetto; in tali casi, la predizione è quasi sempre corretta. Il caso peggiore è quello in cui l'istruzione di salto ha un esito casuale, con una probabilità del 50% di eseguire il salto; in tale caso, la predizione è corretta mediamente solo la metà delle volte, ma non è impossibile che sia sempre sbagliata.

Nei colli di bottiglia, si dovrebbero evitare le istruzioni di salto difficilmente prevedibili. Se un salto è predetto molto male, anche sostituendolo con una serie di istruzioni piuttosto lenta si può ottenere un incremento di velocità.

In questa sezione, vengono presentate alcune tecniche per sostituire le istruzioni di salto con istruzioni equivalenti.

5.5.1 Verifica di intervallo

Se devi verificare se un numero intero `i` è compreso tra i due numeri interi `min_i` e `max_i`, estremi inclusi, e sei sicuro che `min_i <= max_i`, usa la seguente espressione:

```
unsigned(i - min_i) <= unsigned(max_i - min_i)
```

Nelle condizioni date, la suddetta formula è equivalente alla seguente formula, più intuitiva:

```
min_i <= i && i <= max_i
```

La prima formula esegue due differenze e un confronto, mentre la seconda esegue due confronti e nessuna differenza. Per i processori con pipeline, i confronti sono più lenti delle differenze, perché comportano dei salti condizionati.

Inoltre, se `min_i` è un'espressione costante di valore zero, le due differenze non sono più necessarie.

In particolare, per verificare se un numero `i` è valido come indice per accedere a un array di `size` elementi, la formula si riduce alla seguente:

```
unsigned(i) < unsigned(size)
```

Ovviamente, se le espressioni utilizzate sono già di un tipo `unsigned`, le conversioni non sono necessarie.

5.5.2 La *look-up table* binaria.

Invece di un'espressione condizionale con entrambi i casi costanti, usa una *look-up table* a due valori.

Ossia, se hai del codice come il seguente, in cui `c` e `d` rappresentano espressioni costanti e `b` rappresenta un'espressione booleana:

```
a = b ? c : d;
```

che è equivalente al seguente codice:

```
if (b) a = c;
else a = d;
```

prova a sostituirla con il seguente codice, equivalente ma forse più veloce:

```
static const tipo lookup_table[] = { d, c };
```

```
a = lookup_table[b];
```

L'espressione condizionale viene compilata in un salto condizionato. Se tale salto non è predetto bene, costa di più della lookup-table.

Ovviamente questa tecnica può essere estesa a cascate di espressioni condizionali. Per esempio, invece del seguente codice:

```
a = b1 ? c : b2 ? d : b3 ? e : f;
```

che è equivalente al seguente codice:

```
if (b1) a = c;
else if (b2) a = d;
else if (b3) a = e;
else a = f;
```

puoi provare a vedere se il seguente codice equivalente è più veloce:

```
static const tipo lookup_table[] = { f, e, d, d, c, c, c, c };
a = lookup_table[b1 * 4 + b2 * 2 + b3];
```

5.5.3 Anticipazione del calcolo degli indirizzi

Cerca di calcolare il valore di un puntatore o iteratore un po' prima di quando devi accedere all'oggetto referenziato.

Per esempio, in un ciclo, la seguente istruzione:

```
a = *++p;
```

può essere un po' meno efficiente della seguente:

```
a = *p++;
```

Nel primo caso il valore del puntatore o iteratore è calcolato appena prima di accedere all'oggetto referenziato, mentre nel secondo caso è calcolato nell'iterazione precedente. In un processore con pipeline, nel secondo caso, l'incremento dell'iteratore o puntatore può essere eseguito contemporaneamente all'accesso all'oggetto referenziato, mentre nel primo caso le due operazioni devono essere serializzate.

5.6 Accesso alla memoria

L'accesso alla memoria principale da parte del processore fa implicitamente uso sia delle varie cache del processore, che del meccanismo di swapping su disco del gestore della memoria virtuale del sistema operativo.

Sia le cache del processore che il gestore della memoria virtuale elaborano i dati a blocchi, per cui il software è più veloce se pochi blocchi di memoria contengono il codice e i dati usati da un solo

comando. Il principio che i dati e il codice elaborati da un comando debbano stare vicini in memoria è detto *località dei riferimenti*.

Questo principio diventa ancora più importante per le prestazioni di applicazioni multi-threaded su sistemi multi-core, dato che se più thread in esecuzione in core distinti accedono allo stesso blocco di cache, la contesa provoca un degrado delle prestazioni.

In questa sezione vengono proposte delle tecniche che ottimizzano l'uso delle cache del processore e della memoria virtuale, tramite un aumento della località dei riferimenti del codice e dei dati.

5.6.1 Avvicinare il codice

Poni vicine nella stessa unità di compilazione tutte le definizioni di funzioni appartenenti allo stesso collo di bottiglia.

In tal modo, il codice macchina generato compilando tali funzioni avrà indirizzi vicini, e quindi maggiore località dei riferimenti del codice.

Un'altra conseguenza positiva è che i dati statici locali dichiarati e usati da tali funzioni avranno indirizzi vicini, e quindi maggiore località dei riferimenti dei dati.

5.6.2 Le `union`

In array o collezioni medi o grandi, usa le `union`.

Le `union` permettono di risparmiare memoria in strutture di tipi variabili, e quindi di renderle più compatte.

Però non usarle in oggetti piccoli o piccolissimi, in quanto non si hanno vantaggi significativi per il risparmio di memoria, e con alcuni compilatori le variabili poste nelle `union` non vengono tenute nei registri del processore.

5.6.3 I *bit-field*

Se un oggetto medio o grande contiene più numeri interi con un range limitato, trasformali in *bit-field*.

I *bit-field* riducono le dimensioni dell'oggetto,

Per esempio, invece della seguente struttura:

```
struct {
    bool b;
    unsigned short ui1, ui2, ui3; // range: [0, 1000]
};
```

che occupa 8 byte, puoi definire la seguente struttura:

```
struct {
    unsigned b: 1;
    unsigned ui1: 10, ui2: 10, ui3: 10; // range: [0, 1000]
};
```

che occupa solamente (1 + 10 + 10 + 10 = 31 bit, 31 <= 32) 4 byte.

Per fare un altro esempio, invece del seguente array:

```
unsigned char a[5]; // range: [-20, +20]
```

che occupa 5 byte, puoi definire la seguente struttura:

```
struct {
    signed a1: 6, a2: 6, a3: 6, a4: 6, a5: 6; // range: [-20, +20]
};
```

che occupa solamente (6 + 6 + 6 + 6 + 6 = 30 bits, 30 <= 32) 4 bytes.

Tuttavia, c'è una penalità prestazionale nell'impaccare e disimpaccare i campi. Inoltre, nell'ultimo esempio, i campi non sono più accessibile tramite un indice.

5.6.4 Codice di template non dipendente dai parametri

Se in un template di classe una funzione membro non banale non dipende da nessun parametro del template, definisci una funzione non-membro avente lo stesso corpo, e sostituisci il corpo della funzione originale con una chiamata alla nuova funzione.

Supponiamo di aver scritto il seguente codice:

```
template <typename T>
class C {
public:
    C(): x_(0) { }
    int f(int i) { body(); return i; }
private:
    T x_;
};
```

Può convenire sostituire tale codice con il seguente:

```
template <typename T>
class C {
public:
    C(): x_(0) { }
    void f(int i) { return f_(i); }
private:
    T x_;
};

void f_(int i) { body(); return i; }
```

Ad ogni istanziazione di un template di classe che fa uso di una funzione di quel template di classe, tutto il codice di quella funzione viene istanziato. Se una funzione di quel template di classe non dipende dai parametri del template, ad ogni istanziazione di tale funzione il suo codice macchina verrà duplicato. Tale replicazione di codice ingrandisce inutilmente il programma.

In un template di classe o in un template di funzione, una grossa funzione potrebbe avere una grande porzione che non dipende da nessun parametro di template. In tal caso, in primo luogo scorpora tale porzione di codice come una funzione distinta, e poi applica questa linea-guida.

5.7 Operazioni veloci

Alcune operazioni elementari, per quanto concettualmente altrettanto semplici di altre, sono molto più veloci per il processore. Un abile programmatore sa scegliere le istruzioni più veloci per eseguire un dato compito.

Tuttavia, ogni buon compilatore ottimizzante è già in grado di scegliere le istruzioni più veloci per il processore target, per cui alcune tecniche sono inutili su alcuni compilatori.

Inoltre, alcune tecniche possono persino peggiorare le prestazioni su alcuni processori.

In questa sezione vengono presentate alcune tecniche che possono offrire vantaggi prestazionali su alcune combinazioni di compilatore/processore.

5.7.1 Ordinamento dei campi di strutture

Disponi le variabili membro di classi e strutture in modo che le variabili più usate siano nei primi 128 byte, e poi in ordine dall'oggetto più lungo a quello più corto.

Se nella seguente struttura il membro `msg` viene usato solamente per messaggi d'errore, mentre gli altri membri sono usati per effettuare calcoli:

```
struct {
    char c[400];
    double d;
    int i;
};
```

si possono velocizzare i calcoli trasformando tale struttura nella seguente:

```
struct {
    double d;
    int i;
    char c[400];
};
```

Su alcuni processori, l'indirizzamento di un membro è più efficiente se la sua distanza dall'inizio della struttura non supera i 128 byte.

Nel primo esempio, per indirizzare i campi `d` e `i` usando un puntatore all'inizio della struttura, si è costretti a usare un offset di almeno 400 byte.

Invece, nel secondo esempio, contenente gli stessi campi in un altro ordine, gli offset per indirizzare i campi `d` e `i` sono di pochi byte, e ciò permette l'uso di istruzioni più compatte.

Adesso, supponiamo di aver scritto la seguente struttura:

```
struct {
    bool b;
    double d;
    short s;
    int i;
};
```

Tale struttura, a causa dei requisiti di allineamento dei campi, tipicamente occupa 1 (bool) + 7 (padding) + 8 (double) + 2 (short) + 2 (padding) + 4 (int) = 24 byte.

La seguente struttura è ottenuta dalla precedente ordinando i campi dal più lungo al più corto:

```
struct {
    double d;
    int i;
    short s;
    bool b;
};
```

Tale struttura tipicamente occupa 8 (double) + 4 (int) + 2 (short) + 1 (bool) + 1 (padding) = 16 byte. L'ordinamento ha minimizza gli spazi per l'allineamento (padding), e così genera una struttura più compatta.

5.7.2 Conversione da numero a virgola mobile a numero intero

Sfrutta routine non-standard per arrotondare a interi i numeri in virgola mobile.

Il linguaggio C++ non fornisce una primitiva per arrotondare numeri a virgola mobile. La tecnica più semplice per convertire un numero a virgola mobile `x` all'intero più vicino `n`, è la seguente istruzione:

```
n = int(floor(x + 0.5f));
```

Usando tale tecnica, se `x` è esattamente equidistante tra due interi, `n` sarà l'intero superiore (per esempio, 0.5 genera 1, 1.5 genera 2, -0.5 genera 0, e -1.5 genera -1).

Purtroppo, su alcuni processori (in particolare quelli della famiglia Pentium), tale espressione viene compilata in un codice macchina molto lento. Ma alcuni processori hanno istruzioni specifiche per arrotondare i numeri.

In particolare, la famiglia Pentium ha l'istruzione macchina `fistp`, che, usata nel seguente codice, risulta molto più veloce, sebbene non esattamente equivalente:

```
#if defined(__unix__) || defined(__GNUC__)
    // Per a Linux 32-bit, con sintassi Gnu/AT&T
    __asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );
#else
    // Per Windows a 32-bit, con sintassi Intel/MASM
    __asm fld qword ptr x;
    __asm fistp dword ptr n;
#endif
```

Il codice precedente arrotonda `x` all'intero più vicino, ma se `x` è esattamente equidistante tra due interi, `n` sarà l'intero pari più vicino (per esempio, 0.5 genera 0, 1.5 genera 2, -0.5 genera 0, e -1.5 genera -2).

Se questo risultato è tollerabile o addirittura desiderato, e ti è consentito usare il linguaggio assembly, allora questo codice è consigliabile. Ovviamente, non è portabile ad altre famiglie di processori.

5.7.3 Manipolazione dei bit di numeri interi

Manipola i bit dei numeri interi sfruttando la conoscenza del formato di rappresentazione.

Una raccolta di trucchi di questo tipo si trova qui. Alcuni di questi trucchi sono in realtà già utilizzati da alcuni compilatori, altri servono per risolvere problemi rari, altri sono utili solo su alcune piattaforme.

5.7.4 Manipolazione dei bit di numeri a virgola mobile

Manipola i bit dei numeri a virgola mobile, dopo averli reinterpretati come numeri interi, sfruttando la conoscenza del formato di rappresentazione.

Per le operazioni più comuni, i compilatori generano già codice ottimizzato, ma alcune operazioni meno comuni possono diventare leggermente più veloci se i bit sono manipolati usando operatori interi bit-a-bit.

Una di tali operazioni è la moltiplicazione o la divisione per una potenza di due. Per eseguire tali operazioni, basta aggiungere l'esponente di tale potenza all'esponente del numero a virgola mobile.

Per esempio, data una variabile `f` del tipo `float` conforme al formato IEEE 754, e data un'espressione intera positiva `n`, invece della seguente istruzione:

```
f *= pow(2, n);
```

si può usare il seguente codice:

```
if (*(int*)&f & 0x7FFFFFFF) { // se f==0 non fare niente
    *(int*)&f += n << 23; // aggiungi n all'esponente
}
```

5.7.5 Dimensione delle celle di array

Assicurati che la dimensione (ottenibile con l'operatore `sizeof`) delle celle non grandi degli array e dei *vector* sia una potenza di due, e che la dimensione delle celle grandi degli array e dei `vector` non sia una potenza di due.

L'accesso diretto alla cella di un array viene fatto moltiplicando l'indice per la dimensione di ogni cella, che è una costante. Se il secondo fattore di questa moltiplicazione è una potenza di due, tale operazione è molto più rapida, in quanto è implementata da uno scorrimento dei bit. Analogamente, negli array multidimensionali, tutte le dimensioni, eccetto al più la prima, dovrebbero essere potenze di due.

Questo dimensionamento si ottiene aggiungendo alle strutture dei campi inutilizzati e agli array delle celle inutilizzate. Per esempio, se ogni cella è una terna di oggetti float, basta aggiungere a ogni cella un quarto oggetto float *dummy* (cioè *fantoccio*).

Tuttavia, nell'accedere alle celle di un array multidimensionale in cui una dimensione diversa dalla prima è una potenza di 2 abbastanza grande, si può cadere nel fenomeno della contesa per la cache dei dati (in inglese *data cache conflict* o *data cache contention*), che può rallentare l'elaborazione fino a più di 10 volte. Questo fenomeno si verifica solo quando le celle dell'array superano una certa dimensione, che dipende dal sistema, ma che orientativamente è da 1 a 8 KB. Pertanto, nel caso in cui un algoritmo deve elaborare un array le cui celle hanno o potrebbero avere come dimensione

una potenza di 2 maggiore o uguale a 1024 byte, in primo luogo si deve scoprire se si ha la contesa per la cache, e in caso affermativo evitare tale fenomeno.

Per esempio, una matrice di 100 x 512 `float` è un array di 100 array di 512 `float`. Ogni cella dell'array di primo livello è grande 512 x 4 = 2048 byte, e quindi è a rischio di contesa per la cache dei dati.

Per scoprire l'esistenza della contesa per la cache, basta aggiungere una cella array di ultimo livello, ma continuare a elaborare le stesse celle di prima, e misurare se il tempo di elaborazione si riduce sostanzialmente (di almeno il 20%). In caso affermativo, si deve fare in modo che tale riduzione ci sia sempre. A tale scopo, si può adottare una delle seguenti tecniche:

- Aggiungere una o alcune celle inutilizzate alla fine di ogni riga. Per esempio l'array `double a[100][1024]` potrebbe essere trasformato in `double a[100][1026]`, anche se nel codice si terrà conto che la dimensione utile rimane 100x1024.
- Lasciare le dimensioni appropriate dell'array, ma suddividere le matrici in blocchi rettangolari, ed elaborare tutte le celle di ogni blocco prima di passare al blocco successivo.

5.7.6 Espansione *inline* esplicita

Se non usi le opzioni di ottimizzazione dell'intero programma e di espansione *inline* automatica, prova a spostare nelle intestazioni e a dichiarare `inline` le funzioni chiamate dai colli di bottiglia.

Come spiegato nella linea-guida "Funzioni espanse *inline*" della sezione 3.1, le singole funzioni espanse *inline* sono più veloci, ma un eccesso di funzioni espanse inline rallenta complessivamente il programma.

Prova a dichiarare `inline` un paio di funzioni per volta, fin tanto che si ottengono miglioramenti significativi della velocità (almeno del 10%) per un dato comando.

5.7.7 Operazioni con potenze di due

Se devi scegliere una costante intera per cui devi moltiplicare o dividere spesso, scegli una potenza di due.

Le operazioni di moltiplicazione, divisione e modulo tra numeri interi sono molto più veloci se il secondo operando è una potenza di due costante, in quanto in tal caso vengono implementate come scorrimenti di bit o mascherature di bit.

5.7.8 Divisione intera per costanti

Se un numero intero `signed` è sicuramente non-negativo, quando lo dividi per una costante, convertilo in `unsigned`.

Se `s` è un intero `signed`, `u` è un intero `unsigned`, e `c` è un'espressione costante intera (positiva o negativa), l'operazione `s / c` è più lenta di `u / c` e l'operazione `s % c` è più lenta di `u % c`, soprattutto quando `c` è una potenza di due, ma anche quando non lo è, in quanto nei primi due casi si deve tenere conto del segno.

D'altra parte, la conversione da `signed` a `unsigned` non costa niente, in quanto si tratta solo di una reinterpretazione degli stessi bit. Pertanto, se `s` è un intero con segno, che sai essere

sicuramente positivo o nullo, ne velocizzi la divisione se usi le seguenti espressioni equivalenti: `unsigned(s) / c` e `unsigned(s) % c`.

5.7.9 Processori con bus dati ridotto

Se il processore target ha il bus dati più piccolo dei registri interni, se possibile, usa tipi interi non più grandi del bus dati per tutte le variabili eccetto i parametri di funzione e le variabili locali più utilizzate.

I tipi `int` e `unsigned int` sono quelli più efficienti, una volta caricati nei registri del processore. Tuttavia, su alcune famiglie di processori, potrebbero non essere i più efficienti da accedere in memoria.

Per esempio, esistono processori che hanno registri da 16 bit, ma bus dati da 8 bit, e altri processori che hanno registri da 32 bit, ma bus dati da 16 bit. Per i processori che hanno il bus dati più piccolo dei registri interni, solitamente i tipi `int` e `unsigned int` corrispondono alla dimensione dei registri interni.

Per tali sistemi, la lettura e la scrittura in memoria di un oggetto di tipo `int` richiedono un tempo maggiore di quello che sarebbe richiesto da un tipo intero non più grande del bus dati.

I parametri di funzione e le variabili locali più utilizzate sono solitamente allocate in registri, e quindi non richiedono accessi in memoria.

Capitolo 6 – Appendice

6.1 GNU Free Documentation License

Version 1.2, November 2002

```
Copyright (C) 2000,2001,2002  Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been

arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher

of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.