4 - C++

ADT Matrice in C e C++

Analizziamo come implementare un tipo di dato astratto per una matrice sia in C che in C++.

La specifica sintattica di questo ADT prevede tre tipi che sono matrice, intero e tipoelem mentre gli operatori fondamentali sono `CreaMatrice()` che restituisce una matrice, `LeggiMatrice(matrice, intero, intero)` che restituisce un tipoelem e `ScriviMatrice(matrice, intero, intero, tipoelem)` che restituisce una matrice.

In C possiamo implementare questa struttura definendo `tipoelem` come `double` e `matrice` come un puntatore a puntatore di `tipoelem`.
Supponendo di avere dieci righe e dieci colonne la funzione `LeggiMatrice` può essere implementata restituendo semplicemente l'elemento in posizione `M[r,c]` mentre la funzione `ScriviMatrice` assegna il valore `e` alla posizione `M[r,c]` della matrice.

Requisito di Astrazione

Il requisito di astrazione richiede che le operazioni su un ADT siano indipendenti dalla sua implementazione.
Consideriamo ad esempio la costruzione di una matrice nulla in C dove una prima implementazione potrebbe utilizzare due cicli annidati che scorrono le righe e le colonne assegnando zero a ogni elemento tramite accesso diretto `M[i,j] = 0`.
Il problema è che questa implementazione viola il requisito di astrazione perché dipende direttamente dalla realizzazione della matrice quindi l'approccio corretto consiste nell'utilizzare la funzione `ScriviMatrice` all'interno dei cicli scrivendo `ScriviMatrice(M, i, j, 0)`

```
nulla(matrice & M){
    for (int i=0; i<righe; i++)
        for (int j=0; j<colonne; j++)
            M[i,j] = 0;
}
```

La versione corretta sarebbe:

```
nulla(matrice & M){
    for (int i=0; i<righe; i++)
        for (int j=0; j<colonne; j++)
```

```
            M[i,j] = 0;
}
```

Un ulteriore problema è che le variabili `righe` e `colonne` che sono proprie dell'astrazione matrice non sono protette né incapsulate nella realizzazione e questo significa che possono essere modificate da qualunque parte del codice. Cosa succederebbe se un programmatore scrivesse `righe++` o addirittura `colonne=-1`? Questo porterebbe a comportamenti imprevedibili e potenzialmente catastrofici per il programma.

Requisito di Protezione

In C è molto facile violare il requisito di protezione come dimostrano diversi esempi pratici. Supponiamo di avere le variabili `matrice A, B`, un array bidimensionale `double C[3][3]`, una variabile `double s` e una variabile `tipoelem e`. Le operazioni `e = LeggiMatrice(A,3,2)` e `ScriviMatrice(B,2,2,e)` non violano il requisito di protezione in quanto utilizzano i tipi corretti.

```
matrice A, B;
double C[3][3];
double s;
tipoelem e;

// Operazioni corrette
e = LeggiMatrice(A,3,2);
ScriviMatrice(B,2,2,e);

// Violazioni del requisito di protezione
s = LeggiMatrice(A,1,1); // s non è di tipo tipoelem
ScriviMatrice(C,2,2,e); // C non è di tipo matrice
A[3,1] = 7.31; // accesso diretto ad A!!!
righe = righe * 2; // modifica diretta di righe
colonne = -1; // modifica diretta di colonne
e = LeggiMatrice(C,1,1); // C non è di tipo matrice
e = 3.0; // anche tipoelem andrebbe protetto!!!
```

Tuttavia esistono numerosi modi per violare questo requisito come ad esempio l'istruzione `s = LeggiMatrice(A,1,1)` che è problematica perché `s` non è di tipo `tipoelem` oppure `ScriviMatrice(C,2,2,e)` che è errata perché `C` non è di tipo `matrice`. L'accesso diretto `A[3,1] = 7.31` bypassa completamente l'interfaccia dell'ADT mentre la modifica diretta `righe = righe * 2` o `colonne = -1` viola l'incapsulamento. Anche l'istruzione `e = LeggiMatrice(C,1,1)` è scorretta perché `C` non è di tipo `matrice` e infine pure l'assegnamento `e = 3.0` evidenzia che il tipo `tipoelem` stesso andrebbe protetto.

Soluzione in C++

Il C++ fornisce tutti gli strumenti necessari per garantire sia il requisito di astrazione che quello di protezione per i nostri ADT e l'ADT matrice può essere realizzata utilizzando una classe che incapsula la struttura dati facendo in modo che i metodi siano applicabili solo su oggetti di tipo matrice.

```
class matrice {
public:
    matrice(){ // definisce una matrice 5x5
        righe = 5;
        colonne = 5;
    }

    tipoelem LeggiMatrice(int r, int c){
        return M[r,c];
    }

    void ScriviMatrice(int r, int c, tipoelem e){
        M[r,c] = e;
    }

private:
    double M[10][10];
    int righe, colonne;
};

// Utilizzo
matrice A();
A.ScriviMatrice(3,4,3.0);
A.righe = 10; // non ammesso. Righe è nella parte private, fallisce la
compilazione
```

La definizione della classe prevede una parte pubblica e una privata dove il costruttore senza parametri inizializza una matrice 5x5 impostando le variabili `righe` e `colonne` a 5. Il metodo `LeggiMatrice(int r, int c)` restituisce l'elemento in posizione `M[r,c]` mentre il metodo `ScriviMatrice(int r, int c, tipoelem e)` assegna il valore `e` alla posizione `M[r,c]`. La parte privata della classe contiene l'array bidimensionale `double M[10][10]` e le variabili `int righe` e `colonne`.

Per utilizzare questa classe si crea un oggetto `matrice A()` e si invocano i metodi sulla classe con ad esempio `A.ScriviMatrice(3,4,3.0)`. Un tentativo di accesso diretto come `A.righe = 10` non è ammesso perché `righe` si trova nella parte privata della classe e il compilatore genererà un errore di compilazione impedendo questo tipo di violazione.

Basics - Fondamenti del Linguaggio

Hello World

Il programma più semplice in C++ è il classico "Hello World" che inizia con la direttiva al preprocessore `#include <iostream>` per includere la libreria per l'input/output. La funzione `main()` è una free function che rappresenta il punto di ingresso del programma e il tipo restituito è un `int` che rappresenta lo status code dove zero indica successo mentre un valore diverso da zero indica un fallimento. Per accedere ai nomi del namespace standard si utilizza il prefisso `std::` mentre l'oggetto `cout` è un oggetto speciale che rappresenta lo schermo e l'operatore `<<` è l'operatore di output che permette di inviare dati verso `cout`.

```
#include <iostream>

int main() {
    std::cout << "Hello world!" << endl;
    return 0;
}
```

Parametri a Riga di Comando

È possibile accedere ai parametri forniti a riga di comando definendo la funzione `main` con la seguente firma `int main (int argc, char* argv[])` dove il parametro `argc` rappresenta il numero di parametri passati incluso il nome del programma stesso mentre il parametro `argv` è un array di stringhe in stile C che contiene i parametri effettivi.

Consideriamo un esempio concreto dove se eseguiamo il programma con il comando `./myprog -a myfile.txt` avremo che `argc` vale 3 mentre `argv[0]` contiene la stringa `"./myprog"`, `argv[1]` contiene `"-a"`, `argv[2]` contiene `"myfile.txt"` e `argv[3]` vale 0 che rappresenta il terminatore dell'array.

```
int main (int argc, char* argv[]) {
    // argc = numero di parametri
    // argv = array di stringhe
}

// Esempio esecuzione: ./myprog -a myfile.txt
// argc = 3
```

```
// argv[0] = "./myprog"
// argv[1] = "-a"
// argv[2] = "myfile.txt"
// argv[3] = 0
```

Commenti

In C++ esistono due stili per i commenti dove i commenti multi-linea utilizzano la sintassi `/* ... */` e possono estendersi su più righe mentre i commenti su singola riga utilizzano invece la sintassi `//` e si estendono fino alla fine della riga corrente. I commenti sono fondamentali per documentare il codice e spiegare la logica implementativa rendendo il codice più comprensibile sia per noi stessi che per altri sviluppatori.

```
/* To calculate the final grade, we sum all the weighted
   midterm and homework scores and then divide by the number
   of scores to assign a percentage. This percentage is
   used to calculate a letter grade. */

// To generate a random item, we're going to do the following:
// 1) Put all of the items of the desired rarity on a list
// 2) Calculate a probability for each item based on level and weight
factor
// 3) Choose a random number
// 4) Figure out which item that random number corresponds to
// 5) Return the appropriate item
```

Variabili

Una variabile in C++ è il nome associato a una porzione di memoria e la dichiarazione `int x;` crea una variabile di tipo intero. È possibile stampare il valore di una variabile utilizzando l'istruzione `cout << x;` mentre in C++ le variabili sono note come l-value (left side) ovvero valori che hanno un indirizzo di memoria associato in contrapposizione con gli r-value (right side) che si riferiscono a valori che vengono assegnati a un l-value.

```
int x;
cout << x;
```

Variabili Statiche

Un attributo di una classe in C++ può essere dichiarato statico analogamente a quanto avviene in Java e quando un attributo è statico esiste una sola variabile condivisa per tutti gli oggetti della classe. In C++ anche le funzioni possono avere variabili statiche dove una

variabile statica all'interno di una funzione viene creata alla prima chiamata della funzione e il suo valore viene mantenuto nelle successive chiamate.

Un esempio pratico è il conteggio del numero di chiamate di una funzione dove definendo una variabile statica `counter` inizializzata a zero e incrementandola ad ogni chiamata è possibile tenere traccia di quante volte la funzione è stata invocata durante l'esecuzione del programma.

```
void f() {
    static int counter = 0;
    counter++;
    …
}
```

Assegnamenti e R-value

Gli assegnamenti in C++ seguono regole precise dove l'istruzione `int y;` dichiara `y` come variabile intera e successivamente `y = 4;` assegna il valore 4 a `y`. L'espressione `y = 2 + 5;` valuta prima la somma che risulta 7 e poi assegna questo valore a `y`. Dichiarando una nuova variabile `int x;` e scrivendo `x = y;` il valore 7 viene assegnato a `x` mentre l'assegnamento `x = x;` è perfettamente legale e assegna a `x` il suo stesso valore quindi 7. Infine l'espressione `x = x + 1;` valuta prima la somma `x + 1` che risulta 8 e poi assegna questo nuovo valore a `x`.

```
int y; // dichiara y come variabile integer
y = 4; // 4 viene assegnato a y
y = 2 + 5; // 2 + 5 è uguale a 7, assegnato a y
int x; // dichiara x come variabile integer
x = y; // y è uguale a 7, assegnato a x
x = x; // x è uguale a 7, assegnato a x
x = x + 1; // x + 1 è uguale a 8, assegnato a x
```

Cin

L'oggetto `cin` è l'opposto di `cout` e permette di leggere input dalla console. Un programma che chiede all'utente di inserire un numero utilizza `cout` per stampare il messaggio "Enter a number:" e poi dichiara una variabile intera `x` utilizzando `cin >> x;` per leggere il numero dalla console e assegnarlo a `x` per infine stampare il valore inserito con un messaggio appropriato. L'operatore `>>` estrae i dati da `cin` e li assegna alla variabile specificata permettendo l'interazione con l'utente.

```cpp
#include <iostream>

int main() {
    using namespace std;
    cout << "Enter a number: ";
    int x;
    cin >> x;
    cout << "You entered " << x << " and we do not like "<< x << endl;
    return 0;
}
```

Funzioni

Le funzioni in C++ sono blocchi di codice riutilizzabili che possono essere chiamati da diverse parti del programma. Una funzione chiamata `doPrint()` di tipo `void` che non restituisce alcun valore può stampare un messaggio sullo schermo mentre la funzione `main()` può invocare `doPrint()` semplicemente scrivendone il nome seguito dalle parentesi. Quando il programma viene eseguito prima viene stampato "Starting main()" poi viene chiamata `doPrint()` che stampa "In doPrint()" e infine viene stampato "Ending main()" prima che il programma termini restituendo zero per indicare l'esecuzione corretta.

```cpp
// Declaration of function doPrint()
void doPrint()
{
    using namespace std;
    cout << "In doPrint()" << endl;
}

// Declaration of main()
int main()
{
    using namespace std;
    cout << "Starting main()" << endl;
    doPrint(); // chiamata a DoPrint()
    cout << "Ending main()" << endl;
    return 0;
}
```

Parametri di Funzione

Le funzioni possono accettare parametri per operare su dati diversi rendendo il codice più flessibile e riutilizzabile. Una funzione `add(int x, int y)` restituisce la somma di `x` e `y` mentre una funzione `multiply(int z, int w)` restituisce il prodotto di `z` e `w`. Nella funzione `main` è possibile invocare queste funzioni in vari modi come con valori letterali tipo `add(4, 5)` oppure con variabili come `add(a, b)` dove `a` e `b` sono state precedentemente dichiarate o anche con chiamate annidate come `add(1, multiply(2, 3))` dove il risultato di `multiply(2, 3)` viene passato come secondo parametro ad `add` permettendo di comporre operazioni complesse.

```cpp
#include <iostream>

int add(int x, int y){
    return x + y;
}

int multiply(int z, int w){
    return z * w;
}

int main(){
    using namespace std;
    cout << add(4, 5) << endl;
    cout << add(3, 6) << endl;
    cout << add(1, 8) << endl;

    int a = 3;
    int b = 5;
    cout << add(a, b) << endl;
    cout << add(1, multiply(2, 3)) << endl;
    cout << add(1, add(2, 3)) << endl;

    return 0;
}
```

Forward Declaration

In C++ il compilatore legge il codice dall'alto verso il basso quindi una funzione deve essere dichiarata prima di essere utilizzata altrimenti il compilatore non saprebbe della sua esistenza. Se vogliamo chiamare una funzione `add` nella funzione `main` ma definire il corpo di `add` dopo `main` dobbiamo utilizzare una forward declaration che consiste nel dichiarare il prototipo della funzione prima di `main` scrivendo `int add(int x, int y);`. In

questo modo il compilatore sa che la funzione esiste e può compilare correttamente la chiamata in `main` anche se l'implementazione effettiva della funzione viene fornita successivamente nel codice.

```
#include <iostream>

int add(int x, int y); // forward declaration prototype

int main()
{
    using namespace std;
    cout << "The sum of 3 and 4 is: " << add(3, 4) << endl;
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

Dati Primitivi

I tipi di dati primitivi in C++ includono `char` per i caratteri singoli, `int`, `short` e `long` per i numeri interi di diverse dimensioni, `double` e `float` per i numeri in virgola mobile e `bool` per i valori booleani. Nel vecchio C++ gli interi venivano spesso utilizzati per rappresentare valori booleani dove qualsiasi valore diverso da zero era considerato vero mentre zero era considerato falso.

Un esempio di questa pratica è dichiarare una variabile intera `a` e leggerla con `cin` per poi utilizzarla in una condizione `if (a)` che è equivalente a scrivere `if (a != 0)` quindi se `a` vale zero la condizione è falsa altrimenti è vera. Similmente quando si lavora con puntatori un'istruzione come `while (p)` è equivalente a `while (p != 0)` dove zero rappresenta il puntatore nullo che in C++98 si indicava con 0 mentre nelle versioni più recenti si usa `nullptr` per rendere il codice più chiaro e sicuro.

```
// Utilizzo di int come booleano (vecchio C++)
int a;
cin >> a;
if (a) { ... } // equivalent to if (a != 0)

Point* p = list.getFirstPoint();
```

```
while (p) { ... } // equivalent to while (p != 0)
// 0 is 'nullptr' for C++98
```

Reference (Riferimenti)

Un riferimento in C++ è un nome alternativo per identificare un oggetto, essenzialmente un alias che permette di accedere allo stesso oggetto attraverso un nome diverso. I riferimenti sono comunemente utilizzati come parametri di funzione per evitare copie costose o per permettere modifiche agli argomenti passati. Una caratteristica fondamentale dei riferimenti è che devono essere inizializzati al momento della creazione e, una volta inizializzati, non possono essere modificati per riferirsi a un altro oggetto. Non esistono riferimenti null in C++, a differenza dei puntatori.

```
int ival = 1024;
// indirizzo di refval impostato all'indirizzo di ival
int& refval = ival;
refval += 2; // cosa contiene refval dopo questa istruzione?
int ii = refval;
```

Nell'esempio sopra, dopo l'istruzione `refval += 2`, sia `refval` che `ival` conterranno il valore 1026, poiché entrambi si riferiscono allo stesso oggetto in memoria.

Passaggio di Parametri

In C++, i parametri delle funzioni sono passati per valore di default, il che significa che viene creata una copia del valore del parametro reale. Quando si desidera modificare il valore del parametro reale all'interno della funzione, è necessario passare l'argomento per riferimento.

```
void swap(int &v1, int &v2) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}

int main() {
    int i = 10;
    int j = 20;
    swap(i, j);
}
```

Il passaggio per riferimento è particolarmente utile anche quando si devono passare oggetti di grandi dimensioni senza crearne una copia, migliorando così le prestazioni del

programma. La vecchia modalità che effettua una copia delle stringhe sarebbe `bool isShorter(string s1, string s2)`, mentre la nuova modalità più efficiente che non effettua copie è:

```
bool isShorter(const string& s1, const string& s2) {
    return s1.size() < s2.size();
}
```

La parola chiave `const` è essenziale quando non si vuole modificare l'oggetto passato per riferimento. Senza `const`, non si potrebbe passare una constant string alla funzione e si potrebbe inavvertitamente cambiare lo stato di un oggetto all'interno della funzione. La regola generale è di passare sempre per const reference a meno che non si necessiti esplicitamente di modificare il parametro reale.

Puntatori

Un puntatore in C++ è una variabile che memorizza l'indirizzo di memoria di un'altra variabile, puntando quindi a una locazione di memoria specifica. I puntatori possono far riferimento a qualsiasi tipo di dato e vengono dichiarati usando l'asterisco tra il tipo e il nome della variabile.

```
int* ip1; // can also be written int *ip1
Point* p = nullptr; // pointer to object of class Point
// nullptr is "no object", same as Java's null.
// nullptr is new in C++11, C++98 used 0
```

Un modo per ottenere un valore puntatore è utilizzare l'operatore indirizzo `&` prima del nome della variabile:

```
int ival = 1024;  // ival vale 1024 - &ival contiene l'indirizzo di ival
int* ip2 = &ival; // ip2 contiene l'indirizzo - *ip2 contiene 1024
```

Dereferenziazione

Il contenuto della memoria a cui punta un puntatore è accessibile attraverso l'operatore di dereferenziazione `*`. Questo operatore permette di accedere e modificare il valore memorizzato all'indirizzo contenuto nel puntatore.

```
int ival = 1024;
int* ip2 = &ival;
```

```
cout << *ip2 << endl; // prints 1024
*ip2 = 1025;
```

Le differenze principali tra puntatori e riferimenti sono che i puntatori possono essere dereferenziati mentre i riferimenti no, i puntatori possono essere indefiniti o null mentre i riferimenti devono sempre riferirsi a un oggetto valido, e i puntatori possono essere modificati per puntare a oggetti diversi mentre i riferimenti rimangono legati all'oggetto originale.

Alias di Tipi

C++ permette di definire nuovi nomi di tipo come sinonimi di tipi esistenti. Tradizionalmente si usa `typedef`, ma nel nuovo standard C++11 esiste una dichiarazione di alias più leggibile con `using`.

```
using newType = existingType; // C++11
typedef existingType newType; // equivalent, still works

// Esempi
typedef unsigned long counter_type;
typedef std::vector<int> table_type;
using counter_type = unsigned long;
using table_type = std::vector<int>;
```

Auto (C++11)

La parola chiave `auto` introdotta in C++11 permette al compilatore di dedurre automaticamente il tipo di una variabile dal suo inizializzatore. Questo è particolarmente utile quando i nomi di tipo sono lunghi o complessi, come nel caso degli iteratori.

```
vector<int> v;
// ...
auto it = v.begin(); // begin() returns vector<int>::iterator
```

È importante non abusare di `auto` quando il tipo è ovvio, come nel caso dei letterali. Per esempio, `auto sum = 0;` è considerato poco appropriato poiché è chiaro che `sum` è di tipo `int`.

Array

Bjarne Stroustrup ha affermato che "The C array concept is broken and beyond repair", e infatti i programmi C++ moderni utilizzano normalmente `vector` invece degli array built-in. Gli array in C++ non hanno verifiche sulla indicizzazione a runtime, il che significa che con

un indice errato è possibile accedere e potenzialmente distruggere elementi fuori dall'array. Esistono due modi principali di allocare array: sullo stack o sullo heap.

Array allocato sullo stack:

```
void f() {
    int a = 5;
    int x[3]; //size must be a compile-time constant
    for (size_t i = 0; i != 3; ++i) {
        x[i] = (i + 1) * (i + 1);
    }
    // ...
}
```

Quando si usa il nome di un array, il compilatore spesso lo sostituisce con un puntatore al primo elemento, rendendo possibili le seguenti assegnazioni:

```
int* px1 = x;
int* px2 = &x[0];
```

È possibile utilizzare puntatori per accedere agli elementi di un array, dove i puntatori fungono da iteratori:

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = x; px != x + 5; ++px) {
    cout << *px << endl;
}
```

Quando si incrementa un puntatore, gli incrementi sono in base alla dimensione del tipo di dato puntato, quindi `px + 1` significa `px + sizeof(T)` per un array di tipo T. È anche possibile sottrarre due puntatori per ottenere il numero di elementi tra di essi.

Begin e End (C++11)

Con C++11 sono state introdotte le funzioni `begin` e `end` che possono essere utilizzate anche con gli array built-in, non solo con i container della libreria standard.

```
int x[] = {0, 2, 4, 6, 8};
for (int* px = begin(x); px != end(x); ++px) {
    cout << *px << endl;
}
```

Array su Heap

Gli array allocati sull'heap sono simili agli array in Java e permettono una dimensione dinamica determinata a runtime.

```
void g(size_t n) {
    int a;
    int* px = new int[n]; // dynamic size >= 0
    for (size_t i = 0; i < n; ++i) {
        px[i] = (i + 1) * (i + 1);
    }
    // ...
    delete[] px; // note the [ ]
}
```

È importante notare che gli heap-allocated array si accedono tramite puntatori, non contengono informazioni sulla loro lunghezza, le funzioni iteratore `begin()` e `end()` non possono essere utilizzate con essi, e `delete[]` è necessaria per deallocare correttamente l'array altrimenti gli oggetti nell'array non verranno distrutti.

Casting

In C++ la conversione di tipo può essere implicita, ma è preferibile utilizzare cast espliciti per chiarezza e sicurezza del codice.

```
d = 35.67;
int x = d; // x == 35; implicit
int x = static_cast<int>(d); // explicit cast preferito
```

Oltre al cast statico, C++ offre diversi tipi di cast: `dynamic_cast<type>(pointer)` per il downcasting in una gerarchia di ereditarietà, `const_cast<type>(variable)` per rimuovere la constness da una variabile (usato raramente), e `reinterpret_cast<type>(expr)` per reinterpretare il pattern di bit in modo diverso (usato nella programmazione di basso livello). Il casting alla C con la sintassi `(int)d` è permesso ma sconsigliato in C++.

Vector

La classe `vector` è una classe template per la memorizzazione di elementi di tipo arbitrario, rappresentando l'alternativa moderna e sicura agli array C-style.

```
#include <iostream>
#include <vector>
```

```
#include <string>
using namespace std;

int main() {
    vector<string> v;
    string word;
    while (cin >> word)
        v.push_back(word);
    for (int i = v.size() - 1; i >= 0; --i)
        cout << v[i] << endl;
}
```

Un vector è inizialmente vuoto e gli elementi vengono aggiunti in coda con il metodo `push_back`. Lo spazio per la memorizzazione viene allocato automaticamente secondo necessità. Si accede agli elementi con l'operatore `[]`, i vector possono essere copiati con `V1 = V2` e confrontati con `V1 == V2`.

Iteratori

Gli iteratori sono il modo standard per accedere agli elementi dei container della libreria standard. Un iteratore "punta" a uno degli elementi della collection o a una posizione immediatamente successiva all'ultimo elemento. Si può dereferenziare con `*` per accedere all'elemento puntato e può essere incrementato con `++` per passare al successivo elemento. I container hanno i metodi `begin()` e `end()` che restituiscono rispettivamente un iteratore al primo elemento e uno alla posizione dopo l'ultimo elemento.

Scansionare un vettore:

```
vector<int> v;
// ...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    *it = 0;
```

Meglio con auto (C++11):

```
for (auto it = v.begin(); it != v.end(); ++it)
    *it = 0;
```

Ancor meglio con un for range-based (C++11):

```
for (int& e : v)
```

```
    e = 0;
```

Aritmetica degli Iteratori

Gli iteratori di vector supportano operazioni aritmetiche che permettono di navigare efficacemente nel container. Per esempio, per trovare il primo numero negativo nella seconda metà del vector:

```
auto it = v.begin() + v.size() / 2;
while (it != v.end() && *it >= 0)
    ++it;
if (it != v.end())
    cout << "Found at index " << it - v.begin() << endl;
else
    cout << "Not found" << endl;
```

Iteratori Const

Un normale iteratore può essere utilizzato sia per leggere che per scrivere elementi, mentre un `const_iterator` può solo leggere. Le funzioni `cbegin()` e `cend()`, introdotte in C++11, restituiscono iteratori const.

```
vector<int> v;
for (auto it = v.cbegin(); it != v.cend(); ++it)
    cout << *it << endl;
```

Quando un container è un oggetto costante, anche `begin()` e `end()` restituiscono automaticamente iteratori costanti:

```
void f(const vector<int>& v) {
    for (auto it = v.begin(); it != v.end(); ++it)
        *it = 0; // Wrong -- 'it' is a constant iterator
}
```

Classi - Basics

Le classi in C++ permettono di incapsulare dati e funzionalità. Ecco un esempio di una classe Point dove le coordinate non possono essere negative:

```
class Point {
public:
```

```
    using coord_t = unsigned int;
    Point(coord_t ix, coord_t iy);
    coord_t get_x() const;
    coord_t get_y() const;
    void move_to(coord_t new_x, coord_t new_y);
private:
    coord_t x;
    coord_t y;
};
```

Si noti l'alias di tipo pubblico che permette agli utenti della classe di utilizzare quel nome. Le funzioni accessorie che non cambiano lo stato dell'oggetto dovrebbero essere dichiarate `const`.

Funzioni Membro

L'implementazione delle funzioni membro può essere fornita separatamente dalla dichiarazione della classe:

```
Point::Point(coord_t ix, coord_t iy) : x(ix), y(iy) {}
Point::coord_t Point::get_x() const { return x; }
Point::coord_t Point::get_y() const { return y; }
void Point::move_to(coord_t new_x, coord_t new_y) {
    x = new_x;
    y = new_y;
}
```

È importante sapere che `this` è un puntatore all'oggetto corrente. Una `struct` è come una classe ma con tutti i membri pubblici di default. Quando due classi fanno riferimento l'una all'altra, è necessaria una forward declaration:

```
class B; // class declaration, "forward declaration"
class A {
    B* pb;
};
class B {
    A* pa;
};
```

Esempio Fibonacci

Ecco un esempio di una classe che calcola i numeri di Fibonacci:

```
class Fibonacci {
public:
    Fibonacci();
    unsigned int value(unsigned int n) const;
};

unsigned int Fibonacci::value(unsigned int n) const {
    int nbr1 = 1;
    int nbr2 = 1;
    for (unsigned int i = 2; i < n; ++i) {
        int temp = nbr1 + nbr2;
        nbr1 = nbr2;
        nbr2 = temp;
    }
    return nbr2;
}
```

Per migliorare l'efficienza usando una cache, possiamo modificare la classe utilizzando la parola chiave `mutable` che permette di modificare membri anche in funzioni const:

```
class Fibonacci {
public:
    // ... (come prima)
private:
    mutable vector<unsigned int> values;
};

Fibonacci::Fibonacci() {
    values.push_back(1);
    values.push_back(1);
}

unsigned int Fibonacci::value(unsigned int n) const {
    if (n < values.size()) // dovrei anche controllare che n > 0
        return values[n-1];
    for (unsigned int i = values.size(); i < n; ++i)
        values.push_back(values[i - 1] + values[i - 2]);
```

```
    return values[n-1];
}
```

Inizializzazioni

I membri di una classe possono essere inizializzati in tre modi diversi:

```
class A {
    // ...
private:
    int x = 123; // direct initialization, new in C++11
    const int b;
};

A::A(int ix) : x(ix) {} // constructor initializer
A::A(int ix) { x = ix; } // assignment
```

Il constructor initializer è da preferire. I membri che sono riferimenti o const non possono essere assegnati e devono essere inizializzati attraverso la lista di inizializzazione.

Delega ad Altri Costruttori (C++11)

Un costruttore può delegare le inizializzazioni ad altri costruttori della stessa classe:

```
class Complex {
public:
    Complex(double r, double i) : re(r), im(i) {}
    Complex(double r) : Complex(r, 0) {}
    Complex() : Complex(0, 0) {}
private:
    double re;
    double im;
};
```

In questo esempio si potrebbero anche usare parametri di default: `Complex(double r = 0, double i = 0) : re(r), im(i) {}`.

Membri Statici

I membri statici appartengono alla classe piuttosto che alle singole istanze. Ecco una classe che conta il numero di oggetti creati:

```
class Counted {
public:
    Counted() { ++nbrObj; }
    ~Counted() { --nbrObj; }
    static unsigned int getNbrObj() { return nbrObj; }
private:
    static unsigned int nbrObj;
};
unsigned int Counted::nbrObj = 0;
```

Un membro statico deve essere inizializzato fuori dalla classe.

New e Delete

Gli operatori `new` e `delete` vengono tradotti in chiamate alle funzioni di allocazione/deallocazione della memoria:

```
void* operator new(size_t bytes);
void operator delete(void* p) noexcept;
```

Esempio di utilizzo:

```
Point* p = new Point(10, 20);
// allocate raw memory, initialize the object
// Point* p = static_cast<Point*>(::operator new(sizeof(Point)));
// p->Point::Point(10, 20);
delete p;
// destruct the object, free memory
// p->~Point();
// ::operator delete(p);
```

Copia di Oggetti

Gli oggetti vengono copiati in diversi contesti nel programma. Durante le inizializzazioni:

```
Person p1("Bob");
Person p2(p1);
Person p3 = p1;
```

Nel passaggio per valore:

```
void f(Person p) { ... }
f(p1);
```

Nell'assegnamento:

```
Person p4("Alice");
p4 = p1;
```

Nelle funzioni che restituiscono un valore:

```
Person g() {
    Person p5("Joe");
    // ...
    return p5;
}
```

Inizializzazione e Assegnamento

È importante distinguere tra inizializzazione e assegnamento poiché vengono gestiti diversamente:

```
Person p1("Bob");
Person p3 = p1; // initialization
Person p4("Alice");
p4 = p1; // assignment
```

L'inizializzazione crea un nuovo oggetto inizializzato con una copia di un altro oggetto, gestita dal costruttore di copia `Classname(const Classname&)`. L'assegnamento sovrascrive un oggetto esistente con una copia di un altro oggetto, gestito dall'operatore di assegnamento `Classname& operator=(const Classname&)`.

Funzioni Copia

Quando una classe non definisce esplicitamente costruttore di copia e operatore di assegnamento, il compilatore ne sintetizza versioni di default che effettuano una copia membro a membro. Questo è sufficiente per classi che non gestiscono risorse dinamiche:

```
class Person {
public:
    // this is the copy constructor that the compiler
    // creates for you, and you cannot write a better one
```

```
    Person(const Person& p) : name(p.name), age(p.age) {}
private:
    string name;
    unsigned int age;
};
```

Una Classe String

Vediamo un esempio di classe che gestisce memoria dinamica:

```
class String {
public:
    String(const char* s) : chars(new char[strlen(s) + 1]) {
        strcpy(chars, s); // copy s to chars
    }
    ~String() { delete[] chars; }
private:
    char *chars;
};
```

Problemi Senza Costruttore di Copia

Senza un costruttore di copia personalizzato, il compilatore effettua una copia superficiale:

```
void f() {
    String s1("abc");
    String s2 = s1;
}
```

In questo caso `s1.chars` e `s2.chars` punteranno alla stessa area di memoria, causando un errore quando entrambi i distruttori tenteranno di deallocare la stessa memoria.

Nel passaggio per valore:

```
void f(String s) {
    // ...
}
void g() {
    String s1("abc");
```

```
    f(s1);
}
```

Quando `s` viene distrutto all'uscita da `f`, l'area di memoria viene deallocata ma `s1.chars` continua a puntare a quella memoria ormai invalida.

Definire un Costruttore di Copia

La soluzione è implementare un costruttore di copia che effettua una copia profonda:

```
String(const String& rhs): chars(new char[strlen(rhs.chars) + 1]) {
    strcpy(chars, rhs.chars);
}
```

Questo costruttore alloca nuova memoria e copia il contenuto, evitando che due oggetti condividano la stessa area di memoria.

Problemi con l'Assegnamento

Problemi simili si verificano con l'assegnamento:

```
void f() {
    String s1("abc");
    String s2("xy");
    s2 = s1;
}
```

Senza un operatore di assegnamento personalizzato, `s1.chars` e `s2.chars` punteranno alla stessa area e il riferimento al vettore "xy" verrà perso causando un memory leak.

Sovraccaricare l'Operatore di Assegnamento

L'istruzione `s1 = s2` viene convertita dal compilatore in `s1.operator=(s2)`. Ecco l'implementazione corretta:

```
String& String::operator=(const String& rhs) {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = new char[strlen(rhs.chars) + 1];
    strcpy(chars, rhs.chars);
```

```
    return *this;
}
```

I dettagli importanti sono: cancellare il vecchio stato per evitare memory leak, restituire l'oggetto `*this` per permettere assegnamenti concatenati, e verificare gli auto-assegnamenti con `if (&rhs == this)`.

Spostare Oggetti

Il costruttore di copia fa una copia profonda, ma ci sono casi in cui l'oggetto sorgente non verrà più usato. Per queste situazioni C++11 introduce il costruttore di spostamento. I valori temporanei possono essere spostati poiché verranno distrutti dopo l'uso:

```
String s1("abc");
String s2("def");
String s3 = s1 + s2; // the result of '+' is a temporary value

void f(String s);
f("abcd"); // f(String("abcd")), the argument is temporary

String g() {
    // ...
    return ...; // the return value is a temporary
}
```

Un lvalue è persistente (variabili) mentre un rvalue non lo è. C++11 introduce gli rvalue reference:

```
String s1("abc");
String s2("def");
String& sref = s1; // reference bound to a variable
String&& srr = s1 + s2; // rvalue reference bound to a temporary
```

Il costruttore di spostamento:

```
String(String&& rhs) noexcept : chars(rhs.chars) {
    rhs.chars = nullptr;
}
```

E l'operatore di assegnamento per movimento:

```
String& operator=(String&& rhs) noexcept {
    if (&rhs == this) {
        return *this;
    }
    delete[] chars;
    chars = rhs.chars;
    rhs.chars = nullptr;
}
```

Idiomi di Costruzione

Quando una classe gestisce risorse dinamiche deve implementare la "regola dei cinque": un distruttore, un costruttore di copia, un operatore di assegnamento, un costruttore di spostamento e un operatore di assegnamento per movimento. Il costruttore deve inizializzare le componenti, il costruttore di copia deve fare una copia profonda, l'operatore di assegnamento deve rilasciare le vecchie risorse, e il distruttore rilascia tutte le risorse.

Riepilogo delle firme:

- Costruttore di copia: `ClassName(const ClassName& rhs)` - attivato quando si crea un nuovo oggetto da uno esistente
- Operatore di assegnamento: `ClassName& operator=(const ClassName& rhs)` - attivato quando si assegna un nuovo valore a un oggetto esistente
- Costruttore di spostamento: `ClassName(ClassName&& rhs) noexcept` - attivato quando si crea un oggetto da un temporaneo
- Operatore di assegnamento per movimento: `ClassName& operator=(ClassName&& rhs) noexcept` - attivato quando si assegna un temporaneo a un oggetto esistente

Vettori di Oggetti

È preferibile usare `vector` invece degli array C-style:

```
vector<Person> v; // vector of Person objects
Person p1("Bob");
v.push_back(p1); // p1 is copied
v.push_back(Person("Alice")); // the object is moved
// ...
for (const auto& p : v) {
    cout << p.getName() << endl;
}
```

Vettori di Puntatori ad Oggetti

Si possono memorizzare puntatori ad oggetti sull'heap:

```
vector<Person*> pv;
Person* p = new Person("Bob");
pv.push_back(p);
// ...
for (auto pptr : pv) {
    cout << pptr->getName() << endl;
}
// ...
for (auto pptr : pv) {
    delete pptr;
}
```

Memory Leaks

Un memory leak si verifica quando la memoria allocata dinamicamente non viene mai deallocata. La memoria resta allocata finché non viene esplicitamente deallocata, ma i puntatori hanno regole di visibilità:

```
void doSomething(){
    int *pnValue = new int;
}
```

Questa funzione alloca un intero ma non lo dealloca mai. Quando la funzione termina, `pnValue` esce dallo scope e l'area di memoria allocata diventa irraggiungibile.

Memory leak si verifica anche con riassegnamento di puntatori:

```
int nValue = 5;
int *pnValue = new int;
pnValue = &nValue; // old address lost, memory leak results
```

O con doppia allocazione:

```
int *pnValue = new int;
pnValue = new int; // old address lost, memory leak results
```

Sovraccarico di Funzioni

Il sovraccarico permette di avere funzioni con lo stesso nome ma parametri differenti:

```
int add(int nX, int nY){
    return nX + nY;
}

double add(double nX, double nY){
    return nX + nY;
}

double add(double nX, double nY, double nZ){
    return nX + nY + nZ;
}
```

Puntatori a Funzione

I puntatori a funzione puntano a una funzione invece che a una variabile. Una funzione come `int foo()` è essa stessa un puntatore costante. Possiamo dichiarare puntatori non costanti: `int (*pFoo)()` è un puntatore a una funzione senza parametri che restituisce un int.

Esempio con selection sort:

```
bool ascending(int nX, int nY) {
    return nX < nY;
}

void selectionSort(int *anArray, int nSize) {
    using namespace std;
    for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++) {
        int nBestIndex = nStartIndex;
        for (int nIndex = nStartIndex + 1; nIndex < nSize; nIndex++){
            if (ascending(anArray[nIndex], anArray[nBestIndex]))
                nBestIndex = nIndex;
        }
        swap(anArray[nStartIndex], anArray[nBestIndex]);
    }
}
```

Versione con puntatore a funzione come parametro:

```
#include <algorithm> // for swap
void selectionSort(int *anArray, int nSize, bool (*pComparison)(int, int))
{
    using namespace std;
    for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++) {
        int nBestIndex = nStartIndex;
        for (int nIndex = nStartIndex + 1; nIndex < nSize; nIndex++) {
            if (pComparison(anArray[nCurrentIndex], anArray[nBestIndex]))
                nBestIndex = nCurrentIndex;
        }
        swap(anArray[nStartIndex], anArray[nBestIndex]);
    }
}
```

Utilizzo con diverse funzioni di confronto:

```
bool ascending(int nX, int nY){
    return nY > nX;
}

bool descending(int nX, int nY){
    return nY < nX;
}

void printArray(int *pArray, int nSize){
    for (int iii=0; iii < nSize; iii++)
        cout << pArray[iii] << " ";
    cout << endl;
}

int main(){
    using namespace std;
    int anArray[9] = { 3, 7, 9, 5, 6, 1, 8, 2, 4 };
    selectionSort(anArray, 9, descending);
    printArray(anArray, 9);
    selectionSort(anArray, 9, ascending);
    printArray(anArray, 9);
    return 0;
}
```

Funzioni Template

Le funzioni template permettono di scrivere codice generico che funziona con diversi tipi:

```
int max(int nX, int nY){
    return (nX > nY) ? nX : nY;
}

double max(double nX, double nY){
    return (nX > nY) ? nX : nY;
}
```

Invece di duplicare il codice, possiamo usare un template con parametri di tipo:

```
template <typename Type> // this is the template parameter declaration
Type max(Type tX, Type tY){
    return (tX > tY) ? tX : tY;
}
```

La parola chiave `template` indica al compilatore che segue una lista di parametri template racchiusi tra `<>`. Per definire un parametro template si usa `typename` o `class` seguito dal nome del segnaposto. Il template può essere utilizzato con diversi tipi:

```
int nValue = max(3, 7); // returns 7
double dValue = max(6.34, 18.523); // returns 18.523
char chValue = max('a', '6'); // returns 'a'
```

Il tipo utilizzato nel template deve supportare l'operatore `>` utilizzato nella funzione:

```
class Cents{
private:
    int m_nCents;
public:
    Cents(int nCents) : m_nCents(nCents) { }
    friend bool operator>(Cents &c1, Cents&c2) {
        return (c1.m_nCents > c2.m_nCents) ? true: false;
    }
};
```

La parola chiave `friend` indica una funzione che può accedere ai membri protetti e privati della classe.

Classi Template

Le classi template permettono di creare classi generiche che funzionano con diversi tipi. Ecco un esempio di array di interi:

```
#include <assert.h> // for assert()
class IntArray {
private:
    int m_nLength;
    int *m_pnData;
public:
    IntArray(int nLength) {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }
    ~IntArray() { delete[] m_pnData; }
    int& operator[](int nIndex) {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
    }
    int GetLength() { return m_nLength; }
};
```

Versione template che funziona con qualsiasi tipo:

```
#include <assert.h> // for assert()
template <typename T>
class Array{
private:
    int m_nLength;
    T *m_ptData;
public:
    Array(int nLength) {
        m_ptData= new T[nLength];
        m_nLength = nLength;
    }
    ~Array() { delete[] m_ptData; }
    T& operator[](int nIndex) {
```

```
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_ptData[nIndex];
    }
    int GetLength(){return m_nLength;}
};
```

Utilizzo della classe template:

```
int main() {
    Array<int> anArray(12);
    Array<double> adArray(12);
    for (int nCount = 0; nCount < 12; nCount++) {
        anArray[nCount] = nCount;
        adArray[nCount] = nCount + 0.5;
    }
    for (int nCount = 11; nCount >= 0; nCount--;)
        std::cout << anArray[nCount] << "\t" << adArray[nCount] <<
std::endl;
    return 0;
}
```

Specializzazione di Template

A volte è utile fornire un'implementazione specifica di un metodo per un tipo particolare:

```
using namespace std;
template <typename T>
class Storage{
private:
    T m_tValue;
public:
    Storage(T tValue) {
        m_tValue = tValue;
    }
    ~Storage() { }
    void Print() {
        std::cout << m_tValue << std::endl;
    }
};
```

Per stampare i double in formato scientifico, possiamo specializzare il metodo `Print`:

```
void Storage<double>::Print() {
    std::cout << std::scientific << m_tValue << std::endl;
}
```

Eccezioni

C++ gestisce le eccezioni con i costrutti `throw`, `try` e `catch`. L'istruzione `throw` segnala che si è verificata un'eccezione:

```
throw -1; // throw a literal integer value
throw ENUM_INVALID_INDEX; // throw an enum value
throw "Can not take square root of negative number"; // throw a literal
char* string
throw dX; // throw a double variable that was previously defined
throw MyException("Fatal Error"); // Throw an object of class MyException
```

Le eccezioni vanno catturate all'interno di un blocco `try`:

```
#include "math.h" // for sqrt() function
using namespace std;
int main(){
    cout << "Enter a number: ";
    double dX;
    cin >> dX;
    try {
        if (dX < 0.0)
            throw "Can not take sqrt of negative number";
        cout << "The sqrt of " << dX << " is " << sqrt(dX) << endl;
    }
    catch (char* strException) {
        cerr << "Error: " << strException << endl;
    }
}
```

Esempio con funzione modulare:

```
#include "math.h" // for sqrt() function
using namespace std;
```

```
double MySqrt(double dX){
    if (dX < 0.0)
        throw "Can not take sqrt of negative number";
    return sqrt(dX);
}

int main(){
    cout << "Enter a number: ";
    double dX;
    cin >> dX;
    try {
        cout << "The sqrt of " << dX << " is " << MySqrt(dX) << endl;
    }
    catch (char* strException) {
        cerr << "Error: " << strException << endl;
    }
}
```

Schema Chiamante/Chiamato per le Eccezioni

Il pattern tipico per la gestione delle eccezioni prevede che la funzione chiamante gestisca le eccezioni lanciate dalla funzione chiamata:

```
// ... funzioneChiamante(....)
{
    bool successo = false;
    while(!successo) {
        // chiedi i dati all'utente o acquiscili da altra sorgente
        try {
            // ...
            FunzioneChiamata(...)
            // ... questo non viene eseguito in caso di eccezione
            successo = true;
        }
        catch(TipoExc1 a) {
            // ...
            // eventualmente comunica all'utente il fallimento per causa 1
        }
        // ...
```

```
        catch(TipoExc n) {
            // ...
            // eventualmente comunica all'utente il fallimento per causa n
        }
    }
}

// ...funzioneChiamata(...)
{
    // se qualche precondizione fallisce
    throw ...
    // ...
    return ...
}
```

Stringhe in C++

La libreria standard C++ fornisce la classe `string` che offre un'interfaccia sicura e conveniente per la gestione delle stringhe:

```
#include <string>
string sSource("012345678");
```

Operazioni principali sulle stringhe includono ottenere la lunghezza con `sSource.length()`, verificare se la stringa è vuota con `sSource.empty()`, e accedere ai singoli caratteri con l'operatore `[]`:

```
cout << sSource[0];
sSource[3] = '2';
```

Assegnamento di Stringhe

Le stringhe supportano diverse forme di assegnamento:

```
string sString;
// Assign a string value
sString = string("One");
cout << sString << endl;

const string sTwo("Two");
sString.assign(sTwo);
```

```
cout << sString << endl;

// Assign a C-style string
sString = "Three";
cout << sString << endl;
sString.assign("Four");
cout << sString << endl;

// Assign a char
sString = '5';
cout << sString << endl;

// Chain assignment
string sOther;
sString = sOther = "Six";
cout << sString << " " << sOther << endl;
```

È anche possibile assegnare una sottostringa:

```
const string sSource("abcdefg");
string sDest;
sDest.assign(sSource, 2, 4);
// assign a substring of source from index 2 of length 4
cout << sDest << endl;
```

Concatenazione di Stringhe

Le stringhe possono essere concatenate usando l'operatore `+=` o il metodo `append`:

```
string sString("one");
sString += string(" two");
string sThree(" three");
sString.append(sThree);
cout << sString << endl;
```

Il metodo `append` supporta anche l'aggiunta di sottostringhe:

```
string sString("one ");
const string sTemp("twothreefour");
sString.append(sTemp, 3, 5);
```

```
// append substring of sTemp starting at index 3 of length 5
cout << sString << endl;
```

L'operatore `+` può essere usato per concatenazione:

```
string sString("one");
sString += " two";
sString.append(" three");
cout << sString << endl;
```

Inserimento in Stringhe

Il metodo `insert` permette di inserire contenuto in una posizione specifica:

```
string sString("aaaa");
cout << sString << endl;
sString.insert(2, string("bbbb"));
cout << sString << endl;
sString.insert(4, "cccc");
cout << sString << endl;
```

Output:

```
aaaa
aabbbbaa
aabbccccbbaa
```