Risposte prog1

1)Si discuta dei principali tipi di memoria in C++, indicandone i nomi e i loro utilizzi.

Si facciano degli esempi (in codice C++) dei diversi tipi di allocazione. Si faccia

uno schema di come ciascuna allocazione modifica la memoria

1)I tipi di memoria in C++ sono tre:

Allocazione statica

• Dichiarazione di variabili al di fuori da qualunque blocco.

• Segmento di memoria ospitante `e detto segmento DATA.

• Ciclo di vita / scope: inizia e termina con il programma stesso

Allocazione dinamica in C++

L’allocazione dinamica permette di usare un’area di memoria condivisa dall’intero programma, detta heap. Per allocare una variabile dinamicamente, va usata la keyword new. La new restituirà un puntatore alla locazione di memoria definita nello heap.

Va notato che una variabile creata in un blocco di codice sarà disponibile anche al di fuori di esso, a patto che si passi l’indirizzo di memoria mediante una variabile con una visibilità più ampia.

Da notare che non è sufficiente tenere un riferimento ad un puntatore per accedere a una variabile dichiarata in un blocco di codice. E’ invece necessario che la variabile venga creata con allocazione dinamica.

Dato che le variabili allocate in maniera dinamica sono persistenti, esse vanno eliminate esplicitamente mediante la parola chiave delete quando non più utili. Questa operazione viene detta anche deallocazione.

//array allocato dinamicamente:

1. Int\* a=new int[5]; //array dinamico di 5 elementi;
2. Int\* a=new int[5]{1, 2, 3, 4, 5}; //array dinamico di 5 elementi, inizializzato;
3. Delete[] a; //deallocazione array;

//array bidimenzionale allocato dinamicamente;

1. int \*\*d = new int\*[3]; //dichiarazione;
2. for (int i=0; i<3; i++) //inizializzazione;
3. d[i] = new int[3];

Allocazione dinamica in C

malloc() alloca dinamicamente un blocco di memoria:

• argomento `e dimensione in byte (importante l’uso di sizeof: portabilit`a!)

• restituisce un puntatore generico, ovvero di tipo void \*, per

questo bisogna operare un type casting al tipo desiderato.

La funzione free() libera la memoria precedentemente allocata, come delete in C++.

1 #i n c l u d e <c s t d l i b >

2 double ∗ a r r = ( double ∗) malloc( s i z e o f ( double ) ∗ 1 0 ) ;

3 // . .

4 free(arr) ; // d e a l l o c a z i o n e

Allocazione automatica

• la memoria viene allocata mediante una dichiarazione di una variabile locale ad una funzione.

• Scope/visibilit`a limitato al blocco di codice in cui `e stata dichiarata.

• Ciclo di vita del blocco allocato termina con la fine dell’esecuzione del blocco in cui viene

• Area di memoria usata `e denominata STACK.

La porzione di memoria riservata allo stoccaggio dei dati utili alla esecuzione delle istruzioni contenute nel corpo delle funzioni `e denominata stack (pila).

Lo stack `e una struttura in cui i dati vengono inseriti e prelevati in base al meccanismo LIFO (Last In - First Out).

Il singolo dato viene depositato (push) sempre sul top dello stack.

Si pu`o prelevare un dato alla volta (pop), solo dal top dello stack.

2) Si introducano i diversi tipi di dato in C++, discutendone il loro utilizzo (perché sono necessari? Quando è meglio preferirne uno a un altro?) e definendo come essi vengono rappresentati in memoria secondo lo standard IEEE.

2)I tipi di variabili sono tanti ma sono divisi in macro classi:

Tipi carattere: codificano caratteri quali c, ; o %;

Tipi numerici interi: rappresentano numeri interi quali 2 o −777, che possono essere sia con segno (e dunque positivi e negativi) che senza segno;

Tipi nomrici in virgola mobile: si tratta di numeri con la virgola, quali 2.2 o −7.8;

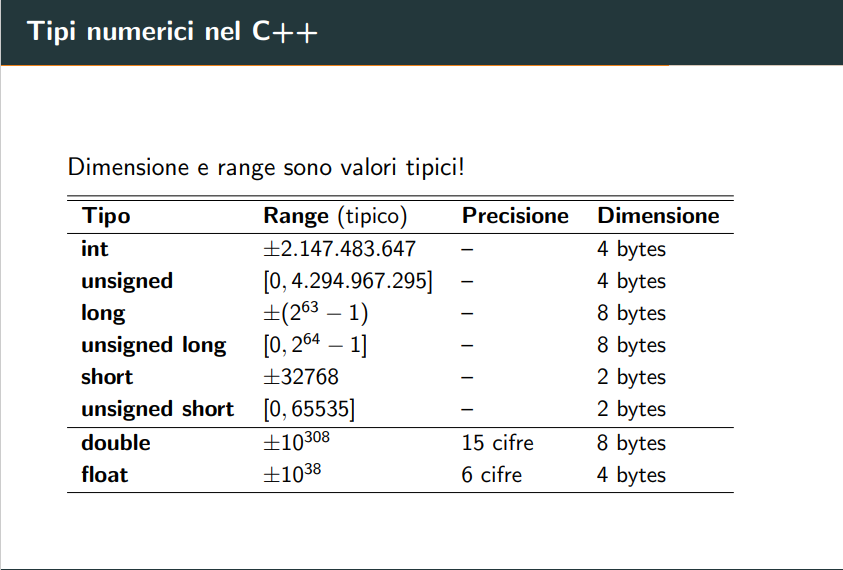
Tipo booleani: codifica solo due possibili valori, true e false, che indicano dei valori di verità associati a delle proposizioni (es. 5>7?);

Per rappresentare gli interi (con o senza segno), i bit si impiegano per rappresentare:

• il valore assoluto (o modulo) del numero stesso.

• Eventuale segno. In questo caso si “perde” un bit: il range di valori rappresentabili, in modulo, `e

dimezzato rispetto alla rappresentazione senza segno.

Immagine che contiene tavolo

Descrizione generata automaticamente

**-Numeri interi senza segno**

Codifica di Numeri interi senza segno con 16 bit:

Intervallo numerico: [0, 216 − 1] = [0, 65.535]

Codifica di numeri interi senza segno con 32 bit:

Intervallo numerico: [0, 2 32 -1] = [0, 4294967295]

**-Numeri interi con segno**

Codifica di numeri interi con segno a 16 bit:

Intervallo numerico ±(215 − 1) = 32767 (1 bit per il segno)

Codifica di numeri interi con segno a 32 bit:

Intervallo numerico ±(231−1) = 2.147.483.647 (1 bit per il segno)

**Numeri reali**

Nel calcolatore, i numeri reali sono rapprentati mediante un formato detto in virgola mobile (floating point).

Si tratta di una rappresentazione in forma compatta che deriva dalla rappresentazione scientifica.

Esempio in base 10

a) 96.103 = 0.96103 × 10+2

b) 2.96 = 0.296 × 10+1

c) 2.96 = 29.6 × 10−1

0.96103, 0.296 e 29.6 sono denominati Mantissa o significando.

10 `e la base.

+2 e +1 e -1 sono denominati esponente

Lo standard IEEE 754 definisce il formato per la

rappresentazione dei numeri in virgola mobile:

• 1 bit per la rappresentazione del segno (s);

• 8 o 11 bit per la rappresentazione dello esponente (E);

• 23 o 52 bit per la rappresentazione del significando o mantissa (M);

Tali che:

N = (−1) S× 2 E × M

**Errori di approssimazione quando**

• il numero non `e rappresentabile con numero finito di

cifre.

• il numero di cifre significative del numero da

rappresentare `e maggiore della precisione p

• Underflow quando il numero da rappresentare, in valore

assoluto, `e troppo piccolo per essere rappresentato.

• Overflow: il valor assoluto del numero `e maggiore, del

numero piu grande (in valore assoluto) rappresentabile con

quella codifica.

3) Si introducano le variabili in C++ e se ne discuta l’utilizzo (a cosa servono?). Si

discuta cosa si intende per visibilità di una variabile facendo degli esempi in

codice C++. Si discuta dei diversi modi di definire una costante in C++ e qual è

l’utilità delle costanti. Si discuta come fare ad accedere a una variabile fuori dalla

porzione di codice in cui è inizializzata? (esempio: restituire un array)3) Cosa `e una variabile?

Un contenitore di dati identificato da un nome all’interno del programma. La variabile corrisponde ad un certo indirizzo nella memoria del calcolatore nel quale si esegue il programma. Una variabile `e associato un certo tipo, che deve essere adeguato a rappresentare l’informazione che si vuole memorizzare all’interno della variabile.

1. Int a=5; //variabile intera
2. Char b=’c’; //variabile char

Inizializziamo le variabili con espressioni che usano letterali e/o altre variabili. Un valore letterale `e un elemento del programma che rappresenta un valore. true, 1.0, 40 sono letterali di tipo, rispettivamente booleano, double, e intero. "acqua" `e un valore letterale di tipo stringa, mentre ’c’ `e un letterale di tipo char

Si vuole assegnare un nome ad uno o pi`u valori costanti.

La parola riservata const per una variabile permette di indicare al compilatore che

• il valore di tale variabile const non pu`o cambiare rispetto al suo valore iniziale.

• quella variabile const va inizializzata in fase di creazione

4) Gestione di input e output da/su standard input, standard output e file. Esempi in

codice C++. Gestione degli errori di input output. Esempi in C++ d codice

funzionante e non funzionante (con commento sul perché è in quali occasioni

non funziona - es. design time o compile time? Con quale input?))

4)cin standard input stream

cout standard output stream

cerr standard error (output) stream

clog standard logging (output) stream

**gestione errori**

cerr:

il modo più corretto di mostrare dei messaggi di errore, consiste nell’utilizzare l’oggetto cerr. In questo modo, il sistema potra distinguere tra l’output del programma e i messaggi di errore.

#include <iostream>

#include <cstdlib>

using namespace std;

int main () {

int x;

cout << "Introdurre un numero dispari: ";

cin >> x;

if(x%2==0) { //il numero non è dispari, errore!

//usiamo cerr per gli errori

cerr << "Errore, il numero inserito non e' dispari!";

return EXIT\_FAILURE;

} else {

//Questo non è un errore, usiamo cout

cout << "Numero dispari, esco regolarmente";

return EXIT\_SUCCESS;

}

}

Fail():

Il metodo fail() permette di identificare possibili errori di input output che possono verificarsi quando ad esempio un utente inserisce una stringa al posto di un valore numerico. Il metodo fail() viene richiamato sull’oggetto cin dopo aver letto da input. Esso restituirà true se c’è stato un errore di input (es. una stringa che non può essere convertita in int).

#include <iostream>

#include <cstdlib>

using namespace std;

int main () {

int x;

cout << "Introdurre un numero intero: ";

cin >> x;

if (cin.fail()){

cerr << "Errore!";

return EXIT\_FAILURE;

} else {

cout << "Il numero inserito e' " << x;

}

}

Clear() e ignore():

Il metodo clear() di cin permette di resettare i flag di errore di cin ed è utile quando si vuole fare un secondo tentativo di lettura. Dopo aver richiamato clear(), se vogliamo provare a leggere nuovamente da input stream, dobbiamo anche utilizzare l’istruzione cin.ignore (numeric\_limits<streamsize>::max() , '\n') che scarterà tutti i caratteri inseriti (fino al massimo numero di caratteri inseribili), fino al carattere di terminazione riga \n. Questa istruzione serve a svuotare il buffer di cin in modo che sia possibile inserire nuovi caratteri. Consideriamo il seguente esempio:

#include <iostream>

#include <limits>

using namespace std;

int main () {

int x;

cout << "Introdurre un numero intero: ";

bool done = false;

do {

cin >> x;

if (cin.fail()) {

cerr << "Errore! Inserire un numero intero: ";

cin.clear();

cin.ignore (numeric\_limits<streamsize>::max() , '\n') ;

}

else

done = true;

} while(!done);

cout << "Il numero inserito e' " << x;

}

Eof():

Esiste un carattere speciale chiamato eof (end-of-file), che indica la fine di un file, o più in generale di un flusso di dati (vedremo meglio i file più avanti). L’inserimento di questo carattere determina la fine del flusso di dati e imposta di i bit di fallimento di cin.

Il carattere di end-of-file può anche essere specificato dall’utente quando inserisce un input da terminale. In linux, questo si fa con la combinazione CTRL + D, mentre in windows con CTRL + Z. Possiamo controllare che in cin sia stato inserito il carattere di end of file con il metodo cin.eof(). Vediamo un esempio in cui chiediamo all’utente di inserire una serie di numeri e ci fermiamo solo quando l’utente inserisce il carattere di terminazione.

#include <iostream>

using namespace std;

int main(){

int somma=0;

int x;

bool finito = false;

while(!finito) {

cout << "Inserisci un numero (premi CTRL + Z per terminare): ";

cin >> x;

if (cin.eof()) {

finito = true;

} else {

somma += x;

}

}

cout << "La somma dei numeri inseriti e' " << somma;

}

5) Si introducano le basi della programmazione strutturata, indicando le principali

strutture di controllo in C++. Si facciano degli esempi.

5) Ramo if:

La parola chiave if viene utilizzata per eseguire una istruzione se e solo se una data condizione è verificata. La sintassi base del costrutto if è la seguente:

if (condizione)

istruzione;

esempio:

#include<iostream>

using namespace std;

int main(){

int x;

cout << "Indovina il numero: ";

cin >> x;

if (x == 5)

cout << "Complimenti, il numero inserito e\' corretto!";

}

Ramo else:

Quando utilizziamo il costrutto if è anche possibile specificare una istruzione da eseguire nel caso in cui la condizione sia falsa. Questa si inserisce mediante una apposita keywork else. La sintassi è la seguente:

if(condizione)

//codice da eseguire se la condizione è vera

else

//codice da eseguire se la condizione è falsa

Va notato che, quando viene valutato un if, verrà sempre eseguito o il blocco if o il relativo else, mai entrambi insieme! Le parti del costrutto if ed else, spesso vengono anche chiamati “rami” (ramo if e ramo else).

Else if:

I blocchi else possono essere combinati con dei nuovi if. Questo è utile quando serve fare dei controlli a cascata. La sintassi è la seguente:

if(condizione)

//codice

else if (condizione)

//codice

else

//codice

Ciclo while:

Il ciclo while è il più semplice tipo di ciclo. La sua sintassi è la seguente:

while(condizione)

istruzione;

esempio:

#include<iostream>

using namespace std;

int main(){

int x = 1;

while(x<=10)

cout << x++ << endl;

}

Ciascuna ripetizione del ciclo viene detta iterazione. Va notato che l’istruzione cout << x++ << endl; stampa x e incrementa x di una unità allo stesso tempo. Ciò permette di cambiare il valore di verità della condizione del while e quindi permette al programma di terminare. In generale, le istruzioni eseguite all’interno del while devono permettere di cambiare la condizione del while. Ad esempio, il seguente programma stamperà sempre 1 e non terminerà mai! (possiamo interrompere il programma con CTRL + C:

Ciclo do-while:

Il ciclo do-while è molto simile al ciclo while, con la differenza che la condizione viene valutata dopo aver eseguito il blocco, piuttosto che prima. La sintassi è la seguente:

do

istruzione;

while(condizione);

esempio

#include<iostream>

using namespace std;

int main(){

int x;

do {

cout << "Indovina il numero: " << endl;

cin >> x;

} while(x!=5)

}

Ciclo for

Il ciclo for è pensato esplicitamente per eseguire un blocco di istruzioni per un certo numero di volte. In maniera simile ai cicli while, i cicli for eseguono le istruzioni specificati finché una condizione non si verifica. A differenza dei cicli while, però, con i cicli for è possibile specificare una istruzione di inizializzazione e una di aggiornamento. Ciò rende il ciclo molto comodo per utilizzare variabili contatore. La sintassi è la seguente:

for (inizializzazione; condizione; aggiornamento)

istruzione;

Il ciclo for funziona come segue:

1- Viene eseguita l’inizializzazione. Questa istruzione generalmente dichiara una sola variabile e viene eseguita una volta sola all’inizio del ciclo; 2. Viene controllata la condizione. Se è vera, si procede entrando nel corpo del for, altrimenti si termina l’esecuzione del for; 3. Vengono eseguite le istruzioni del corpo del for; 4. Viene eseguita l’istruzione di aggiornamento; 5. Si ritorna al punto 2.

Vediamo un esempio in cui riscriviamo il ciclo while che stampa i numeri da 1 a 10:

#include<iostream>

using namespace std;

int main(){

for (int x = 1; x<=10; x++)

cout << x << endl;

}

Cicli annidati

Quando si comparano diverse strutture sequenziali, si possono utilizzare dei cicli for annidati (nested in inglese). Un ciclo annidato è semplicemente un ciclo definito all’interno del corpo di un altro ciclo:

for(inizializzazione1; condizione1; aggiornamento1) {

for(inizializzazione2; condizione2; aggiornamento2) {

//codice

}

}

Ciclo for basato su range:

I cicli for hanno una sintassi alternativa che può essere utilizzata quando vogliamo scorrere gli elementi di una struttura sequenziale, quale un array o una stringa di caratteri. Non abbiamo ancora parlato di array, quindi per adesso vedremo un esempio con le stringhe.

La sintassi di questo tipo di ciclo for è la seguente:

for (dichiarazione: range)

istruzione;

Vediamo un esempio di ciclo che stampa tutti i caratteri di una stringa seguiti da -:

std::string s = "Hello World!";

for(char c: s)

std::cout << c << " - ";

Salti¶

C++ mette a disposizione delle specifiche istruzioni di “salto” che permettono di alterare il flusso in cui vengono eseguite le istruzioni.

Break¶

L’istruzione break permette di interrompere l’esecuzione di un ciclo e uscire da esso. Tale istruzione va semplicemente inserita all’interno del ciclo nel punto in cui si vuole che si esca dal ciclo. Consideriamo il seguente esempio:

#include<iostream>

using namespace std;

int main(){

for (int i = 0; i<10; i++) {

cout << i << endl;

if (i>5) {

cout << "Interrompo il ciclo" << endl;

break;

}

}

}

Continue

L’istruzione continue permette di saltare l’iterazione corrente di un ciclo e proseguire con la successiva. Consideriamo il seguente esempio che somma i numeri x tra 0 e 100 solo se 2⋅x+1 è multiplo di 3:

#include<iostream>

using namespace std

int main(){

int sum = 0;

for (int x=0; x<=100; x++) {

if((2\*x+1)%3!=0)

continue;

sum += x;

}

cout << sum;

}

Goto

Questa istruzione permette di effettuare un salto ad una parte del codice in termini assoluti. E’ considerata una feature “di basso livello” del C++ e va usata con molta cautela. In generale, tutto ciò che è possibile implementare con un goto, può essere implementato (e andrebbe preferibilmente implementato) facendone a meno. Per utilizzare una istruzione goto è necessario prima definire una etichetta (label) che identifichi la parte di codice alla quale saltare. Si può dunque usare l’istruzione goto label per saltare alla parte del codice considerata. Vediamo un esempio di come è possibile usare una istruzione goto per ripetere un insieme di istruzioni per un certo numero di volte:

#include <iostream>

using namespace std;

int main () {

int x=1;

etichetta:

cout << x << endl;

x++;

if (x<=10) goto etichetta;

}

Switch

Il costrutto di tipo switch permette di definire una serie di istruzioni da eseguire nel caso in cui una espressione assuma determinati valori. La sintassi di uno switch e la seguente:

switch (espression) {

case valore1:

istruzioni\_1;

break;

case valore2:

istruzioni\_2;

break;

...

case valoren:

istruzioni\_n;

break;

default:

istruzioni\_default;

}

Il funzionamento dello switch è il seguente:

Viene valutata l’espressione;

Si valuta in cascata (dall’alto al basso) se l’espressione è uguale a uno dei valori specificati;

Quando viene incontrato il primo valore corrispondente, si eseguono le istruzioni incluse per quel caso. L’istruzione break, serve per evitare che vengano controllate anche le successive;

Se non è stata trovata nessuna corrispondenza, vengono eseguite le istruzioni in default.

Vediamo un esempio:

#include <iostream>

using namespace std;

int main () {

char sex;

cout << "Inserisci il sesso (M o F): ";

cin >> sex;

switch(sex) {

case 'M':

cout << "E' un maschio'!" << endl;

break;

case 'F':

cout << "E' una femmina'!" << endl;

break;

default:

cout << "Non lo so!" << endl;

}

}

6) Si introducano le funzioni in C++ discutendone l’utilizzo e i principali elementi. Si

discutano l’overloading delle funzioni e le funzioni template facendo degli esempi

di definizione e invocazione delle funzioni. Si discutano i meccanismi di

selezione delle diverse implementazioni di funzioni overloaded. Si discuta la

differenza tra una funzione un metodo.

6) Le funzioni in C++ permettono di strutturare i programmi in segmenti di codice che svolgono compiti specifici. Un C++, una funzione può essere vista come un insieme di istruzioni al quale viene dato un nome, che può essere richiamato da qualsiasi punto del programma.

La sintassi più comune per definire una funzione è la seguente:

tipo nome (tipo\_1 parametro\_1, tipo\_2 parametro\_2, ...) { istruzioni }

Dove:

tipo indica il tipo del valore restituito dalla funzione, quando essa viene chiamata;

nome è il nome che identifica la funzione (un po’ come gli identificatori che abbiamo visto nel caso delle variabili);

tipo\_i è il tipo del parametro i-esimo della funzione;

parameter\_i è il nome del parametro i-esimo della funzione;

{ istruzioni} è il “corpo” della funzione, che può includere più istruzioni e specifica cosa la funzione fa in effetti.

// somma due interi

int sum(int x, int y) {

// l'istruzione return permette di restituire il risultato

return x + y;

}

Funzioni void

In C++, non è obbligatorio che una funzione restituisca un valore. Tuttavia, la sintassi richiede che venga sempre dichiarato il tipo del valore in uscita. Quando una funzione non restituisce nessun valore in output, il tipo di uscita va definito come void (che significa “vuoto”). Vediamo un esempio:

void printHello(string name) {

cout << "Hello " << name << endl;

}

printHello("World");

Valori di default:

Quando definiamo una funzione, è anche possibile specificare dei valori di default per i parametri. Ciò si fa inserendo un = valore dopo la dichiarazione del parametro nella definizione della funzione. Vediamo un esempio:

int add(int x, int y=1) {

return x+y;

}

Come abbiamo visto prima, le funzioni vanno dichiarate prima di essere utilizzate. C++ offre però la possibilità di disaccoppiare la dichiarazione della funzione dalla sua implementazione. La dichiarazione è una istruzione che definisce il nome della funzione e i tipo di ingresso e uscita senza però specificare l’implementazione. Questa dichiarazione viene anche detta prototipo della funzione. Una volta dichiarata una funzione, è possibile piazzare la sua implementazione anche più in là nel codice. La dichiarazione del prototipo viene effettuata semplicemente rimuovendo le parentesi graffe e inserendo un punto e virgola dopo le parentesi tonde, come mostrato di seguito:

int sum(int x, int y);

Oveloading di funzioni e metodi:

C++ permette di effettuare l’overloading di funzioni e metodi, ovvero di definire diverse “versioni” di una funzione/metodo che hanno lo stesso nome e lo stesso tipo di ritorno, ma prendono in input numeri diversi di parametri con tipi diversi. Lo scopo è quello di semplificare l’invocazione dei metodi.

string concat(string s1, string s2) {

return s1+s2;

}

string concat(string s1, int x) {

stringstream ss;

ss << s1 << x;

return ss.str();

}

string concat(int x, string s1) {

stringstream ss;

ss << x << s1;

return ss.str();

}

cout << concat("hello", "world") << endl;

cout << concat("hello", 2) << endl;

cout << concat(2, "world") << endl;

Funzioni template:

In C++ è possibile definire funzioni e classi template che operano su più tipi.

Per dichiarare una funzione template, bisogna anteporre al prototipo la notazione template <class C1, class C2, ...> o template <typename T1, typename T2, ...> (typename e class sono intercambiabili in questo contesto). Nel prototipo e nel corpo della funzione, sarà dunque possibile riferirsi a Cx e Tx per ottenere il tipo passato a runtime. Vediamo un esempio con la funzione print:

template<typename T>

void print(T x[], int n) {

for (int i=0; i<n; i++) {

cout << x[i];

if(i!=n-1)

cout << " - ";

}

cout << endl;

}

Quando richiamiamo la funzione template, dobbiamo specificare il tipo esatto tra parentesi angolate:

#include<string>

int a[] = {1,2,3};

double b[] = {2.2, 3.3, 4.4};

string s[] = {"hello", "world"};

print<int>(a, 3);

print<double>(b, 3);

print<string>(s, 2);

E’ possibile effettuare overloading anche di funzioni template. Esempio:

template<typename T>

T\* sum(T a[], T b[], int n) { //prende in input due array di tipo t1 e t2 e un intero n

T \*out = new T(n); //l'array in output è di tipo t1

for(int i=0; i<n; i++) {

out[i] = a[i] + b[i];

}

return out;

}

Questo overloading ci permette di eseguire istruzioni del tipo:

int x[] = {1,2,3};

int y[] = {4,5,6};

int \*out = sum<int>(x, y, 3);

print<int>(out, 3);

C++ mi permette di dare lo stesso nome e di selezionare in automatico quale avviare sulla base dei tipi a runtime, di fatto, il compilatore sceglie in automatico quale delle due versioni della funzione richiamare.

Differenza tra funzione e metodo:  
la differenza principale è data dal fatto che un metodo opera nelle classi quindi è una funzione strettamente legata alle classi e di conseguenza funziona come una funzione(possiamo effettuare un overloading per esempio) .

7) Si introducano gli array in C++, discutendone l’utilità, dettagliando come sono

rappresentati in memoria e riportando esempi dei diversi modi in cui è possibile

istanziarli. Si discuta della relazione tra array e puntatori e si mostri come è

possibile indicizzare un array utilizzando l’aritmetica dei puntatori. Si riportino

degli esempi di istanziazione di un array bidimensionale in allocazione dinamica.

7) Un array è una serie di elementi dello stesso tipo posizionati in locazioni di memoria contigue che possono essere referenziate aggiungendo un indice all’identificatore della variabile array.

Un array si dichiara in maniera simile a una variabile normale, ma è necessario specificare il numero di elementi che verranno contenuti nell’array. La sintassi è la seguente:

tipo nome[elementi];

Ad esempio:

int foo[5];

Inizializzazione degli array

Va notato che, come nel caso delle altre variabili, anche gli array vanno inizializzati opportunamente. Si può inizializzare un array specificando i diversi valori tra parentesi graffe, separati da virgole:

#include<iostream>

//inizializzazione con 5 valori

int foo[5] = {1, 3, 5, 7, 9};

for (int x: foo) {

std::cout << "["<< x << "]";

}

Non è possibile inizializzare un array con più valori di quelli che l’array può contenere.

E’ tuttavia possibile specificare meno valori. I tal caso, i valori mancanti verranno inizializzati a zero

Quando la dichiarazione dell’array viene fatta contestualmente alla sua inizializzazione, è possibile non specificare il numero di elementi, e far sì che il compilatore deduca la dimensione automaticamente:

int foo[] = {1,2,3};

for (int x: foo) {

std::cout << "["<< x << "]";

}

[1][2][3]

Se l’array viene solo dichiarato, tuttavia, la dimensione va specificata obbligatoriamente. Ad esempio, la seguente istruzione genera un errore:

int foo[];

E’ possibile dichiarare e inizializzare un array usando l’inizializzazione uniforma, che deriva dai costrutti di programmazione a oggetti di C++. Questa si ottiene omettendo il simbolo di assegnamento (=):

int foo[] {1,2,3};

for (int x: foo) {

std::cout << "["<< x << "]";

}

Accedere agli elementi di un array¶

Abbiamo visto come accedere in maniera sequenziale agli elementi di un array mediante il ciclo for basato su range. E’ inoltre possibile accedere agli elementi di un array in maniera diretta, specificando l’indice dell’elemento al quale vogliamo accedere. Va ricordato che in C++ l’indicizzazione parte da zero, per cui la prima posizione avrà indice 0, la seconda indice 1, e l’ennesima avrà indice n-1. Vediamo degli esempi:

int x[] {1,2,3,4,5,6};

std::cout << x[1] << std::endl; //posizione 1 - seconda posizione

std::cout << x[5] << std::endl; //posizione 5 - sesta posizione

std::cout << x[0] << std::endl; //posizione 0 - prima posizione

L’accesso può avvenire anche in scrittura:

int x[] {1,2,3,4,5,6};

x[1] = -1; //imposta x[1] a -1

x[3] \*= 2; //moltiplica x[3] per 2

x[5] /= 2; //divide x[5] per 2

for (int xx: x)

std::cout << xx << std::endl;

Va notato che tutti gli elementi di un array devono avere lo stesso tipo, per cui non è possibile assegnare a una locazione di un array un elemento di tipo diverso. Ad esempio:

int x[] = {1,2,3};

x[0] = 2.2; //errore

Inoltre, non è possibile accedere a valori che vanno oltre la dimensione dell’array:

int x[3] = {1,2,3};

cout << x[3]; //errore, x[3] non esiste!

Array multidimensionali

Un array multidimensionale può essere vista come una serie di array annidati. Il caso più semplice è quello bidimensionale. Possiamo vedere un array bidimensionale come una matrice o come un array di array. Gli array multidimensionali si dichiarano in maniera simile agli array monodimensionali:

int jimmy[3][5]; //array bidimensionale 3 x 5

Possiamo inizializzare gli array bidimensionali in maniera simile a quanto visto per i monodimensionali:

int jimmy[3][5] = {}; //inizializziamo con degli zeri

L’accesso all’array avviene specificando questa volta due indici. Ad esempio:

std::cout << jimmy[1][3];

Il primo dei due indici, selezionerà la riga della matrice, mentre il secondo selezionerà la colonna. E’ possibile inizializzare un array multidimensionale per righe. Ad esempio:

int x[2][2] = {1,2,3,4};

Sequenze di caratteri¶

Abbiamo visto come le stringhe possano essere viste come delle strutture sequenziali che contengono dei caratteri. Tuttavia, proprio perché le stringhe sono sequenze di caratteri, possiamo rappresentarle in C++ anche come array di caratteri:

char x[20] = "Hello";

std::cout << x;

Come per gli altri array, possiamo definire un array di char senza specificare le dimensioni come:

char x[] = "Hello";

std::cout << x;

Hello

O alternativamente:

char x[] = {'H','e','l','l','o'};

std::cout << x;

Puntatori e array

Puntatori e array sono due concetti molto legati tra loro. Un array, infatti, si comporta in maniera molto simile a un puntatore al suo primo elemento. In pratica, un array si comporta come un puntatore costante al primo elemento.

int arr[5] = {1,2,3,4,5};

int \*ptra;

cout << ptra << endl;

cout << &arr[0] << endl;

cout << arr << endl << endl;

cout << \*ptra << endl;

cout << arr[0] << endl;

cout << \*arr << endl << endl;

cout << arr[0] <<endl;

\*ptra=25;

cout << arr[0];

Aritmetica dei puntatori

C/C++ permette di eseguire operazioni aritmetiche sui puntatori. Le uniche operazioni permesse sono somme e sottrazioni con numeri interi. In pratica, dato un puntatore a un determinato tipo di dato (es puntatore a float), incrementare di una unità un puntatore permette di far avanzare l’indirizzo di memoria puntato dal puntatore di un numero di byte pari alla dimensione del tipo di dato considerato.

Abbiamo visto che un array è un puntatore costante al primo elemento dell’array stesso. Mediante le regole di base dell’aritmetica dei puntatori è dunque possibile manipolare gli array:

int myarray[5] = {2,8,5,1,-1};

//equivalente a myarray[0] (2)

cout << \*myarray << endl;

// (2)

cout << myarray[0] << endl << endl;

//equivalente a myarray[2] (5)

cout << \*(myarray+2) << endl;

// (5)

cout << myarray[2] << endl << endl;

//equivalente a myarray[4] (-1)

cout << \*(myarray+4) << endl;

// (-1)

cout << myarray[4] << endl << endl;

2

2

5

5

-1

-1

Un array è un puntatore costante, per cui non è possibile modificarne il valore con operazioni del tipo:

myarray++;

Tuttavia, queste operazioni sono possibili su un puntatore sul quale possiamo copiare l’indirizzo di myarray:

Allocazioni array:

Array statico:  
int a[5];

Array dinamico:

int\* a=new int[5];

int\* a=new int[5]{1, 2, 3, 4, 5};

Array multidimenzionale statico:

int a[5][5];

Array multidimenzionale dinamico:

int \*\*a=new int\*[5];

for(int i=0; i<5; i++)

a[i]=new int[5];

Anche se può confondere un po’, è possibile definire un array bidimensionale in maniera ibrida (un po’ statica e un po’ dinamica). In pratica, possiamo dichiarare un array statico di puntatori a int e far puntare ciascun puntatore a int a un array dichiarato dinamicamente:

int\* sd[3];

for (int i=0; i<3; i++) {

sd[i] = new int[3];

for (int j=0; j<3; j++) {

sd[i][j]=i+j;

}

}

for (int i=0; i<3; i++) {

for (int j=0; j<3; j++) {

cout << "s[" << i << "][" << j << "]="<< s[i][j] << endl;

}

}

8) Si introducano i puntatori in C++, indicandone la sintassi e discutendone i

principali vantaggi rispetto alle variabili non puntatore. Si introducano le

reference e si discutano le similarità e differenze tra puntatori e reference. Si

facciano degli esempi in C++ indicando con dei diagrammi in che modo viene

modificata la memoria. Si discuta dei diversi usi della parola “const”

nell’istanziazione di una variabile puntatore.

8) Puntatori

Le variabili in C/C++ rappresentano delle locazioni di memoria alle quali è possibile accedere mediante un identificatore (il nome della variabile). In questo modo, per il programmatore non è necessario preoccuparsi di gestire esplicitamente gli indirizzi fisici che identificano le porzioni di memoria in cui i dati risiedono.

Per un programma C/C++, la memoria viene comunque gestita come una serie di locazioni che vengono assegnate in successione. Ad esempio, quando si definiscono due variabili intere in successione, ad esse vengono assegnati due indirizzi fisici consecutivi.

Operatore di referenziazione &

L’indirizzo di una variabile può essere ottenuto mediante l’operatore di referenziazione &:

#include <iostream>

using namespace std;

int a = 5;

int b = 18;

cout << &a << endl;

cout << &b << endl;

I simboli a e &a rappresentano due cose diverse:

a è una variabile contenente l’intero 5;

&a contiene l’indirizzo di a.

I due simboli hanno due tipi diversi. Nello specifico:

a è di tipo intero;

&a è di tipo puntatore a intero.

Variabile puntatore

Se tipo è un determinato tipo di dato, il tipo di dato puntatore a tipo si indica con tipo \*. E’ dunque possibile definire una variabile di tipo “puntatore a intero” come segue:

int \*ptr;

cout << ptr << endl;

Quando dichiariamo un puntatore senza assegnargli alcun valore, questo viene inizializzato a zero di default. L’indirizzo “0” non è un indirizzo valido e viene alternativamente indicato come NULL o nullptr. Quando un puntatore ha valore zero possiamo assumere che esso “non punta a nulla”.

Adesso ptr è una normale variabile di tipo “puntatore a intero” e dunque possiamo assegnare ad essa un qualsiasi valore del suo stesso tipo. Ad esempio:

// puntatore NULL

ptr=NULL;

cout << ptr << endl;

// stesso effetto: nullptr

ptr=nullptr;

cout << ptr << endl;

// stesso effetto: 0

ptr=0;

cout << ptr << endl;

// assegno a ptr l'indirizzo di a, che ottengo dereferenziando:

ptr=&a;

cout << ptr << endl;

Operatore di dereferenziazione \*

Un puntatore p è una variabile che contiene l’indirizzo di memoria di un’altra variabile x. Pertanto, in genere si dice che p punta a x. I puntatori possono essere utilizzati per accedere al contenuto della memoria che referenziano in maniera diretta. Questo può essere fatto mediante l’operatore di dereferenziazione \* come segue:

int var1 = 228; //variabile di tipo int

int \*p1; //variabile di tipo "puntatore a int"

p1 = &var1; //assegnamo a p1 l'indirizzo di var1

//stampiamo il valore contenuto nella locazione a cui punta p1

cout << \*p1 << endl;

In pratica, anteponendo l’operatore \* al nome della variabile puntatore è possibile accedere al suo valore sia in lettura che in scrittura.

Cercare di dereferenziare un puntatore nullptr restituirà un errore.

Pertanto, può essere utile controllare che un puntatore non sia nullptr prima di dereferenziarlo.

Abbiamo visto che una data variabile e un puntatore ad essa possono in genere essere utilizzati in maniera “intercambiabile”. E’ cioè possibile scrivere e leggere il contenuto della cella di memoria relativa utilizzando direttamente la variabile o dereferenziandone il puntatore.

In alcuni casi, tuttavia, può essere utile definire dei puntatori di “sola lettura”. Si tratta di puntatori attraverso i quali è possibile leggere il contenuto della cella di memoria relativa ma non scriverci sopra. Questo tipo di puntatore può essere utile quando abbiamo bisogno di passarlo come parametro di una funzione e vogliamo assicurarci che la funzione acceda ai dati in sola lettura.

Per dichiarare un puntatore di sola lettura, dobbiamo anteporre la parola chiave const al tipo del puntatore:

int var3 = 118;

const int\* ptr3 = &var3;

//questa istruzione genererebbe un errore

//\*ptr3=18;

//accesso in lettura consentito

cout << \*ptr3 << endl;

Puntatori costanti

La keyword const può essere utilizzata anche per definire dei puntatori costanti. Un puntatore costante si comporta in maniera simile a una variabile costante: contestualmente alla sua definizione, è possibile assegnare un solo valore (ovvero un solo indirizzo) che non è più possibile cambiare in futuro. E’ possibile dichiarare un puntatore costante inserendo const tra tipo\* e il nome della variabile, come segue:

int var = 12;

int\* const const\_ptr = &var;

Puntatori costanti di sola lettura

Un puntatore può essere sia costante che di sola lettura. Per ottenere ciò, bisogna usare la parola const due volte nella definizione della variabile puntatore. Ad esempio:

int x = 0;

const int\* const p = &x;

Puntatori a puntatori, puntatori a funzioni e puntatori a strutture¶

Abbiamo visto come i puntatori possano puntare a diversi tipi di dato, quali int e double. In pratica, un puntatore può anche puntare a un altro puntatore. Per definire un puntatore a puntatore, bisogna utilizzare due asterischi:

int i1 = 15; //variabile int

int \* pi1 = &i1; //pi1 è un puntatore a i1

int \*\* ppi1 = &pi1; //ppi1 è un puntatore a pi1, che a sua volta punta a i1

cout << i1 << endl; //valore di i1

cout << &i1 << endl; //indirizzo di i1

cout << pi1 << endl; //valore di pi1 -> indirizzo di i1

cout << &pi1 << endl; //indirizzo di pi1

cout << \*pi1 << endl; //deferenziazione di pi1 -> valore di i1

cout << ppi1 << endl; //valore di ppi1 -> indirizzo di pi1

cout << \*ppi1 << endl; //deferenziazione di ppi1 -> valore di pi1 -> indirizzo di i1

cout << \*\*ppi1 << endl; //doppia deferenziazione di ppi1 -> valore di i1

Riferimenti

C++ mette a disposizione le reference, che permettono di avere i pro dei puntatori (overhead limitato) cercando di risolverne alcuni contro (sintassi più verbosa e rischi derivati da nullptr).

Una reference può essere definita con la seguente sintassi <tipo>& nome\_reference. Ad esempio:

#include<iostream>

using namespace std;

class Classe {

int x;

public:

Classe() : x(0) {}

Classe(int \_x): x(\_x) {}

void setX(int \_x) {x=\_x;}

int getX() {return x;}

};

Classe o;

Classe &ref\_o = o;

Il simbolo ref\_o è adesso un alias di o. Esso può essere usato esattamente come o (sintassi standard degli oggetti):

ref\_o.setX(12);

cout << ref\_o.getX();

A differenza di un puntatore, che può essere inizializzato a nullptr, una reference deve essere inizializzata contestualmente alla sua creazione.

Differentemente dai puntatori, non si può far puntare una reference a un nuovo oggetto dello stesso tipo mediante un assegnamento.

Riferimenti di sola lettura¶

I riferimenti visti finora sono di tipo lvalue. Ciò indica che possono trovarsi nel lato sinistro (left value) di un assegnamento, e dunque possono essere utilizzati per modificare gli oggetti/i valori ai quali puntano. Come per i puntatori, può essere utile definire un riferimento “di sola lettura”, che permetta di accedere in lettura al valore puntato (o allo stato dell’oggetto puntato), ma che non ne permetta l’accesso in scrittura. Per definire un riferimento di sola lettura, dobbiamo usare la keyword const:

int x = 8;

const int &r = x;

cout << r << endl;

differenza tra referenze e puntatori:

Il differenza fondamentale tra un puntatore e un riferimento è quello un puntatore è una variabile che memorizza l'indirizzo della posizione di memoria

di un'altra variabile mentre a riferimento è una variabile che fa riferimento a un'altra variabile. I due meccanismi, il pointer e riferimento, hanno

sintassi e utilizzo diversi.

Definizione

Pointer: Un puntatore è l'indirizzo di memoria di un oggetto memorizzato nella memoria di calcolo.

Riferimento: Un riferimento è un identificativo alternativo o un alias per un oggetto.

Caratteristiche del puntatore e riferimento

Dichiarazione

Pointer: Un puntatore è dichiarato con l'operatore \*.

Riferimento: Un riferimento è dichiarato con l'operatore &.

dereferenziazione

Pointer: Una variabile del puntatore richiede che l'operatore \* sia dereferenziato.

Riferimento: Una variabile di riferimento non richiede che nessun operatore venga deferenziato.

L'inizializzazione

Pointer: I puntatori possono essere inizializzati su null. Tali variabili sono chiamate puntatori nulli.

Riferimento: I riferimenti non possono essere inizializzati su null. Non esiste un riferimento nullo. Un riferimento deve sempre fare riferimento a un oggetto.

riassegnazioni

Pointer: Una variabile puntatore può essere riassegnata per fare riferimento a oggetti diversi.

Riferimento: Una variabile di riferimento non può essere riassegnata. Si riferisce sempre all'oggetto con cui è stato inizializzato.

uso

Pointer: Le variabili del puntatore dovrebbero essere utilizzate quando esiste la possibilità di riferirsi a nulla o quando è necessario fare riferimento

a cose diverse in momenti diversi.

Riferimento: Le variabili di riferimento dovrebbero essere usate quando ci sarà sempre un oggetto a cui fare riferimento e quando non è richiesto di usare

quella variabile di riferimento per riferirsi a qualsiasi altra cosa che non sia quell'oggetto.

Applicazione di utilizzo

Pointer: Le variabili puntatore possono essere utilizzate per implementare algoritmi e strutture dati.

Riferimento: Le variabili di riferimento possono essere utilizzate nei parametri di funzione e i tipi restituiti definiscono interfacce utili.

Lingue usate:

Pointer: Linguaggi di programmazione come C, C ++, Pascal e molti strumenti di supporto per il linguaggio assembly.

Riferimento: Linguaggi di programmazione come C ++, Java, Python, Perl, PHP, Ruby, ecc. Riferimenti di supporto.

9) Si introducano le stringhe in C++. Si discutano i due tipi di stringhe disponibili e

le rispettive differenze. Si facciano degli esempi in C++ di creazione e

manipolazione delle stringhe.

9) Introduzione alle stringhe¶

Vediamo adesso una breve introduzione a un tipo di dato speciale: le stringhe. Si tratta di un tipo di dato “composto”, che permette di rappresentare sequenze di caratteri come parole o frasi. Le stringhe si comportano come le altre variabili, per cui è possibile dichiararle, inizializzarle e cambiarne il valore durante l’esecuzione. Per usare le stringhe, è necessario includere l’header string. L’oggetto string si trova in std. Vediamo degli esempi:

#include<string>

std::string hello = "Hello";

std::string world("world");

std::string space{" "};

std::string exclamation ="!";

std::cout << hello << space << world << exclamation;

Va notato che le stringhe vanno dichiarate usando i doppi apici, a differenza dei caratteri, che vanno dichiarati usando i singoli apici:

char c = 'c'; //corretto

char x = "hello"; //errato, "hello" è una stringa, non un carattere

char a = "c"; //errato, "c" rappresenta una stringa di un solo carattere

std::string s = 's'; //errato, 's' rappresenta un tipo carattere

C++ mette a disposizione alcuni strumenti per manipolare le stringhe. Dato che le stringhe sono degli oggetti, esse contengono dei metodi. Vedremo meglio cosa sono i metodi, per adesso ci basti sapere che possono essere richiamate sull’oggetto in questione (la stringa) mediante la notazione oggetto.metodo().

Per ottenere una sottostringa, possiamo utilizzare substr:

std::string s = "Hello World!";

std::cout << s.substr(0,5) << std::endl; //inizia da 0 ed estrae 5 caratteri

std::cout << s.substr(6,6) << std::endl; //inizia da 6 ed estrae 6 caratteri

std::cout << s.substr(6,8) << std::endl; //inizia da 6 ed estrae 8 caratteri (sono 6 al massimo!)

Possiamo calcolare la lunghezza di una stringa con length:

std::string s = "Hello World!";

std::cout << s.length();

Possiamo combinare length e substr per estrarre gli ultimi n caratteri di una stringa:

std::string s = "Hello World!";

std::cout << s.substr(s.length()-3, 3) << std::endl; //ultimi 3 caratteri

std::cout << s.substr(s.length()-5, 5) << std::endl; //ultimi 5 caratteri

Stringhe come array di caratteri

Abbiamo visto che una stringa può essere definita in C++ mediante l’oggetto string, ma anche come array di caratteri:

char s[] = "stringa";

L’esempio sopra, è equivalente a scrivere:

char s[] = {'s','t','r','i','n','g','a',0}

o

char s[] = {'s','t','r','i','n','g','a','\0'}

Dove 0 o \0 sono dei caratteri di terminazine che servono a indicare che la stringa è terminata. Nella prima notazione, il carattere di terminazione viene inserito automaticamente dal compilatore. La seguente definizione invece non va bene perché non include il carattere di terminazione e può dare problemi alle funzioni che permettono di manipolare le strighe:

char s[] = {'s','t','r','i','n','g','a'}

Un letterale stringa è un puntatore costante a caratteri, pertanto la seguente definizioen non è corretta e genera un warning di compilazione:

char \*s = "stringa";

La seguente definizione è invece corretta:

const char \*s = "stringa";

Come scorrere una stringa:

string s=”ciao”;

for(int i=0; i<s.length(); i++)

cout << s[i] << endl;

char c[]={‘c’, ‘i’, ‘a’, ‘o’, ‘/0’}

int i=0;

while(c[i] != ’/0’){

cout << c[i] << endl;

i++;

}

10) Si introducano le classi e gli oggetti in C++, discutendo la loro utilità e i vantaggi

del loro uso rispetto alla classica programmazione procedurale. SI discutano i

principi fondamentali della programmazione a oggetti. Si facciano esempi di

definizione di classi e istanziazione dei relativi oggetti in allocazione statica e

dinamica.

10) Introduzione alle classi¶

Le classi permettono di specificare delle strutture dati, ovvero dei tipi non primitivi che permettono essenzialmente di memorizzare dati ed effettuare operazioni su di essi.

Una classe fornisce una sorta di progetto che specifica la struttura di una specifica istanza di una struttura dati, noto come oggetto. Una classe può essere specificata come segue:

class ClassName {

access\_specifier\_1:

member1;

access\_specifier\_2:

member2;

...

}

Dove:

ClassName è un identificatore della classe. E’ pratica comune esprimerlo in camel case;

access\_specifier\_\* indica il modificatore di accesso degli elementi che seguono. L’access specifier può essere private, protected o public;

member\* indica un membro della classe. Può essere una variabile (attributo o campo) o una funzione (metodo). Gli attributi permettono di memorizzare dati, mentre i metodi permettono di “comunicare” con l’oggetto, interrogandone o cambiandone lo stato.

Esempio di classe:

#include<iostream>

using namespace std;

class Rectangle {

int width, height;

public:

void set\_values (int,int);

int area ();

};

STATO

• Insieme di attributi

• Un attributo in generale rappresenta una propriet`a definita

in modo astratto.

• Anche detto “campo” o “variabile” perch`e spesso

mappato su una variabile

COMPORTAMENTO

Modellato o descritto da un insieme di metodi anche detti

funzioni membro.

Principio di identit`a.

Ogni oggetto di un determinato tipo ha una propria

identit`a.

Di conseguenza esso `e distinguibile da tutti gli

oggetti dello stesso tipo (es: automobile).Principio di conservazione dello stato.

Durante l’esecuzione del programma gli oggetti

mantengono le informazioni al proprio interno per

un tempo di indefinito.

Le classi vengono progettate in una delle fasi finali della

realizzazione del software:

1. Fase di Analisi (Object Oriented analysis):

• Analisi del dominio della applicazione.

• Analisi dei requisiti.

2. Fase di progettazione (o modellazione o design) delle

classi (Object Oriented Design).

3. Fase di implementazione (Object Oriented

Programming). L’implementazione `e costituita dalla

codifica delle classi utilizzando un linguaggio di

programmazione ad oggetti (ES: C++).

Programmazione strutturata/procedurale: una

sequenza di invocazioni di funzioni...

VS

Programma ad oggetti: l’esecuzione di un

programma ad oggetti flusso di messaggi tra

oggetti, ovvero una sequenza di invocazione di

metodi.

Nello sviluppo software, usando la metodologia della

programmazione procedurale, l’interesse principale è

rivolto alla sequenza di operazioni da svolgere: si crea un

modello indicando le procedure da eseguire in maniera

sequenziale per arrivare alla soluzione.

Lo spostamento di attenzione dalle procedure agli oggetti ha

portato all’introduzione della programmazione ad oggetti.

Gli oggetti sono intesi come entità che hanno un loro stato e

che possono eseguire certe operazioni.

L’algoritmo perde importanza a vantaggio del concetto di

Sistema

Un algoritmo è un insieme di

istruzioni che a partire dai dati

di input permettono di

ottenere i risultati di output.

Un algoritmo deve essere

riproducibile, deve avere una

durata finita e non deve essere

ambiguo. Il modo di

programmare pone attenzione

sulla sequenza di esecuzione.Un sistema è una parte del

mondo che si sceglie di

considerare come un intero,

composto da componenti.

Ogni componente è

caratterizzata da proprietà

rilevanti, e da azioni che

creano interazioni tra le

proprietà e le altre

componenti

I linguaggi procedurali hanno dei limiti nel creare

componenti software riutilizzabili.

I programmi sono fatti da funzioni, che

rappresentano codice riutilizzabile, ma che spesso

fanno riferimento a headers e/o variabili globali

che devono essere importate insieme al codice

delle funzioni.

I linguaggi procedurali non si prestano bene alla

modellazione di concetti ad alti livelli di astrazione,

utili per rappresentare entità complesse che

interagiscono in un sistema reale.

In altre parole, i linguaggi procedurali separano le

strutture dati e gli algoritmi

11) Si introduca l’ereditarietà tra classi, discutendone i principali vantaggi. Si

facciano degli esempi in C++. Si definiscano metodi e classi astratte e si

discutano gli usi del modificatore “virtual”. Si parli di overriding di metodi.

11) Non sempre occorre partire dal nulla nel costruire una

classe, soprattutto se si dispone già di una classe che è

simile a quella che si vuole costruire. In questo caso si può

pensare di estendere la classe già esistente per adattarla

alle nostre necessità.

L’ereditarietà è lo strumento che permette di costruire

nuove classi utilizzando quelle già sviluppate.

Quando una classe viene creata in questo modo, riceve

tutti gli attributi ed i metodi della classe generatrice (li

eredita). La classe generata sarà quindi costituita da tutti gli

attributi e i metodi della classe generatrice più tutti quelli

nuovi che saranno definiti

La classe che è stata derivata prende il nome di

sottoclasse, mentre la classe generatrice si chiama

sopraclasse.

Queste relazioni individuano una gerarchia che si può

descrivere usando un grafo di gerarchia.

La nuova classe si differenzia dalla sopraclasse in due

modi:

• Per estensione: aggiungendo nuovi attributi e

metodi

• Per ridefinizione: modificando i metodi ereditati,

specificando una implementazione diversa di un

metodo (override, overload)

Mezzi di trasporto

Esistono due tipologie di eridarieta:

eridarieta singola e multipla.

Considerando una relazione di ereditarietà, le sottoclassi hanno la

possibilità di ridefinire i metodi ereditati (mantenendo lo stesso

nome) oppure lasciarli inalterati perché già soddisfacenti.

Il polimorfismo indica la possibilità per i metodi di assumere forme,

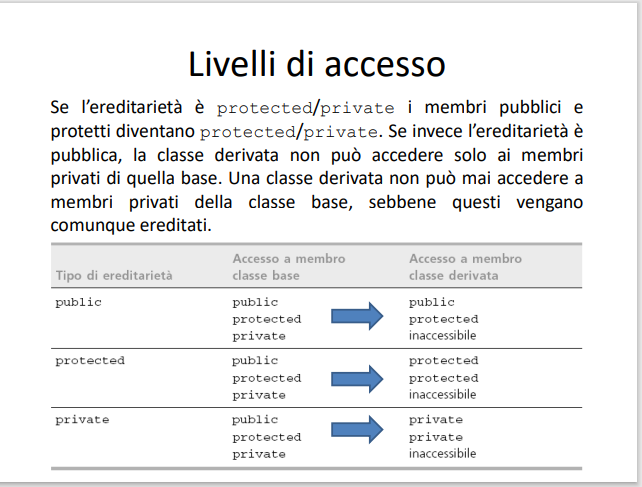
cioè implementazioni, diverse all’interno della gerarchia delle classi.

Esempio: tutti i veicoli a motore possiedono il metodo ‘’accelera’’.

Le sottoclassi ‘’automobile’’ e ‘’moto’’ è probabile che lo

ridefiniscano per adeguarlo alle particolari esigenze (es. pedale vs.

manopola)



Ereditarietà

L’ereditarietà permette di definire una classe che acquisisce specifici elementi di un’altra classe. Questa pratica mira a favorire il riutilizzo del codice. Vediamo un esempio:

#include <iostream>

using namespace std;

class A{

int x;

//B b;

public:

A(){

x = 10;

}

void f() { cout << "A.f(), x=" << x << endl;}

int getX() { return x;}

};

Definiamo adesso una semplice classe B che eredita da A:

class B: public A{

float y;

public:

B() {y=18;}

void g() { cout << "B.g(), y=" << y << endl;}

};

Possiamo ridefinire i metodi di A mediante overriding:

class B: public A{

float y;

public:

// ridefinendo una funzione con lo stesso nome

// facciamo overriding

void f() { cout << "override di A.f(), x=" << getX() << endl;}

void g() { cout << "B.g(), y=" << y << endl;}

};

Attenzione a non confondere overriding con overloading. Se effettuiamo overloading di un metodo, in automatico effettuiamo anche overriding.

Polimorfismo¶

In C++, un puntatore o una reference ad un oggetto di tipo A, può puntare ad oggi di tipo A e a qualsiasi sottotipo di A, ovvero tipi che ereditano da A con modificatore public. Vediamo un esempio:

A \*pa = new A(10);

B \*pb = new B(20, 4.4);

pb->g();

pa = pb;

pa->f();

Funzioni virtuali puri, classe astratte e interfacce¶

In certi casi, può avere senso non implementare alcuni metodi nella classe di base, in quanto il comportamente è definito bene solo dalle classi che ereditano. In questi casi, è possibile definire il metodo come “pure virtual”. Vediamo un esempio:

class Animale {

public:

virtual void parla() = 0;

};

class Cane: public Animale {

public:

void parla() {cout << "bau"<<endl;}

};

class Gatto: public Animale {

public:

void parla() {cout << "miao"<<endl;}

};

Animale \*a;

string animale = "gatto";

if(animale=="gatto")

a = new Gatto();

else if(animale=="cane")

a = new Cane();

a->parla();

In questo caso, la classe Animal è detta una classe astratta in quanto presenta almeno un metodo virtuale puro. Le classi astratte non possono essere instanziate, pertanto la seguente riga di codice darebbe un errore:

Animale a;

Una classe base astratta che contiene solo attributi costanti e funzioni virtuali pure si dice interfaccia. La classe Animale è una interfaccia. Un esempio di classe astratta ma non interfaccia è il seguente:

class Animale {

protected:

string name;

public:

virtual void parla() = 0;

string getName() {return name;}

};

Classi polimorfi: costruttori e distruttori virtuali¶

Un costruttore non può essere dichiarato come virtuale. Se vogliamo che la classe venga vista come polimorfa, il distruttore va invece dichiarato come virtuale. In questo modo, quando effettuiamo la delete di un puntatore di tipo A\* che punto a un oggetto di tipo B (con B che eredita da A con modificatore public), verrà correttamente richiamato il distruttore di B. Per far si che A sia polimorfa nell’esempio precedente, dobbiamo dunque dichiarare il distruttore come virtuale:

#include<iostream>

using namespace std;

class A{

int x;

//B b;

public:

A(int \_x){

x = \_x;

}

void f() { cout << "A.f(), x=" << x << endl;}

int getX() { return x;}

virtual ~A() {}

};

class B: public A{

float y;

public:

B(int x, float \_y) : A(x) {y=\_y;}

void f() {cout << "override di A.f(), x="<< getX() <<endl;}

void g() { cout << "B.g(), y=" << y << endl;}

};

12) Si introducano le classi template, evidenziandone i principali usi. Si facciano

degli esempi in C++ di definizione di classi template e istanziazione dei relativi

oggetti.

12) Classi template¶

In C++ `e possibile definire classi e funzioni template, che

operano su tipi generici.

I template consentono al programmatore di trattare i tipi dei dati

sui quali operazioni funzioni e classi come parametri.

I template esistono per consentire la riusabilit`a del codice rispetto

a tipi differenti (user-defined e primitivi):

• strutture dati (ES: pile, liste, code)

• algoritmi (ordinamento, ricerca, etc)

• ...

I template permettono di definire anche classi generiche. In questo caso, la specifica typename rende disponibile un tipo generico all’interno dell’intera classe:

template<typename T>

class Point2D {

T x, y;

public:

Point2D(T \_x, T \_y) : x(\_x), y(\_y) {}

T getX() {return x;}

T getY() {return y;}

void print() {cout << "("<<x<<","<<y<<")"<<endl;}

};

Una volta definita la classe generica, possiamo istanziare un oggetto della classe specificando i tipi:

Point2D<int> p\_int(2,3);

p\_int.print();

Vediamo un esempio in cui scorporiamo la dichiarazione della classe dall’implementazione dei metodi:

template<typename T>

class Point2D {

T x, y;

public:

Point2D(T \_x, T \_y);

T getX();

T getY();

void print();

};

In questo caso, i metodi vanno implementati come se fossero delle funzioni template. Inoltre l’oggetto va richiamato specificando i tipi definiti dal template:

template<typename T>

Point2D<T>::Point2D(T \_x, T \_y) : x(\_x), y(\_y) {}

template<typename T>

T Point2D<T>::getX() {return x;}

template<typename T>

T Point2D<T>::getY() {return y;}

template<typename T>

void Point2D<T>::print() {cout << "("<<x<<","<<y<<")"<<endl;}

Una classe generica può anche contenere funzioni membro generiche:

template<typename T>

class Point2D {

T x, y;

public:

Point2D(T \_x, T \_y);

T getX();

T getY();

template<typename T2>

void sum(T2 num);

void print();

};

Vediamo le implementazioni dei metodi in questo caso:

template<typename T>

Point2D<T>::Point2D(T \_x, T \_y) : x(\_x), y(\_y) {}

template<typename T>

T Point2D<T>::getX() {return x;}

template<typename T>

T Point2D<T>::getY() {return y;}

template<typename T>

template<typename T2>

void Point2D<T>::sum(T2 num) {x+=num; y+=num;}

template<typename T>

void Point2D<T>::print() {cout << "("<<x<<","<<y<<")"<<endl;}

I parametri delle classi template possono essere anche di uno dei tipi di dati primitivi. Esempio:

class Carta {

int valore;

public:

Carta(int v) : valore(v) {};

void print() {cout << valore << endl;}

};

template<typename C, int NUM=40>

class Mazzo {

C \*carte[NUM];

public:

Mazzo() {

for(int i=0; i<NUM; i++) {

carte[i] = new C(i);

}

}

C\* getCarta(int i) {return carte[i];}

};

13) Si discuta del meccanismo di overloading degli operatori, indicandone i vantaggi.

Si facciano deli esempi in C++ di overloading di operatori su classi.

13) Overloading degli operatori¶

L’overloading degli operatori permette di abilitare l’uso degli operatori disponibili in C++ per tipi diversi da quelli previsti di default. Un esempio classico di operatore overloaded è l’operatore + per la concatenazione di stringhe. Mentre l’operatore + è naturalmente definito tra numeri (es. 3+5), l’overloading dello stesso operatore tra stringhe permette di concatenarle.

Overloading con funzioni non membro¶

Vediamo un esempio di overloading degli operatori con funzioni non membro della classe. Supponiamo di avere una classe che implementa un punto nello spazio 2D:

class Point2D {

double x, y;

public:

Point2D(double \_x, double \_y) : x(\_x), y(\_y) {}

Point2D(): x(0), y(0) {}

double getX() {return x;}

double getY() {return y;}

};

#include<iostream>

using namespace std;

class Point2D {

double x, y;

public:

Point2D(double \_x, double \_y) : x(\_x), y(\_y) {}

Point2D(): x(0), y(0) {}

double getX() {return x;}

double getY() {return y;}

};

Point2D operator+(Point2D a, Point2D b){

return Point2D(a.getX()+b.getX(), a.getY()+b.getY());

}

int main() {

Point2D p1(2,3);

Point2D p2(3,3);

Point2D p3 = p1+p2;

cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;

}

Overloading con funzioni membro¶

In alternativa, è possibile effettuare l’overloading degli operatori mediante funzioni membro. In questo caso, dovremo passare in input solo l’operando di sinistra nel caso di operatori binari o nessun operando nel caso di operatori unari. Ad esempio:

#include<iostream>

using namespace std;

class Point2D {

double x, y;

public:

Point2D(double \_x, double \_y) : x(\_x), y(\_y) {}

Point2D(): x(0), y(0) {}

Point2D(double \_x) : x(\_x), y(\_x) {}

double getX() {return x;}

double getY() {return y;}

Point2D operator+(Point2D a){

return Point2D(a.x+x, a.y+y);

}

Point2D operator-() {

return Point2D(-x,-y);

}

};

int main() {

Point2D p1(2,3);

Point2D p2(1,2);

Point2D p3 = -(p1 + p2);

cout << "("<<p3.getX() << "," << p3.getY() << ")" << endl;

}

In questo caso, possiamo richiamare gli operatori anche come segue:

Point2D p3 = p1.operator+(p2);

Point2D p4 = p3.operator-();

Si noti che in questo caso, l’operazione che segue è permessa:

Point2D p3 = p1 +2; //promozione di 2 a Point2D permessa

Mentre la seguente non lo è:

Point2D p3 = 2 + p1; //promozione non permessa! l'operatore non è membro di "2"

Operatori speciali¶

Operatore <<¶

L’overloading dell’operatore << permette di “stampare” facilmente oggetti a schermo. Ricordiamo che la stampa avviene così:

cout << obj << obj <<obj << ...

Dove obj rappresenta un oggetto (es. una stringa). L’operatore << è dunque un operatore binario che ha due operandi:

Un ostream (ad esempio cout);

Un oggetto di un determinato tipo.

Per mantenere l’associatività, l’operatore restituisce un riferimento a ostream. Infatti, la catena sopra può essere letta come:

(cout << obj) << obj ...

Dove (cout << obj) restituisce un riferimento a cout. Se vogliamo abilitare l’operatore << nel nostro oggetto, dobbiamo implementare l’operatore come funzione friend (non possiamo implementarlo come membro perché il primo operando non è del tipo di obj). Vediamo un esempio:

#include<iostream>

using namespace std;

class Point2D {

double x, y;

public:

Point2D(double \_x, double \_y) : x(\_x), y(\_y) {}

double getX() {return x;}

double getY() {return y;}

friend ostream& operator<<(ostream &, Point2D);

};

ostream& operator<<(ostream &s, Point2D p) {

// inserisco una rapresentazione di p nel riferimento a ostream

s << "(" << p.x << "," << p.y << ")";

return s; //restituisco il riferimento

}

int main() {

Point2D p(2,3);

cout << p;

}

Operatore di cast¶

E’ possibile definire l’operatore di cast a un tipo specifico per regolare la conversione di tipo. Esempio:

#include<iostream>

using namespace std;

class IntPoint2D {

int x, y;

public:

IntPoint2D(int \_x, int \_y) : x(\_x), y(\_y) {}

int getX() {return x;}

int getY() {return y;}

friend ostream& operator<<(ostream &, IntPoint2D);

};

ostream& operator<<(ostream &s, IntPoint2D p) {

// inserisco una rapresentazione di p nel riferimento a ostream

s << "(" << p.x << "," << p.y << ")";

return s; //restituisco il riferimento

}

class Point2D {

double x, y;

public:

Point2D(double \_x, double \_y) : x(\_x), y(\_y) {}

double getX() {return x;}

double getY() {return y;}

operator IntPoint2D() {

return IntPoint2D(int(x), int(y));

}

friend ostream& operator<<(ostream &, Point2D);

};

ostream& operator<<(ostream &s, Point2D p) {

// inserisco una rapresentazione di p nel riferimento a ostream

s << "(" << p.x << "," << p.y << ")";

return s; //restituisco il riferimento

}

int main() {

Point2D p(2.2,3.3);

cout << p << endl;

cout << IntPoint2D(p);

}

14) Si parli della clausola namespace. A cosa serve e perché è utilizzata in C++? Si

facciano degli esempi, utilizzando anche l’operatore di selettore di scope ::

14) Namespace¶

I namespace in C++ permettono di definire dei “contenitori” per i simboli (nomi di variabili, funzioni, classi). Lo scopo dei namespace è quello di evitare il name clash, ovvero evitare che diverse librerie definiscano funzioni con nomi uguali. Finora abbiamo visto l’utilizzo del namespace std, ma altre librerie ne mettono a disposizione altri, ed è possibile definire il proprio namespace. Vediamo come:

namespace geometric {

class Rectangle {

double base;

double height;

public:

Rectangle(double b, double h) :base(b), height(h) {}

double area() {return base\*height;}

};

class Square {

double side;

public:

Square(double s) : side(s) {}

double area() {return side\*side;}

};

class Circle {

double r;

public:

Circle(double \_r): r(\_r) {}

double area() {return 3.14\*r\*r;}

};

}

Una volta definite le classi nel namespace, posso accedervi in diversi modi. Usando il selettore di scope:

#include<iostream>

using namespace std;

geometric::Square s(2);

cout << s.area();

4

Importando il simbolo Square:

using geometric::Square;

Square s(2);

cout << s.area();

4

oppure includendo using namespace geometric:

using namespace geometric;

Square s(2);

cout << s.area();

4

Includere tutto il namespace mi permetterà di accedere anche agli altri simboli dello stesso namespace:

Rectangle r(3,2);

cout << r.area();

6

15) Si introduca il concetto di polimorfismo, discutendo a cosa serve e come si

implementa. Si parli di Run-Time Type Identification (RTTI)

15) Polimorfismo:

Considerando una relazione di ereditarietà, le sottoclassi hanno la

possibilità di ridefinire i metodi ereditati (mantenendo lo stesso

nome) oppure lasciarli inalterati perché già soddisfacenti.

Il polimorfismo indica la possibilità per i metodi di assumere forme,

cioè implementazioni, diverse all’interno della gerarchia delle classi.

Esempio: tutti i veicoli a motore possiedono il metodo ‘’accelera’’.

Le sottoclassi ‘’automobile’’ e ‘’moto’’ è probabile che lo

ridefiniscano per adeguarlo alle particolari esigenze (es. pedale vs.

manopola)

Durante l’esecuzione del programma, un’istanza della

classe ‘’veicoli a motore’’ può rappresentare sia una

‘’automobile’’ che una ‘’moto’’.

Quando viene richiesta l’attivazione del metodo ‘’accelera’’

è importante garantire che, tra tutte le implementazioni,

venga scelta quella corretta.

Il collegamento dinamico è lo strumento utilizzato per la

realizzazione del polimorfismo. È dinamico perché

l’associazione tra l’oggetto e il metodo corretto da eseguire

è effettuata a run-time, cioè durante l’esecuzione del

programma.

RTTI: Run-Time Type Identification

Abbiamo visto che un puntatore di tipo A può puntare a un oggetto di tipo B se B eredita da A con modificatore public. Abbiamo però visto che se invochiamo un metodo contenuto anche in A mediante il puntatore a B di tipo A\*, verrà richiamata la versione di A a meno che il metodo non sia dichiarato virtual:

A \*p = new B(10, 4.4);

p->f();

A.f(), x=10

Per richiamare il metodo f di B, possiamo effettuare un cast dinamico:

B \*pp = dynamic\_cast<B\*>(p);

if(pp)

pp->f();

override di A.f(), x=10

Il dynamic\_cast è un cast sicuro che restituisce un nullptr se il cast non è possibile. Una alternativa non sicura derivata dal C, sarebbe quella di fare un cast standard:

((B\*)p)->f();

override di A.f(), x=10

O in alternativa (variante C++):

static\_cast<B\*>(p)->f();

override di A.f(), x=10

Questi cast non sono sicuri e possono creare problemi a runtime. Invece, va usato il cast dinamico.

Una soluzione alternativa al dinamyc\_cast consiste nel combinare l’operatore typeid che controlla i tipi a runtime con un cast statico:

if(typeid(\*p)==typeid(B))

static\_cast<B\*>(p)->f();

override di A.f(), x=10