**C++**

**Algoritmi e Strutture Dati A.A. 16/17**

Informatica Università degli Studi di Bari “Aldo Moro”

Nicola Di Mauro

**Un pò di storia**

●

La prima implentazione del C++ risale al 1980 (scritto da Bjarne Stroustrup)

– AT&T Bell Labs

●

Originariamente pre-compilato in C

– Segue un passo di compilazione C

●

Attualmente la precompilazione viene effettuata durante la fase di compilazione C++

●

Estensioni: .cc e .cpp

●

1980, ISO standard (stndard template)

●

2011, il nuovo ISO C++11

●

Linguaggio orientato ad oggetti

– Encapsulation

– Ereditarietà

– Modularità

– Polimorfismo – Operatori e sovraccarico di operatori

– Template

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 2

**Risorse**

●

Editor:

– GNU Emacs, Code

– Code::Blocks http://www.codeblocks.org/

●

Compilatore

– GCC, the GNU Compiler Collection (http://gcc.gnu.org/) – clang++ è un altro compilatore – Linux: apt-get install gcc – Windows: MinGW Minimalist GNU for Windows

●

Debugger

– GDB, the GNU Project Debugger (

http://www.gnu.org/software/gdb/)

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 3

**ADT Matrice in C e C++**

●

Specifica sintattica

– Tipi: matrice, intero, tipoelem

– Operatori:

●

CreaMatrice() → matrice ●

LeggiMatrice(matrice, intero, intero) → tipoelem ● ScriviMatrice(matrice, intero, intero, tipoelem) → matrice ●

In C possiamo operare come segue

typedef double tipoelem; typedef typoelem \*\* matrice; int righe = 10, colonne = 10;

tipoelem LeggiMatrice(matrice M, int r, int c){

return M[r,c]; }

void ScriviMatrice(matrice &M, int r, int c, tipoelem e){

M[r,c] = e; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 4

**ADT Matrice in C e C++ /2**

●

Requisito di astrazione

– La costruzione di una matrice nulla in C

nulla(matrice & M){

for (int i=0; i<righe; i++) for (int j=0; j<colonne; j++) M[i,j] = 0; } – L'implementazione della funzione

nulla

dipende dalla realizzazione

●

violato il requisito di astrazione – Invece, la funzione andrebbe scritta

nulla(matrice & M){

for (int i=0; i<righe; i++) for (int j=0; j<colonne; j++) ScriviMatrice[i,j] = 0; } – Inoltre, le variabili righe e colonne, proprie dell'astrazione matrice,

non sono protette ed incapsulate nella realizzazione

– possono essere modificate da chiunque a da qualunque posizione – cosa succede se un programmatore scrive righe++ ??? – cosa succede se si scrive colonne=-1 ???

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 5

**ADT Matrice in C e C++ /3**

●

Requisito di protezione

– In C possiamo violare facilmente il requisito di protezione – Supponiamo di avere le seguenti variabili

matrice A, B; double C[3][3]; double s; tipoelem e; – Non violano il requisito di protezione

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!!!

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 6

**ADT Matrice in C e C++ /4**

●

Come vedremo in seguito il C++ ci fornisce tutti gli strumenti per garantire il requisito di astrazione e di protezione delle nostre ADT

– ad esempio la nostra ADT matrice potrebbe essere realizzata come segue

●

la classe incapsula la nostra struttura dati

●

i metodi saranno 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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 7

**Compilazione**

●

Compilatore g++ della Free Software Foundation

●

Fasi per la creazione di un eseguibile

– Compilazione del sorgente (C, C++) in Assembly

●

di solito trasparente al programmatore – Assemblaggio del codice Assembly in codice oggetto

●

estensione .o – Linkaggio del codice oggetto

●

linkaggio di più file oggetto

●

risolve le dipendenza fra le librerie

●

produce l'eseguibile

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 8

**Compilazione /2**

●

Supponiamo di disporre di un programma progetto.cpp che include l'header della libreria lib.h la cui implementazione risiede nel file lib.cpp

– la direttiva #include “lib.h” dice al compilatore (in realtà al

preprocessore) di inserire il prototipo di una certa funzione x in progetto.cpp, la cui definizione risiede, però, in lib.cpp

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 9

**Passi di compilazione**

●

Passo 1: compilazione del programma principale

– g++ -c progetto.cpp

●

produce il file progetto.o

●

Passo 2: compilazione delle librerie

– g++ -c lib.c

●

produce il file lib.o

●

Passo 3: linkaggio dei file oggetto

– g++ progetto.o lib.o -o progetto

●

procude l'eseguibile progetto

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 10

**Compilazione separata**

●

In alcuni sistemi questi passi vengono effettuati in modo automatico

– Dev-C++

●

In programmi complessi è difficile tenere traccia dei singoli file modificati che richiedono una ricompilazione

●

utilizzo del comando make per l'esecuzione di un makefile che contiene tutte le istruzioni di compilazione e linking

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 11

**Makefile**

●

make è un sistema progettato per creare programmi costituiti da tantissimi file sorgente

– utilizza un makefile per ogni directory – un makefile contiene le istruzioni per costruire il programma

●

Componenti di un makefile

– target

– dipendenze

– istruzioni

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 12

**Makefile /2**

●

un makefile è costituito da coppie di linee

– governa l'aggiornamento di un file

●

da chi dipende l'eseguibile progetto?

– da progetto.o e lib.o

●

progetto non può essere creato senza progetto.o e lib.o

●

da chi dipende progetto.o?

– da progetto.cpp e lib.h

●

da chi dipende lib.o?

– da lib.cpp e lib.h

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 13

**Makefile /3**

●

un makefile contiene una coppia di linee per ogni file da costruire

●

nel nostro esempio avrà tre coppie di linee

– progetto

– progetto.o

– lib.o

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 14

**Makefile /4**

●

La prima linea di ogni coppia di linee in un makefile ha la seguente struttura

TargetFile: DependencyFile1 DependencyFile2 ... DependencyFilen

– TargetFile è il file che necessita l'aggiornamento – ogni DependencyFilei è un file da cui dipende TargetFile

●

La seconda linea è il comando (Unix) per creare TargetFile

**– il comando deve essere preceduto da un TAB e deve**

**terminare con un invio**

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 15

**Esempio**

progetto: progetto.o lib.o

g++ progetto.o lib.o -o progetto progetto.o: progetto.cpp lib.h

g++ -c progetto.cpp lib.o: lib.cpp lib.h g++ -c lib.cpp

●

Ora possiamo eseguire il comando make

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 16

**Esecuzione di make**

●

Nota che progetto dipende da progetto.o, e

– controlla progetto.o che dipende da progetto.cpp e lib.h – determina se progetto.o è out-of-date – se si, esegue il comando per creare project.o

g++ -c progetto.cpp

●

Nota che progetto dipende anche da lib.o, e

– controlla lib.o che dipende da lib.cpp e lib.h – determina se lib.o è out-of-date – se si, esegue il comando per creare lib.o

g++ -c lib.cpp;

●

Nota che tutto ciò da cui dipende project.exe è ora up-to-date, e quindi esegue il comando per creare progetto

g++ progetto.o lib.o -o progetto

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 17

**Dettagli**

●

Possono esserci più comandi dopo la linea delle dipendenze

progetto: progetto.o lib.o

g++ progetto.o lib.o -o progetto rm project.o rm lib.o

●

E' possibile invocare uno specifico target

ndm:~$ make lib.o

●

E' possibile inserire dei target speciali

clean:

rm -f progetto \*.o \*~ \*#

ndm:~$ make clean

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 18

**Le variabli**

●

E' possibile utilizzare delle variabili all'interno di un makefile

CC=g++ CFLAGS=-c -Wall

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 19

**Makefile con variabili**

# La variabile CC indica il compilatore da usare CC=g++ # Opzioni da passare al compilatore CFLAGS=-c -Wall

all: progetto

progetto: progetto.o lib.o

$(CC) progetto.o lib.o -o progetto

progetto.o: progetto.cpp lib.h

$(CC) $(CFLAGS) progetto.cpp

lib.o: lib.cpp lib.h

$(CC) $(CFLAGS) lib.cpp

clean:

rm -rf \*o hello

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 20

**Approfondimenti**

●

GNU Make

– http://www.gnu.org/software/make/

●

Dev-C++

– utilizza come compilatore il porting MinGW (Minimalist GNU for

Windows) di GCC (GNU Compiler Collection)

●

GCC: GNU Compiler Collection

– http://gcc.gnu.org/

– The Linux GCC HOWTO

●

http://www.pluto.it/files/ildp/HOWTO/GCC-HOWTO/GCC-HOWTO.html

●

MinGW (Minimalist GNU for Windows)

– http://www.mingw.org/

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 21

**Basics**

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 22

**Hello, world ...**

#include <iostream> int main() {

std::cout << "Hello world!" << endl; return 0; }

●

#include <iostream>

– direttiva al preprocessore

●

main() è una free function

– Il tipo restituito è un int, lo status code

●

0 successo, !=0 fallimento

●

std:: utilizzato per accedere ai nomi del namespace std

●

cout è un oggetto speciale che rappresenta lo screen

●

<< è un operatore (operatore di output)

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 23

**Parametri a riga di comando**

●

E' possibile accedere ai parametri dati a riga di comando definendo il main come segue

– int main (int argc, char\* argv[]) { ... }

●

Argc è il numero di parametri, incluso il nome del programma

●

Argv è un array di C-stringhe contenenti i parametri

●

Esempio

./myprog -a myfile.txt

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 24

**Commenti**

/\* 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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 25

**Variabili**

●

Una variabile è il nome di una porzione di memoria

– int x;

●

E' possibile stamparne il suo valore usando cout

– cout << x;

●

In C++ le variabili sono note come l-value (left side)

– Un valore che ha un indirizzo di memoria

●

Al contrario di un r-value (right side) che si riferisce ad un valore assegnato ad un l-value

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 26

**Variabili statiche**

●

Un attributo di una classe in C++ (come in Java) può essere statico

– Esiste una sola variabile per tutti gli oggetti della classe

●

In C++, anche le funzioni possono avere variabili statiche

– Viene creata alla prima chiamata della funzione e il suo valore

ricordato nelle successive chiamate di funzione

●

Contare il numero di chiamate della funzione

void f() {

static int counter = 0; counter++; ... }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 27

**Assegnamenti e r-value**

int y; // dichiara y come variabile integer y = 4; // 4 è uguale a 4, che viene asegnato 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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 28

**cin**

●

Cin è l'opposto di cout

#include <iostream>

int main() {

using namespace std; cout << "Enter a number: "; int x; cin >> x; cout << "You entered " << x << endl; return 0; }

– Cin >> x;

●

Legge il numero dalla console e lo assegna a x

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 29

**Funzioni**

// 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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 30

**Parametri di funzione**

#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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 31

**Forward declaration**

#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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 32

**Dati primitivi**

●

char

●

int, short, long

●

double, float

●

bool

– Nel vecchio C++ gli int erano usati per i booleani

int a; cin >> a; if (a) { ... } // equivalent to if (a != 0)

Point\* p = list.getFirstPoint(); while (p) { ... } // equivalent to while (p != 0) // 0 is C++98 for ’nullptr’

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 33

**Reference**

●

Un riferimento è un nome alternativo per identificare un oggetto, un alias

– Non come in Java che è un puntatore – Di solito usati come parametri di funzione

●

Un riferimento deve essere inizializzato quando creato, ed una volta inizializzato non può essere cambiato

– No n esistono riferimenti null

int ival = 1024; int& refval = ival; refval += 2; int ii = refval;

A che servono i riferimenti?

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 34

**Puntatori**

●

Un puntatore punta ad una locazione di memoria e conserva l'indirizzo di quella locazione

– Può far riferimento a qualsiasi cosa

●

Un puntatore viene dichiarato con un \* fra 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 (non il più comune come vedremo in seguito) per ottenere un valore puntatore è quello di far precedere il nome della variabile dal simbolo &, l'operatore indirizzo:

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

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 35

**Dereferenziare**

●

Il contenuto della memoria al quale punta un puntatore è accessibile con l'operatore \* (dereferenziazione)

int ival = 1024; int\* ip2 = &ival; cout << \*ip2 << endl; // prints 1024 \*ip2 = 1025;

●

Puntatori e riferimenti

– I puntatori possono essere dereferenziati, i riferimenti no – I puntatori possono esse indefiniti o null, i riferimenti no – I puntatori possono essere cambiati per puntare ad altro, i

riferimenti no

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 36

**Alias**

●

Un nome può essere definito come sinonimo di un nome di tipo già esistente

– Tradizionalmente si usa il typedef – Nel nuovo standard esiste una nuova dichiarazione di alias

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>;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 37

**auto (C++11)**

●

A volte i nomi di tipo sono lunghi da scrivere, oppure difficile per il programmatore ricordare il tipo esatto di una espressione

●

Il compilatore però non ha nessun problema nel dedurre un tipo

●

La parola chave auto dice al compilatore di dedurre il tipo da un inizializzatore

●

Esempi:

vector<int> v; // a vector is like a Java ArrayList ... auto it = v.begin(); // begin() returns vector<int>::iterator auto func = [](int x) { return x \* x; }; // a lambda function

●

Non usare auto quando il tipo è ovvio (ad es., per i letterali)

– auto sum = 0; // silly, sum is int

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 38

**auto rimuove & e const (C++11)**

●

Quando si usa auto per dichiarare variabili, i “top-level” & e const vengono rimossi

●

Esempi

int i = 0; int& r = i; auto a = r; // a is int, the value is 0 auto& b = r; // b is ref to int const int ci = 1; auto c = ci; // c is int, the value is 1 const auto d = ci; // d is const int, the value is 1

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 39

**decltype (C++11)**

●

A volte vogliamo definire una variabile con un tipo che il compilatore deduce da una espressione, ma non vogliamo assegnare il valore dell'espressione alla variabile

●

decltype restituisce il tipo del suo operando:

sometype f() { ... } decltype(f()) sum = 0; vector<Point> v; ... for (decltype(v.size()) i = 0; i != v.size(); ++i) {

... }

●

f() and v.size() are not evaluated.

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 40

**Array**

●

“The C array concept is broken and beyond repair.”

Bjarne Stroustrup

●

Programmi C++ moderni normalmente usano vettori invece degli array built-in

●

Non esistono verifiche sulla indicizzazione a runtime

– Con un indice errato è possibile accedere e distruggere un

elemento fuori dall'array

●

Esistono due modi di allocare array: sullo stack o sullo heap

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 41

**Array /2**

●

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); } ... }

●

In molti casi quando si usa il nome di un array, il compilatore sostituisce un puntatore al primo elemento dell'array

int\* px1 = x; int\* px2 = &x[0]; // equivalent

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 42

**Array /2**

●

E' possibile usare puntatori per accedere agli elementi di un array

– I puntatori sono gli iteratori per gli array

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 misura del tipo i dato indirizzato

– px + 1 significa px + sizeof(T) per un array di tipo T.

●

Si possono acre sottrarre due puntatori per ottenere il numero di elementi di un array fra due puntatori

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 43

**begin e end (C++11)**

●

Nell'esempio precedente abbiamo ottenuto il puntatore al primo elemento dell'array con il suo nome, e il puntatore a dopo l'ultimo elemento con il suo nome più il numero di elementi

●

Con un vector, avremmo usato i metodi begin e end

●

Poichè gli array sono un tipo built-in non hanno funzioni membro

– Possiamo usare le funzioni di libreria

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

cout << \*px << endl; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 44

**Array su heap**

●

Simili agli array in Java

void g(size\_t n) {

int a; int\* px = new int[n]; // size may be dynamic, >= 0 for (size\_t i = 0; i != n; ++i) {

px[i] = (i + 1) \* (i + 1); } ... delete[] px; // note [] }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 45

**Array: note**

●

Ad un heap-allocated array si accede con puntatori

●

Un heap-allocated array non contiene informazioni sulla sua lunghezza

●

Le funzioni iteratore begin() e end() non possono essere utilizzate per gli heap-allocated arrays

●

delete[] è necessaria per cancellare l'array

– Altrimenti gli oggetti nell'array non verranno distrutti

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 46

**Casting**

●

In C++ la convesione di tipo è implicita

d = 35.67; int x = d; // x == 35; implicit

●

Si dovrebbe usare un cast esplicito

– int x = static\_cast<int>(d);

●

Altri cast oltre quello statico sono

– dynamic\_cast<type>(pointer) per il “downcasting” in una gerarchia di

ereditarietà – const\_cast<type>(variable) rimuove la “constness” da una variabile

●

Non spesso usato – reinterpret\_cast<type>(expr) converte qualcosa in un'altra cosa

●

Reinterpreta il pattern di bit in modo diverso

●

Usato nella programmazine di basso livello

●

Il casting alla C (e alla Java), (int) d, è permesso in C++ ma non dovrebbe essere usato

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 47

**Passaggio di parametri**

●

I parametri ad una funzione sono passati per valore di default

– Il valore viene calcolato usando il parametro reale

●

Se si vuole cambiare il valore del parametro reale si può 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); }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 48

**Passaggio di parametri /2**

●

Il passaggio per riferimento viene usato anche per passare oggetti di grandi dimensioni senza farne una copia

bool isShorter(const string& s1, const string& s2) {

return s1.size() < s2.size(); }

●

La parola chiave const è essenziale se non si vuole modificare l'oggetto

– Senza const, non si potrebbe pasare una constant string – Senza const, si potrebbbe innavvertitamente cambiare lo

stato di un oggetto nella funzione

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 49

**Vector**

●

La classe vector è una classe template per la memorizzazione di elementi di tipo arbitrario

#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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 50

**Vector: note**

●

Un vector è inizialmente vuoto

– Gli elementi vengono aggiunti in coda con il metodo

push\_back

●

Lo spazio di per la memorizzazione di nuovi elementi viene allocato automaticamente

●

Si accede agli elementi con v[index]

●

I vector possono essere copiati

– V1 = V2

●

e confrontati

– V1 == V2

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 51

**Iteratori: intro**

●

vector è una classe container della libreria standard library. Di solito, si usano gli iteratori per accedere agli elementi di un container

●

Un iteratore “punta” ad uno degli elementi della collection (or ad una posizione immediatmente successiva all'ultimo elemento)

– si può dereferenziare con \* per accedere all'elemento al quale

punta – suò essere incrementato ++ per farlo puntare al successivo

elemento

●

I containers hanno i metodi begin() e end() che restituiscono un iteratore al primo elemento ed uno alla posizione dopo l'ultimo elemento del container

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 52

**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;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 53

**Aritmetica degli iteratori**

●

Gli itertori vector supportno alcune operazioni aritmetiche

●

Trovare il primo numero negativo nella seconda metà del vector auto it = v.begin() + v.size() / 2; // midpoint while (it != v.end() && \*it >= 0)

++it; if (it != v.end())

cout << "Found at index " << it - v.begin() << endl; else

cout << "Not found" << endl;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 54

**Iteratori const**

●

Un normale iteratore può essere utilizzato per leggere e scrivere elementi

●

Un const\_iterator può solo essere unsato per leggere elementi

– cbegin() e cend(), nuovi 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 un iteratore costante

void f(const vector<int>& v) {

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

\*it = 0; // Wrong -- ’it’ is a constant iterator }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 55

**Classi**

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 56

**Basics**

●

Una classe punto 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 ti tipo pubblico

– Vogliamo fare in modo che gli utenti possano usare quel nome

●

Le funzioni accessorie non cambiano lo stato dell'oggetto

– Dovrebbero essere dichiarate const

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 57

**Funzioni membro**

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; }

●

L'implementazione di funzioni membro può essere datat anche nella stessa definizione di classe

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 58

**Note**

●

this è un puntatore all'oggetto corrente

●

Una struct è come una classe ma tutte le funzioni membro sono pubbliche di default

●

Se due classi fanno riferimento l'una all'altra, è necessaria una dichiarazione di classe:

class B; // class declaration, "forward declaration" class A {

B\* pb; }; class B {

A\* pa; };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 59

**Esempio**

●

Fibonacci (1,1,2,3,5,8,....) 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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 60

**Esempio /2**

●

Vogliamo rendere fiibonacci più efficiente usando una cache per salvare i risultati precedenti. Aggiungiamo un vector<unsigned int> come funzione membro. Quando Fibonacci::value(n) viene invocato, tutti i numeri di Fibonacci fino a e incluso n saranno salvati nel vettore.

●

Il problema è che value è una funzione const, ma necessita di modificre una componente della classe

– Per rendere possibile questo possiamo dichiarare la componente mutable

class Fibonacci {

... 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()) return values[n]; for (unsigned int i = values.size(); i <= n; ++i)

values.push\_back(values[i - 1] + values[i – 2]); return values[n]; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 61

**Inizializzazioni**

●

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

class A { ... private:

int x = 123; // direct initialization, new in C++11

const int b; };

b = 10;

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 asseganti, devono essere inizializzati

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 62

**Delega ad altri costruttori (C++11)**

●

Un costruttore può delegare le inizilizzazioni ad altri costruttori

class Complex { public:

Complex(double r, double i) : re(r), im(i) {} Complex(double r) : Complex(r, 0) {} Complex() : Complex(0, 0) {} ... };

●

In questo esempio avremmo potuto usare anche i parametri di default:

Complex(double r = 0, double i = 0) : re(r), im(i) {}

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 63

**Membri statici**

●

Una classe che conta il numero di oggetti

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 64

**new e delete**

●

L'uso degli operatori new e delete viene tradotto nella chiamata alle seguenti funzioni che allocano/deallocno memoria

void\* operator new(size\_t bytes); void operator delete(void\* p) noexcept;

●

Esempio

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);

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 65

**Copia di oggetti**

●

Gli oggetti sono copiati in diversi casi

●

Inizializzazioni

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

●

Passaggio per valore

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

●

Assegnamento

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

●

Funzioni che restituiscono un valore

Person g() {

Person p5("Joe"); ... return p5; }

● Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 66

–

**Inizializzazione e assegnamento**

●

Le seguenti istruzioni sembrano simili ma vengono gestite diversamente in C++:

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

●

Inizializzazione:

– Un nuovo oggetto viene inizializzato con una copia di un altro oggetto.

●

Questo viene gestito dal costruttore di copia della classe Classname(const Classname&).

●

Assegnamento:

– Un oggetto esistente viene sovrascritto con una copia di un altro

oggetto

●

Gestito dall'operatore di assegnamento della classe Classname& operator=(const Classname&)

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 67

**Funzioni copia**

●

In ogni classe è possibile implementare il costruttore di copia e l'operatore di assegnamento in modo da avere il comportamento desiderato.

●

Quando non presenti il compilatore ne sintetizza uno che effettua la copia membro a membro

●

Non è necessario scrivere costruttori di copia e operatori di assegnamento se la classe non usa 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; };

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 68

**Una classe stringa**

●

Facciamo un esempio di una classe che alloca e dealloca memoria

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; };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 69

**Senza costruttore di copia**

●

Poiché non abbiamo definito un costruttore di copia il compilatore in caso di necessità effettuerà una copia membro a membro

void f() {

String s1("abc"); String s2 = s1; // see figure for memory state after copy }

●

s1.chars e s2.chars punteranno alla stessa area di memoria

●

Quando si uscirà dalla funzione

– Viene invocato il distruttore su s2 (s2.chars deleted) – Viene invocato il distruttore su s1 (s1.chars deleted) – Errore: non si può deallocare due volte la stessa area

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 70

**Senza costruttore di copia /2**

●

Esempio di passaggio per valore

void f(String s) {

... } void g() {

String s1("abc"); f(s1); }

●

Nella chiamata a f(s1), s1 è copiato in s. ora s.chars e s1.chars puntano allo stesso array

●

Quando si esce da f, viene invocato il distruttore per s, s.chars deleted. Ora però s1.chars punta alla memoria deallocata e s1 non può più essere utilizzato

●

Cosa accade quando si esce da g?

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 71

**Definire un costruttore di copia**

●

La classe String deve avere un costruttore di copia che effettua una copia più profonda

– Non copia i puntatori ma il contenuto dell'area di memoria alla

quale puntano

String(const String& rhs)

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

void f() {

String s1("abc"); String s2 = s1; }

●

Ovviamente ricordate che il costruttore di copia non viene invocato per oggetti passati per riferimento

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 72

**Assegnamento**

●

Problemi di copia si verificano anche quando si effettuano assegnamenti

void f() {

String s1("abc"); String s2("xy"); s2 = s1; }

●

s1.chars e s2.chars puntano alla stessa area di memoria

●

Qui la complicazione ulteriore è che abbiamo perso il riferimento al vettore xy che non potrà più essere referenziato

– Memory leak

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 73

**Sovraccaricare l'assegnamento**

●

Per risolvere il problema definiamo il nostro operatore di assegnamento

class String { public:

String& operator=(const String&); ... };

●

Con questo operatore l'istruzione

s1 = s2

●

viene convertita dal compilatore in

s1.operator=(s2)

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 74

**Sovraccaricare l'assegnamento /2**

●

Implementazione

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; }

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 75

**Sovraccaricare l'assegnamento /3**

●

Dettagli

– Cancellare il vecchio stato per evitare memory leak

– Restituire l'oggetto this – Verificare gli auto assegnamenti

●

if (&rhs == this)

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 76

**Spostare oggetti**

●

Il costruttore di copia fa una copia profonda

String(const String& rhs)

: chars(new char[strlen(rhs.chars) + 1]) {

strcpy(chars, rhs.chars); }

●

Se siamo certi che l'oggetto dal quale copiamo non verrà più usato dovremmo scrivere il costruttore di copia nel seguente modo, detto costruttore di spostamento:

String(String& rhs) : chars(rhs.chars) {

rhs.chars = nullptr; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 77

**Spostare oggetti /2**

●

Ma quando si è certi che l'oggetto dal quale copiamo non verrà più usato?

– La risposta è che i valori temporanei possono essere spostati

poiché verranno distrutti dopo l'uso

●

Il compilatore riconosce i valori temporanei

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 a temporary

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

●

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 78

**Spostare oggetti /3**

●

Un lvalue è persistente (variabili) mentre un rvalue non lo è

●

rvalue refenrence (C++11)

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

// temporary

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 79

**Spostare oggetti /4**

●

Ora che si ha un riferimento ad un rvalue si può scrivere il costruttore di spostamento

String(String&& rhs) noexcept : chars(rhs.chars) {

rhs.chars = nullptr; }

String(const String& rhs)

: chars(new char[strlen(rhs.chars) + 1]) {

strcpy(chars, rhs.chars); }

●

noexcept (C++11) dice al compilatore che il costruttore non lancierà nessuna eccezione

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 80

**Spostare oggetti /5**

●

Una classe che può essere spostata deve avere anche un operatore di assegnamento per movimento

String& operator=(String&& rhs) noexcept {

if (&rhs == this) {

return \*this; } delete[] chars; chars = rhs.chars; rhs.chars = nullptr; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 81

**Spostamento esplicito**

●

A volte il programmatore è certo che è safe spostare un oggetto piuttosto che copiarlo

– Si può usare la funzione standard std::move per ottenere un

riferimento ad un rvlue

template <typename T> inline void swap(T& a, T& b) {

T temp = std::move(a); // a is moved to temp, a is empty a = std::move(b); // but a isn’t used, instead b is

// moved to a, b is empty b = std::move(temp); // but b isn’t used, instead temp is

// moved to b, temp is empty } // but temp isn’t used, instead

// destroyed

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 82

**Idiomi di costruzione**

●

Quando una classe gestisce risorse dinamiche deve avere

– Un distruttore – Un costruttore di copia – Un operatore di assegnamento – Un costruttore di spostamento – Un operatore di assegnamento per movimento

●

Il costruttore deve inizializzare le componenti della classe

●

Il costruttore di copia deve fare una copia profonda delle componenti

●

L'operatore di assegnamento deve rilasciare le vecchie risorse allocate

●

Il distruttore rilascia tutte le risorse

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 83

**Vettori di oggetti**

●

Come abbiamo detto, si dovrebbero usare i vector invece che gli array

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; }

●

La classe Person deve avere un costruttore di copia e di movimento

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 84

**Vettori di puntatori ad oggetti**

●

Possiamo 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; }

●

La classe Person deve avere un costruttore di copia e di movimento

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 85

**Puntatori**

●

Un puntatore è una variabile che memorizza l'indirizzo di un'altra variabile

int \*pnPtr; // a pointer to an integer value double \*pdPtr; // a pointer to a double value int\* pnPtr2; // also valid syntax int \* pnPtr3; // also valid syntax

●

Possiamo assegnare a puntatori solo indirizzi di memoria

– Per ottenere indirizzi di memoria possiamo usare

l'operatore address-of (&)

int nValue = 5; int \*pnPtr = &nValue; // assign address of nValue to

// pnPtr

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 86

**Puntatori /2**

●

E' anche solito vedere istruzioni come le seguenti

int nValue = 5; int \*pnPtr = &nValue; //assign address of nValue to pnPtr cout << &nValue << endl; //print the address of variable nValue cout << pnPtr << endl;//print the address that pnPtr is holding

●

Il tipo di un puntatore deve essere uguale al tipo della variabile puntata

int nValue = 5; double dValue = 7.0; int \*pnPtr = &nValue; // ok double \*pdPtr = &dValue; // ok pnPtr = &dValue; // wrong -- int pointer cannot point to

// double value pdPtr = &nValue; // wrong -- double pointer can not point

// to int value

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 87

**Puntatori /3**

●

L'altro operatore usato con i puntatori è quello di dereferenziazione (\*)

– Un puntatore dereferenziato corrisponde al valore dell'indirizzo

al quale punta

int nValue = 5; cout << &nValue; // prints address of nValue cout << nValue; // prints contents of nValue int \*pnPtr = &nValue; // pnPtr points to nValue cout << pnPtr; // prints address held in pnPtr, which is &nValue cout << \*pnPtr; // prints contents pointed to by pnPtr, which is

// contents of nValue

●

Avremo in output

0012FF7C 5 0012FF7C 5

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 88

**Puntatori e array**

●

int anArray[5]; // declare array of 5 integers

– anArray è un puntatore al primo elemento dell'array

int anArray[5] = { 9, 7, 5, 3, 1 }; // dereferencing an array returns the first element (element 0) cout << \*anArray; // prints 9! char szName[] = "Jason"; // C-style string (also an array) cout << \*szName; // prints 'J'

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 89

**Puntatori e array /2**

const int nArraySize = 7; char szName[nArraySize] = "Mollie"; int nVowels = 0; for (char \*pnPtr = szName; pnPtr < szName + nArraySize; pnPtr++) {

switch (\*pnPtr) { case 'A': case 'a': case 'E': case 'e': case 'I': case 'i': case 'O': case 'o': case 'U': case 'u':

nVowels++; break; } } cout << szName << " has " << nVowels << " vowels" << endl;

Output: Mollie has 3 vowels

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 90

**Allocazione dinamica di memoria**

●

Allocazione di variabili

int \*pnValue = new int; // dynamically allocate an // integer \*pnValue = 7; // assign 7 to this integer

●

Quando allochiamo dinamicamente dobbiamo deallocare

delete pnValue; // unallocate memory assigned to pnValue pnValue = 0;

– L'operatore delete dealloca l'area puntata dal puntatore ma non il

puntatore stesso

●

Allocazione dinamica di vettori

int nSize = 12; int \*pnArray = new int[nSize]; // nSize does not need to be

// constant! pnArray[4] = 7; delete[] pnArray;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 91

**Memory leaks**

●

Memoria allocata dinamicamente non ha una visibilità (scope) effettiva

– Resta allocata fin quando non viene esplicitamente deallocata – Però i puntatori per accedere a tale memoria hanno regole di visibilità

void doSomething(){

int \*pnValue = new int; } – La precedente funzione alloca un intero dinamicamente ma non lo

dealloca mai – Quando la funzione termina pnValue non è più visibile (è una normale

variabile) – Poiché pnValue è l'unica variabile che conosceva l'indirizzo dell'intero

allocato dinamicamente, quando pnValue viene distrutta non ci sarà più nessun modo per far riferimento a quell'area di memoria allocata dinamicamente

●

**Questo è un memory leak**

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 92

**Memory leaks /2**

●

Un memory leak si verifica anche quando un puntatore ad una area di memoria allocata dinamicamente viene riasseganto

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

●

C'è memory leak anche con la doppia allocazione

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 93

**Passaggio di parametri**

●

I parametri vengono passati di default per valore

– Il parametro reale non può essere modificato dalla funzione

void foo(int y) {

using namespace std; cout << "y = " << y << endl; } int main(){

foo(5); // first call int x = 6; foo(x); // second call foo(x+1); // third call return 0; }

●

Vantaggi

– Si possono passare per valore variabili (x), letterali (6) o espressioni (x+1) – Si prevengono i side effects non potendo modificare il parametro reale

●

Svantaggi

– Copiare grandi strutture ha un costo computazionale

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 94

**Passaggio di parametri /2**

●

Quando si vuole modificare il parametro reale all'interno della procedura bisogna passare il parametro per riferimento

void AddOne(int &y) { // y is a reference variable

y = y + 1; } int main() {

int x = 5; cout << "x = " << x << endl; AddOne(x); cout << "x = " << x << endl; return 0; }

●

Output

x = 5; x = 6;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 95

**Passaggio di parametri /3**

●

Un modo per passare un parametro per valore, evitando di modificarlo nella funzione, ma senza farne una copia è usando il passaggio per const reference

void foo(const int &x){

x = 6; // x is a const reference and can not be changed! }

●

Regola: passare sempre per const reference a meno che non si necessiti di modificare il parametro reale

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 96

**Passaggio di parametri /4**

●

Un ulteriore modo per passare parametri è per indirizzo

– Il parametro formale deve essere un puntatore

void foo(int \*pValue){

\*pValue = 6; }

int main(){

int nValue = 5;

cout << "nValue = " << nValue << endl; foo(&nValue); cout << "nValue = " << nValue << endl; return 0; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 97

**Passaggio di parametri /5**

●

Differenza fra passaggio per riferimento e per indirizzo

– Il passaggio per riferimento ha una sintassi più chiara

●

I riferimenti sono più sicuri e più facili da usare – In termini di efficienza sono la stessa cosa – Quando di passa per indirizzo in realtà l'indirizzo è passato per

valore

●

Se si cambia l'indirizzo nella funzione in realtà stiamo cambiando la copia temporanea

– Il parametro reale non verrà cambiato

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 98

**Passaggio di parametri /6**

#include <iostream>

using namespace std;

int nFive = 5; int nSix = 6;

void SetToSix(int \*pTempPtr);

int main() {

// First we set pPtr to the address of nFive. Which means \*pPtr = 5 int \*pPtr = &nFive; cout << \*pPtr; // This will print 5 // Now we call SetToSix (see function below) // pTempPtr receives a copy of the address of pPtr SetToSix(pPtr); // pPtr is still set to the address of nFive! cout << \*pPtr; // This will print 5 return 0; }

// pTempPtr copies the value of pPtr! void SetToSix(int \*pTempPtr){

// This only changes pTempPtr, not pPtr! pTempPtr = &nSix; cout << \*pTempPtr; // This will print 6 }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 99

**return**

●

**Return by value**

int DoubleValue(int nX){ int nValue = nX \* 2; return nValue; // A copy of nValue will be returned here } // nValue goes out of scope here

●

**Return by reference**

– Attenzione non si possono restituire reference a variabili locali

int& DoubleValue(int nX){

int nValue = nX \* 2; return nValue; // return a reference to nValue here } // nValue goes out of scope here – Il compilatore darà un messaggio di errore

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 100

**return /2**

●

**Return by reference (cont.)**

– Di solito usata per restituire argomenti passati per reference

// This struct holds an array of 25 integers struct FixedArray25 {

int anValue[25]; };

// Returns a reference to the nIndex element of rArray int& Value(FixedArray25 &rArray, int nIndex) {

return rArray.anValue[nIndex]; }

int main(){

FixedArray25 sMyArray; // Set the 10th element of sMyArray to the value 5 Value(sMyArray, 10) = 5; cout << sMyArray.anValue[10] << endl; return 0; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 101

**return /3**

●

**Return by address**

– Non si possono restituire indirizzi di variabili locali

int\* AllocateArray(int nSize) {

return new int[nSize]; }

int main() {

int \*pnArray = AllocateArray(25); // do stuff with pnArray

delete[] pnArray; return 0; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 102

**Sovraccarico di funzioni**

●

Funzioni con lo stesso nome ma parametri differenti

– Sommare interi

int Add(int nX, int nY){

return nX + nY; } – E se volessimo sommare anche double

int AddI(int nX, int nY){

return nX + nY; }

double AddD(double dX, double dY){

return dX + dY; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 103

**Sovraccarico di funzioni /2**

●

Overloading

int Add(int nX, int nY){

return nX + nY; }

double Add(double nX, double nY){

return nX + nY; }

– Ma anche con un numero di parametri diverso

double Add(double nX, double nY, double nZ){

return nX + nY + nZ; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 104

**I parametri di default**

●

Un parametro di default è un parametro di una funzione che ha un valore di default

void PrintValues(int nValue1, int nValue2=10){

using namespace std; cout << "1st value: " << nValue1 << endl; cout << "2nd value: " << nValue2 << endl; } int main() {

PrintValues(1); // nValue2 will use default parameter of 10 PrintValues(3, 4); // override default value for nValue2 }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 105

**Puntatori a funzione**

●

I puntatori a funzione invece che puntare ad una variabile puntano ad una funzione

– int foo()

●

foo è un puntatore costante ad una funzione

●

Possiamo dichiarare anche puntatori non costanti a funzioni – int (\*pFoo) ();

●

pFoo è un puntatore ad una funzione senza parametri di input che restituisce un int

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 106

**Puntatori a funzione /2**

●

Selection sort in ordine ascendente

bool Ascending(int nX, int nY) {

return nY > nX; }

void SelectionSort(int \*anArray, int nSize) {

using namespace std; for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++) {

int nBestIndex = nStartIndex; // Search through every element starting at nStartIndex+1 for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize; nCurrentIndex++) {

// Note that we are using the user-defined comparison here

if (Ascending(anArray[nCurrentIndex], anArray[nBestIndex])) // COMPARISON DONE HERE

nBestIndex = nCurrentIndex; } // Swap our start element with our best element swap(anArray[nStartIndex], anArray[nBestIndex]); } }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 107

**Puntatori a funzione /3**

●

Possiamo pensare di avere in input anche il puntatore alla funzione che stabilisce il tipo di ordinamento

#include <algorithm> // for swap // Note our user-defined comparison is the third parameter void SelectionSort(int \*anArray, int nSize, bool (\*pComparison)(int, int)) {

using namespace std; for (int nStartIndex= 0; nStartIndex < nSize; nStartIndex++) {

int nBestIndex = nStartIndex; // Search through every element starting at nStartIndex+1 for (int nCurrentIndex = nStartIndex + 1; nCurrentIndex < nSize; nCurrentIndex++)

{

// Note that we are using the user-defined comparison here if (pComparison(anArray[nCurrentIndex], anArray[nBestIndex])) // COMPARISON DONE HERE

nBestIndex = nCurrentIndex; } // Swap our start element with our best element swap(anArray[nStartIndex], anArray[nBestIndex]); } }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 108

**Puntatori a funzione /4**

// Here is a comparison function that sorts in ascending order // (Note: it's exactly the same as the previous Ascending() function) bool Ascending(int nX, int nY){

return nY > nX; }

// Here is a comparison function that sorts in descending order bool Descending(int nX, int nY){

return nY < nX; }

// This function prints out the values in the array 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 }; // Sort the array in descending order using the Descending() function SelectionSort(anArray, 9, Descending); PrintArray(anArray, 9); // Sort the array in ascending order using the Ascending() function SelectionSort(anArray, 9, Ascending); PrintArray(anArray, 9); return 0; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 109

**Funzioni template**

●

Supponiamo di avere una funzione per il calcolo del massimo fra due numeri

int max(int nX, int nY){

return (nX > nY) ? nX : nY; }

●

Cosa accade se vogliamo calcolare il massimo fra due double?

– Dobbiamo sovraccaricare la funzione

double max(double nX, double nY){

return (nX > nY) ? nX : nY; } – L'implementazione è la stessa della precedente

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 110

**Funzioni template /2**

●

In c++ una funzione template è un pattern per la creazione di funzioni simili tra loro

– Definire una funzione senza specificare esattamente il

tipo delle variabili su cui opera – Usiamo invece dei segnaposto detti parametri template

Type max(Type tX, Type tY){ return (tX > tY) ? tX : tY; } – Dobbiamo però specificare in modo formale al compilatore

cosa intendiamo con il segnaposto Type – Dobbiamo utilizzare la dichiarazione di parametri template

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 111

**Funzioni template /3**

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

– La parola chiave template dice al compilatore che segue

una lista di parametri template – I parametri vengono racchiusi fra <> – Per definire un parametro template usiamo la parola

chiave typename o class

●

Dopo la parola chiave segue il segnaposto

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 112

**Funzioni template /4**

●

Ovviamente sul tipo del parametro template deve essere definito l'operatore <, ad esempio per la funzione max

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; } };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 113

**Classi template**

#include <assert.h> // for assert() class IntArray { private:

int m\_nLength; int \*m\_pnData; public:

IntArray() {

m\_nLength = 0; m\_pnData = 0; } IntArray(int nLength) {

m\_pnData = new int[nLength]; m\_nLength = nLength; } ~IntArray() { delete[] m\_pnData; } void Erase() {

delete[] m\_pnData; // We need to make sure we set m\_pnData to 0 here, otherwise it will // be left pointing at deallocated memory! m\_pnData = 0; m\_nLength = 0; } int& operator[](int nIndex) {

assert(nIndex >= 0 && nIndex < m\_nLength); return m\_pnData[nIndex]; } int GetLength() { return m\_nLength; } };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 114

**Classi template /2**

#include <assert.h> // for assert()

template <typename T> class Array{ private:

int m\_nLength; T \*m\_ptData; public:

Array() {

m\_nLength = 0; m\_ptData = 0; } Array(int nLength) {

m\_ptData= new T[nLength]; m\_nLength = nLength; } ~Array() { delete[] m\_ptData; } void Erase() {

delete[] m\_ptData; // We need to make sure we set m\_pnData to 0 here, otherwise it will // be left pointing at deallocated memory! m\_ptData= 0; m\_nLength = 0; } T& operator[](int nIndex) {

assert(nIndex >= 0 && nIndex < m\_nLength); return m\_ptData[nIndex]; } // The length of the array is always an integer // It does not depend on the data type of the array int GetLength(){return m\_nLength;}; // templated GetLength() function defined below };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 115

**Classi template /3**

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; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 116

**Classi template /4**

●

Expression parameter

– Un parametro di un template che non sostituisce un tipo ma un valore

template <typename T, int nSize> // nSize is the expression parameter class Buffer{ private:

// The expression parameter controls the size of the array T m\_atBuffer[nSize]; public:

T\* GetBuffer() { return m\_atBuffer; } T& operator[](int nIndex) { return m\_atBuffer[nIndex]; } };

int main(){

// declare an integer buffer with room for 12 chars Buffer<int, 12> cIntBuffer; // Fill it up in order, then print it backwards for (int nCount=0; nCount < 12; nCount++)

cIntBuffer[nCount] = nCount; for (int nCount=11; nCount >= 0; nCount--)

std::cout << cIntBuffer[nCount] << " "; std::cout << std::endl; // declare a char buffer with room for 31 chars Buffer<char, 31> cCharBuffer; // strcpy a string into the buffer and print it strcpy(cCharBuffer.GetBuffer(), "Hello there!"); std::cout << cCharBuffer.GetBuffer() << std::endl; return 0; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 117

**Classi template /5**

●

Specializzazione di template

– A volte è utile dare una implementazione di un metodo

specifica per un tipo

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; } };

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 118

**Classi template /5**

●

Se volessimo stampare in output i double in formato scientifico dobbiamo specializzare il metodo print come segue

void Storage<double>::Print() {

std::cout << std::scientific << m\_tValue << std::endl; }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 119

**Eccezioni**

●

In c++ è possibile gestire le eccezioni con i costrutti throw, try e catch

●

Una istruzione throw viene usata per segnalare che si è verificata una eccezione o un errore (alzare 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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 120

**Eccezioni /2**

●

Le eccezioni vanno poi catturate (catch) 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 // Look for exceptions that occur within try block and route to attached catch block(s)

{

// If the user entered a negative number, this is an error condition if (dX < 0.0)

throw "Can not take sqrt of negative number"; // throw exception of type char\* // Otherwise, print the answer cout << "The sqrt of " << dX << " is " << sqrt(dX) << endl; } catch (char\* strException) // catch exceptions of type char\* {

cerr << "Error: " << strException << endl; } }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 121

**Eccezioni /3**

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

// A modular square root function double MySqrt(double dX){

// If the user entered a negative number, this is an error condition if (dX < 0.0)

throw "Can not take sqrt of negative number"; // throw exception of type char\* return sqrt(dX); }

int main(){

cout << "Enter a number: "; double dX; cin >> dX; try // Look for exceptions that occur within try block and route to attached catch block(s) {

cout << "The sqrt of " << dX << " is " << MySqrt(dX) << endl; } catch (char\* strException) // catch exceptions of type char\* {

cerr << "Error: " << strException << endl; } }

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 122

**Stringhe in c++**

●

Definizione di stringhe #include <string>

– string sSource("012345678");

●

Lunghezza

– sSource.length()

●

Empty: stabilisce se la stringa è vuota

– sSource.empty()

●

Accesso ai singoli caratteri

– cout << sSource[0]

– Ssource[3] = 2;

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 123

**Stringhe in c++ /2**

●

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;

–

Algoritmi –

e Strutture Dati – A.A. 15/16, N. Di Mauro 124

**Stringhe in c++ /3**

●

Assegnamento (cont.)

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

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 125

**Stringhe in c++ /4**

●

Append

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;

●

Append con +

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 126

**Stringhe in c++ /5**

●

Inserimento

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

Algoritmi e Strutture Dati – A.A. 15/16, N. Di Mauro 127