

C++

Algoritmi e Strutture Dati

Corso di Laurea Triennale in Informatica
Università degli Studi di Bari “Aldo Moro”

Gianvito Pio

Credit

- Nicola Di Mauro
- Pierpaolo Basile

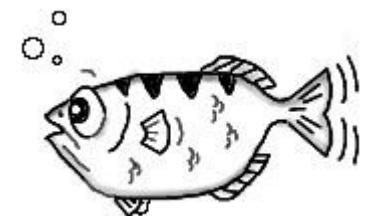
Un pò di storia

- La prima implementazione 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 (standard template)
- 2011, il nuovo ISO C++11
- Linguaggio orientato ad oggetti
 - Incapsulamento
 - Ereditarietà
 - Modularità
 - Polimorfismo
 - Operatori e sovraccarico di operatori
 - Template



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/>)



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 tipoelem ** 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;
}
```

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(M, i, j, 0);  
}
```

- Inoltre, le variabili righe e colonne, proprie dell'astrazione matrice, non sono protette ed encapsulate 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 ???

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!!!
```

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;  
}
```

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

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**

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
 - produce l'eseguibile **progetto**

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

Basics

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)

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
```

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
```

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 **I-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 I-value

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++;  
  
    ...  
}
```

Assegnamenti e r-value

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

cin

- **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

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(); // chiama a DoPrint()
    cout << "Ending main()" << endl;
    return 0;
}
```

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;
}
```

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;
}
```

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 'nullptr' for C++98
```

Reference

- Un riferimento è un nome alternativo per identificare un oggetto, un alias
 - Di solito usati come parametri di funzione
- Un riferimento deve essere inizializzato quando creato, ed una volta inizializzato non può essere cambiato
 - Non esistono riferimenti null

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

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);  
}
```

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 passare una constant string
 - Senza const, si potrebbe inavvertitamente cambiare lo stato di un oggetto nella funzione

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

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;
```

Dereferenziare

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

```
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 essere indefiniti o null, i riferimenti no
 - I puntatori possono essere cambiati per puntare ad altro, i riferimenti no

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>;
```

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 chiave 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

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

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
```

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 di dato indirizzato
 px + 1 significa px + sizeof(T) per un array di tipo T.
- Si possono anche sottrarre due puntatori per ottenere il numero di elementi di un array fra due puntatori

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`
- Poiché gli array sono un tipo built-in non hanno funzioni membro associate
 - Possiamo però usare le funzioni offerte dalla libreria

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

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 []
}
```

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

Casting

- In C++ la conversione 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 programmazione di basso livello
- Il casting alla C (e alla Java), `(int) d`, è permesso in C++ ma non dovrebbe essere usato

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;
}
```

Vector: note

- Un vector è inizialmente vuoto
 - Gli elementi vengono aggiunti in coda con il metodo `push_back`
- Lo spazio 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`

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 (o ad una posizione immediatamente successiva all'ultimo elemento)
 - si può dereferenziare con * per accedere all'elemento al quale punta
 - può 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

Scansionare un vettore

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

Meglio con auto (C++11)

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

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

```
for (int& e : v)  
    e = 0;
```

Aritmetica degli iteratori

- Gli iteratori vector supportano 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;
```

Iteratori const

- Un normale iteratore può essere utilizzato per leggere e scrivere elementi
- Un `const_iterator` può solo essere usato 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
}
```

Classi

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 di 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

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 data anche nella stessa definizione di classe

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;
};
```

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;  
}
```

Esempio /2

- Vogliamo rendere fibonacci 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 modificare 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];  
}
```

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 assegnati, devono essere inizializzati

Delega ad altri costruttori (C++11)

- Un costruttore può delegare le inizializzazioni 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) {}
```

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

new e delete

- L'uso degli operatori new e delete viene tradotto nella chiamata alle seguenti funzioni che allocano/deallocano 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);
```

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;
}
```

-

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&)

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;  
};
```

-

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;  
};
```

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;  
}
```

- 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**

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?

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

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 (e quindi neanche cancellato)
 - **Memory leak**

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)

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;  
}
```

-

Sovraccaricare l'assegnamento /3

- Dettagli
 - Cancellare il vecchio stato per evitare memory leak
 - Restituire l'oggetto this
 - Verificare gli auto assegnamenti
 - if ($\&rhs == this$)
-

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;
}
```

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 temporary
```

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

Spostare oggetti /3

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

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 lancerà nessuna eccezione

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;  
}
```

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 rvalue

```
template <typename T>
inline void swap(T& a, T& b) {
    // a is moved to temp, a is empty
    T temp = std::move(a);

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

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

    // temp isn't used, so it is destroyed
}
```

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

Idiomi di costruzione (riepilogo)

Costruttore di copia

Intestazione: `ClassName (const ClassName& rhs)`

Attivazione: creazione di un oggetto (che non esisteva) a partire da uno che esisteva già

`String a = b;` oppure

`String a(b);`

Operatore di assegnamento

Intestazione: `ClassName& operator=(const ClassName&)`

Attivazione: assegnazione di nuovo valore ad un oggetto che esisteva già, a partire da uno che esisteva già

`a = b;` // a e b esistevano già

Idiomi di costruzione (riepilogo)

Costruttore di spostamento

Intestazione: `ClassName (ClassName&& rhs) noexcept`

Attivazione: creazione di un oggetto (che non esisteva) a partire da un oggetto creato temporaneamente

`String a = b + " altro";` oppure

`String a(b + " altro");` // assumiamo che "+" concatene

Operatore di assegnamento per movimento

Intestazione: `ClassName& operator=(ClassName&& rhs) noexcept`

Attivazione: assegnazione di nuovo valore ad un oggetto che esisteva già, a partire da un oggetto creato temporaneamente

`a = b + " altro";` // a e b esistevano già

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

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

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**

Memory leaks /2

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

```
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
```

Passaggio di parametri per indirizzo

- 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;  
}
```

Passaggio di parametri per indirizzo

- 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 si 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

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
}
```

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**

- 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

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;
}
```

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;  
}
```

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;  
}
```

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;  
}
```

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`

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++){
            if (ascending(anArray[ncurrentIndex], anArray[nBestIndex]))
                nBestIndex = ncurrentIndex;
        }

        // Swap our start element with our best element
        swap(anArray[nstartIndex], anArray[nBestIndex]);
    }
}
```

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;

        for (int ncurrentIndex = nstartIndex + 1; ncurrentIndex < nSize;
ncurrentIndex++)
            {

                if (pComparison(anArray[ncurrentIndex], anArray[nBestIndex]))
                    nBestIndex = ncurrentIndex;
            }

        // Swap our start element with our best element
        swap(anArray[nstartIndex], anArray[nBestIndex]);
    }
}
```

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;
}
```

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

# 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

# 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'
```

# Funzioni template /4

- Ovviamente sul tipo del parametro template deve essere definito l'operatore <

```
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;
 }
};
```

# Funzioni template /4

- Ovviamente sul tipo del parametro template deve essere definito l'operatore <

```
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;
 }
};
```

**friend** è una funzione che  
può accedere ai membri  
protetti e privati della classe

# Class 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; }
};
```

# Class 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
};
```

# 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;
}
```

# Classi template /4

- Expression parameter
  - Un parametro di un template che non sostituisce un tipo ma un valore

```
template <typename T, int nSize
```

# 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;
 }
};
```

# 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;
}
```

# 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 un'eccezione o un errore (*lanciare 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
```

# 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 in the try block and route to 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;
 }
}
```

# 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;
 }
}
```

# 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;`

# 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 s0ther;
sString = s0ther = "Six";
cout << sString << " " << s0ther << endl;
```

# 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;
```

# 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;
```

# 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  
aabbcccccbaa

# Make

# 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

## 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

## 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

# Makefile /4

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

TargetFile: DependencyFile<sub>1</sub> DependencyFile<sub>2</sub> ... DependencyFile<sub>n</sub>

- TargetFile è il file che necessita l'aggiornamento
  - ogni DependencyFile<sub>i</sub> 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**

# 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*

# 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 progetto.exe è ora *up-to-date*, e quindi esegue il comando per creare progetto
  - `g++ progetto.o lib.o -o progetto`

# 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
```

# Le variabili

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

```
CC=g++
CFLAGS=-c -Wall
```

# 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
```

# 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/>