Programmazione C++

Riassunto di "The C++ Programming Language"
4a edizione
Bjarne Stroustrup

Jacopo De Angelis

27 maggio 2020

Indice

Programma esteso

- Introduzione al C++.
- Concetti base di programmazione C++
  - tipi di dati, puntatori, reference, scoping
  - casting,
- C++ come linguaggio ad oggetti
  - classi, costruttori e distruttori, overloading, metodi friend
  - inline, constness"
- Concetti avanzati di programmazione C++
  - overloading degli operatori
  - metodi virtual, abstract, polimorfismo
  - ereditarietà
- Programmazione generica
  - template
  - iteratori
- La libreria Standard (STL)
  - Le classi container
  - Gli algoritmi
  - Funtori
  - Multithread
- Uso delle librerie esterne
  - Librerie statiche
  - Librerie dinamiche
  - La libreria OpenMP
- I nuovi standard C++11, C++14
- Applicazioni GUI

- Ambiente di sviluppo QT Creator

- Sviluppo di interfacce grafiche

- Gestione degli eventi

- Le librerie Qt, QTWidgetscontenuto...

Capitolo 1

Principi base

1.1 Le basi

C++ è un linguaggio compilato. Il processo di compilazione è unione dei file è il seguente: Il programma è creato per uno speci-

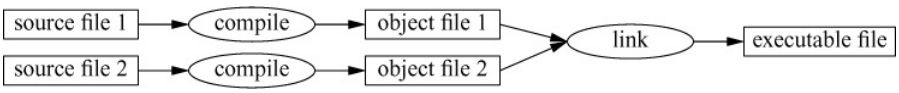

Figura 1.1: Processo di compilazione e unione

fico sistema operativo. C++ è source portable. Lo standard ISO C++ specifica due entità:

- Caratteristiche del linguaggio core: tipi built-in e loop
- Librerie standard: container e operazioni I/O

C++ è un linguaggio fortemente tipizzato e statico.

1.1.1 Hello, World!

```
// Hello, Comment!
```

```
#include <iostream>

int main(){
  std::cout << "Hello, World!\n";
}
```

Cose che possiamo notare:

- #include <iostream>: include la libreria iostream, ovvero la libreria di I/O
- «: prescrive in inserire il secondo argomento nel primo
- //: Commento su singola linea
- std::cout: standard output, fa parte del namespace della libreria standard

```
#include <iostream>
using namespace std; // Rende visibili i nomi di std senza
    ↪ doverlo scrivere

double square(double x){
    return x*x;
}

void print_square(double x){
     cout << "the square of " << x << " is " << square(x)
         ↪ << "\n";
}

int main(){
     print_square(1.234); //output: 1.234^2
}
```

Void indica che la funzione non ha valori di ritorno.

Data type	Size (bit)	Range
short int	16	-32,768 to 32,767
unsigned short int	16	0 to 65,535
unsigned int	32	0 to 4,294,967,295
int	32	-2,147,483,648 to 2,147,483,647
long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
long long int	64	-(2^63) to (2^63)-1
unsigned long long int	64	0 to 18,446,744,073,709,551,615
signed char	8	-128 to 127
unsigned char	8	0 to 255
float	32	
double	64	
long double	96	
wchar_t	16 o 32	1 wide character

1.1.2 Tipi, variabili e aritmetica

Una dichiarazione è una stringa di codice che introduce il nome del programma. É costituita da tipo e nome. I tipi primitivi di C++ sono:

- bool:
- char
- int
- double

In più int, char e double possono anche avere dei modificatori:

- long
- short
- signed
- unsigned

Le combinazioni sono: sizeof(var) permette di ottenere la

dimensione della variabile.

Aritmetica:

- x+y
- +x: +1
- x–y
- –x: -1
- x*y
- x/y
- x%y: modulo della divisione
-
- x+=y: x = x+y
- ++x: x = x+1
- x–=y: x = x-y
- —x: x = x-1
- x*=y: x = x*y
- x/=y: x = x/y
- x%=y: x = x%y

Comparazione:

- x==y: equal
- x!=y: not equal
- x<y: less than
- x>y: greater than
- x<=y: less than or equal

- x>=y: greater than or equal

C++ esegue automaticamente la conversione tra tipi nelle operazioni aritmetiche.

Code quality: mai inizializzare a vuoto se possibile. I tipi definiti dall'utente possono avere un'inizializzazione implicita.

1.1.3 Costanti

C++ supporta due tipi di costanti:

- const: la varibaile è una costante, non viene modificata. Viene usata generalmente per specificare le interfacce, in questo modo i dati possono essere passati alle funzioni senza che queste possano modificarli

- constexpr: la variabile verrà valutata durante la compilazione. Usata soprattutto per specificare le costanti, per permettere il posizionamento dei dati in memoria dove è difficile che vengano corrotti, e per performance

```
const int dmv = 17; // dmv is a named constant
int var = 17; // var is not a constant

constexpr double max1 = 1.4*square(dmv); // OK if
    ↪ square(17) is a constant expression
constexpr double max2 = 1.4*square(var); // error: var is not
    ↪ a constant expression
const double max3 = 1.4*square(var); // OK, may be
    ↪ evaluated at run time
double sum(const vector<double>&); // sum will not modify
    ↪ its argument (§2.2.5)
vector<double> v {1.2, 3.4, 4.5}; // v is not a constant
const double s1 = sum(v); // OK: evaluated at run time
constexpr double s2 = sum(v); // error: sum(v) not constant
    ↪ expression
```

Affinchè una costante venga valutata dal compilatore, essa deve essere definita come constexpr. Per essere una constexpr, la funzione deve essere molto semplice: deve avere solo il return. Le constexpr possono essere chiamate anche a runtime, in questo caso si comportano normalmente.

Le constexpr sono obbligatorie in certi casi che vedremo poi. In altri casi sono usate per performance. Nel caso un oggetto sia immutabile è necessario pensarci.

1.1.4 Puntatori, array e loop

Puntatori

Un array viene dichiarato come char v[6]; mentre un puntatore char* p. Gli array partono da 0. L'arra deve avere una lunghezza costante. Un puntatore può contenere l'indirizzo di un oggetto del tipo appropriato.

```
char* p = &v[3]; // p points to v's fourth element
char x = *p; // *p is the object that p points to
```

Possiamo vedere che * indica "contenuto di" mentre & indica "indirizzo di".

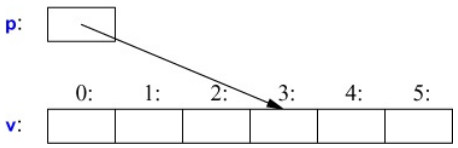

Figura 1.2: Puntatori e indirizzi

Loop

Ecco due implementazioni del ciclo for:

```
void copy_fct(){
```

```
    int v1[10] = {0,1,2,3,4,5,6,7,8,9};
    int v2[10]; // to become a copy of v1

    for (auto i=0; i!=10; ++i) // copy elements
        v2[i]=v1[i];
    // ...
}
```

```
void print(){
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto x : v) // for each x in v
        cout << x << '\n';

    for (auto x : {10,21,32,43,54,65})
        cout << x << '\n';
    // ...
}
```

Una delle prime cose che notiamo è auto: questa parola chiave si occupa di fare inferenza riguardo il contenuto della variabile, serve per non dichiarare esplicitamente che variabile primitiva. Può anche essere utilizzata come tipo di ritorno delle funzioni. Ottima per collegare ad una sola funzione o variabile temporanea più variabili con una notazione generica.

In questi due esempi copiamo il primo vettore nel secondo, ma se volessimo solo dire "incrementa ciò che c'è dentro al vettore ma senza creare una nuova variabile per farlo"? Dovremmo usare un ciclo che implementa dei puntatori e degli indirizzi:

```
void increment(){
    int v[] = {0,1,2,3,4,5,6,7,8,9};

    for (auto& x : v)
        ++x;
    // ...
}
```

In una dichiarazione il suffisso & significa "reference a". Una "reference" è simile ad un puntatore, eccetto che non ti serve il

prefisso * per accedere al valore puntato. In più una reference deve puntare sempre allo stesso oggetto dopo la sua inizializzazione.

Quando usati nelle di dichiarazioni, gli operatori &, * e [] sono chiamati "operatori di dichiarazione":

```
T a[n]; // T[n]: array of n Ts (§7.3)
T* p; // T*: pointer to T (§7.2)
T& r; // T&: reference to T (§7.7)
T f(A); // T(A): function taking an argument of type A
    ↪ returning a result of type T (§2.2.1)
```

Un altro tipo di loop è il while:

```
bool accept3(){
    int tries = 1;

    while (tries<4) {
        cout << "Do you want to proceed (y or n)?\n"; //
            ↪ write question
        char answer = 0;
        cin >> answer; // read answer

        switch (answer) {
        case 'y':
            return true;
        case 'n':
            return false;
        default:
            cout << "Sorry, I don't understand that.\n";
            ++tries; // increment
        }
    }

    cout << "I'll take that for a no.\n";
    return false;
}
```

1.1.5 Test di verità

I test di verità, oltre agli operatori ==, != ecc. abbiamo delle funzioni che si occupano di modificare il flusso del programma in base ai valori. Sono if, while e switch:

```
bool accept(){
	cout << "Do you want to proceed (y or n)?\n"; // write
	    ↪ question

	char answer = 0;
	cin >> answer; // read answer

	if (answer == 'y') return true;
	return false;
}
```

```
bool accept2(){
	cout << "Do you want to proceed (y or n)?\n"; // write
	    ↪ question

	char answer = 0;
	cin >> answer; // read answer
	switch (answer) {
	case 'y':
		return true;
	case 'n':
		return false;
	default:
		cout << "I'll take that for a no.\n";
		return false;
	}
}
```

1.1.6 Tipi definiti dall'utente

Strutture

Il primo modo per costruire un nuovo tipo è, spesso, il riorganizzare i dati in una struttura.

```
struct Vector {
      int sz; // number of elements
      double* elem; // pointer to elements
};

Vector v;
```

Qua possiamo vedere l'implementazione di un vettore tramite struct e la sua inizializzazione.

Per inizializzarlo dobbiamo però creare un'apposita funzione:

```
void vector_init(Vector& v, int s){
      v.elem = new double[s]; // allocate an array of s doubles
      v.sz = s;
}
```

In questo modo la funzione vector_init ottiene un puntatore alla variabile creata esternamente e il numero di elementi. Il fatto di usare Vector& serve in modo da ottenere una variabile non locale ma con effetti sulla variabile originale.

L'operatore new alloca memoria per la variabile. Un tipico uso di Vector è questo:

```
double read_and_sum(int s){
    // read s integers from cin and return their sum; s is
        ↪ assumed to be positive
    Vector v;
    vector_init(v,s); // allocate s elements for v
    for (int i=0; i!=s; ++i)
          cin>>v.elem[i]; // read into elements

    double sum = 0;
```

```
    for (int i=0; i!=s; ++i)
        sum+=v.elem[i]; // take the sum of the elements
    return sum;
}
```

Notiamo che questo vettore non è, ovviamente, il vector della libreria standard.

Classi

Qua veniamo a conoscenza della differenza tra interfaccia e implementazione di una classe. L'interfaccia è accessibile a tutti, l'implementazione è inaccessibile se non tramite appositi metodi.

La classe è definita da una serie di membri (funzioni, dati o strutture). Questi sono privati, accessibili solo tramite la loro interfaccia pubblica.

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { } // construct
        ↪ a Vector
    double& operator[](int i) { return elem[i]; } // element
        ↪ access: subscripting
    int size() { return sz; }
private:
    double* elem; // pointer to the elements
    int sz; // the number of elements
};
```

Una funzione con lo stesso nome della classe è chiamata "costruttore" e ha la funzione di creare la classe con i parametri passati.

Enumerazioni

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };
```

```
Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Le enumerazioni sono usate per rappresentare un riotto set di valori. Sono usate per rendere il codice più leggibile e meno prono ad errori. Notare che queste enumerazioni sono fortemente tipizzate, ad esempio:

```
Color x = red; // error: which red?
Color y = Traffic_light::red; // error: that red is not a Color
Color z = Color::red; // OK

int i = Color::red; // error: Color::red is not an int
Color c = 2; // error: 2 is not a Color
```

Se l'enumerazione non è basata su di una classe ma su di un tipo primitivo basta rimuovere "class".

1.1.7 Modularità

Compilazione separata

C++ supporta la compilazione separata dei vari file. Può essere usato per dividere il programma in blocchi di codice semi indipendenti. Questa separazione può essere usata per ridurre il tempo di compilazione richiesto.

Spesso inseriamo le dichiarazioni che specificano l'interfaccia di un modulo in un file denominato header (*.h). Ad esempio possiamo vedere qua prima l'header Vector.h e poi la sua implementazione:

```
// Vector.h:

class Vector {
public:
      Vector(int s);
      double& operator[](int i);
      int size();
```

```
private:
      double* elem; // elem points to an array of sz doubles
      int sz;
};

// Vector.cpp:

#include "Vector.h" // get the interface

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}{
}

double& Vector::operator[](int i){
    return elem[i];
}

int Vector::size(){
    return sz;
}
```

Per far sì che il compilatore assicuri la consistenza del codice, anche nella deifnizione viene incluso l'header.

Qua il suo uso:

```
#include "Vector.h" // get Vector's interface
#include <cmath> // get the the standard-library math
    ↪ function interface including sqrt()
using namespace std; // make std members visible (§2.4.2)
double sqrt_sum(Vector& v){
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
          sum+=sqrt(v[i]); // sum of square roots
    return sum;
}
```

I file possono essere compilati in maniera indipendente, lo schema è più o meno questo:

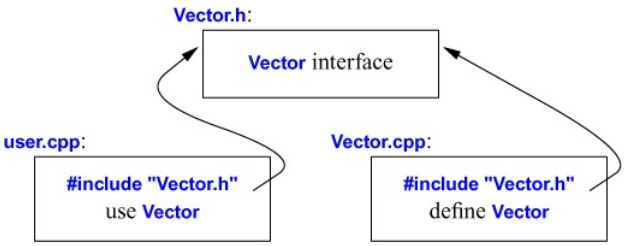

Figura 1.3: Uso dell'header

Namespace

C++ offre anche i namespace come meccanismo per esprimere alcune dichiarazioni che appartengono allo stesso gruppo senza che vadano in conflitto con altri. Per esempio, potremmo sperimentare con il nostro numero complesso e includerlo nel main in modo che non vada in conflitto con la versione originale:

```
namespace My_code {
    class complex { /* ... */ };
    complex sqrt(complex);
    // ...
    int main();
}

int My_code::main(){
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() <<
        ↪ "}\n";
    // ...
};

int main(){
    return My_code::main();
}
```

Un modo per accedere alle variabili di altri namespace è nome_del_namespace::funzione/variabile.Per avere accesso a tutti

i nomi della libreria standard possiamo usare la direttiva using, ad esempio using namespace std;

1.1.8 Gestione degli errori

Eccezioni

Lo schema è il classico try-throw-catch:

- Il codice che potrebbe generare un'eccezione viene messo all'interno di un blocco try
- L'eccezione viene lanciata con throw
- L'eccezione viene gestita da un catch

```
double& Vector::operator[](int i){
    if (i<0 || size()<=i) throw
        ↪ out_of_range{"Vector::operator[]"};
    return elem[i];
}

void f(Vector& v){
    // ...
    try { // exceptions here are handled by the handler
        ↪ defined below

        v[v.size()] = 7; // try to access beyond the end of v
    } catch (out_of_range) { // oops: out_of_range error
        // ... handle range error ...
    }
    // ...
}
```

Invarianti

Prendiamo un esempio classico: cercare di inizializzare un array con un numero negativo. Questa situazione può essere segnalata

con un'eccezione della libreria standard, length_error. In più se il compilatore non riesce ad allocare memoria darà come risposta un'eccezione bad_alloc. Quindi, il codice diventa:

```
Vector::Vector(int s){
    if (s<0) throw length_error{};
    elem = new double[s];
    sz = s;
}

void test(){
     try {
          Vector v–(27);
     }catch (std::length_error) {
          // handle negative size
     }catch (std::bad_alloc) {
          // handle memory exhaustion
     }
}
```

Asserzioni statiche

Le eccezioni riportano errori a runtime. Se un errore può essere trovato a tempo di compilazione allora è meglio individuarlo in quel momento. Per farlo possiamo usare le asserzioni statiche. L'importante è che vengano usate con un'espressione costante. Ad esempio:

```
constexpr double C = 299792.458; // km/s

void f(double speed){
     const double local_max = 160.0/(60*60); // 160 km/h
          ↪ == 160.0/(60*60) km/s

     static_assert(speed<C,"can't go that fast"); // error:
          ↪ speed must be a constant
     static_assert(local_max<C,"can't go that fast"); // OK

     // ...
}
```

Capitolo 2

Astrazione

2.1 Classi

Ci sono tre tipi di classi:

1. classi concrete
2. classi astratte
3. classi nella gerarchia

Classi concrete

L'idea è che si comporti esattamente come un tipo built-in. La caratteristiche che definisce un tipo concreto è che la rappresentazione fa parte della definizione.

Un tipo aritmetico Questo è un tipo tipo aritmetico definito dall'utente:

```
class complex {
      double re, im; // representation: two doubles
public:
```

```
    complex(double r, double i) :re{r}, im{i} {} // construct
        ↪ complex from two scalars
    complex(double r) :re{r}, im{0} {} // construct complex
        ↪ from one scalar
    complex() :re{0}, im{0} {} // default complex: {0,0}

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }

    complex& operator+=(complex z) { re+=z.re,
        ↪ im+=z.im; return *this; } // add to re and im
                                                                    ↪
                                                                    ↪
                                                                    ↪
                                                                    ↪
    complex& –operator=(complex z) { –re=z.re, –im=z.im;
        ↪ return *this; }

    complex& operator*=(complex); // defined out-of-class
        ↪ somewhere
    complex& operator/=(complex); // defined out-of-class
        ↪ somewhere
};
```

Container Un container è un oggetto che contiene una collezione di elementi, ad esempio un vettore. Pro di Vector: offre controllo di accesso sugli indici e offre una comoda funzione size. Difetto: alloca gli elementi usando new ma non li dealloca mai. C++ offre un'interfaccia per il garbage collector ma non viene usata automaticamente, quindi in certi casi potremmo trovarci a preferire un approccio più preciso. Questo meccanismo è un destructor:

```
class Vector {
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
public:
```

```
    Vector(int s) :elem{new double[s]}, sz{s} // constructor:
        ↪ acquire resources
    {
        for (int i=0; i!=s; ++i) elem[i]=0; // initialize
            ↪ elements
    }

    ~Vector() { delete[] elem; } // destructor: release resources

    double& operator[](int i);
    int size() const;
};
```

Il nome del destructor è l'operazione complementare seguita dal nome della classe. É il complemento del costruttore. Il costruttore alloca la memoria nell'heap usando l'operatore new. Il distruttore pulisce usando l'operatore delete.

La tecnica di acquisire le risorse in un costruttore e rilasciarle in un distruttore è nota come "Resource Acquisition Is Initialization" o RAII. Ciò permette di nascondere le operazioni new e delete, separando l'implementazione dall'astrazione.

Inizializzare i container Un container esiste per contenere elementi, dobbiamo quindi avere dei metodi comodi per inserirveli. Possiamo creare un vettore con il numero appropriato di elementi e poi aggiungendoli, ma solitamente ci sono modi più eleganti. Due modi possono essere:

- initalize
- push_back(): aggiunge un elemento in fondo alla lista

```
class Vector {
public:
    Vector(std::initializer_list<double>); // initialize with a
        ↪ list
    // ...
    void push_back(double); // add element at end
        ↪ increasing the size by one
```

```
    // ...
};

Vector read(istream& is)
{
    Vector v;
    for (double d; is>>d;) // read floating-point values into d
        v.push_back(d); // add d to v
    return v;
}

Vector v1 = {1,2,3,4,5}; // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8}; // v2 has 4 elements

Vector::Vector(std::initializer_list<double> lst) // initialize
    ↪ with a list
    :elem{new double[lst.size()]}, sz{lst.size()}
{
    copy(lst.begin(),lst.end(),elem); // copy from lst into elem
}
```

Il ciclo for è stato scelto per avere d solo nel suo scope e non occupare memoria dopo. La fine dello stream è decretata da un end-of-file (EOF) o da un errore di formattazione.

Tipi astratti

Un tipo astratto isola completamente l'utente dall'implementazione. Per farlo, separiamo l'interfaccia. Poichè non sappiamo niente sulla rappresentazione di un tipo astratto, dobbiamo allocare dello spazio e accedervi tramite puntatori. Prima definiamo una classe container che verrà creata come versione astratta del vettore.

```
class Container {
public:
    virtual double& operator[](int) = 0; // pure virtual
        ↪ function
    virtual int size() const = 0; // const member function
        ↪ (§3.2.1.1)
```

```
    virtual ~Container() {} // destructor (§3.2.1.2)
};
```

La classe è una pura interfaccia ad un container specifico definito più avanti. La parola ”virtual” significa ”potrebbe essere ridefinito più avanti in una classe derivata”.

Una classe derivata dalla classe Container offre un'implementazione dell'interfaccia. La curiosa sintassi =0 dice che la funzione è puramente virtuale, significa che obbligatoriamente deve essere implementata dalla classe derivata.

Non è quindi possibile definire un oggetto che è solo un Container, deve obbligatoriamente implementare operator[]() e size(). Una classe puramente virtuale è detta classe astratta.

```
void use(Container& c){
    const int sz = c.size();

    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Notare come use() utilizzi l'interfaccia Container senza avere alcuna conoscenza della sua implementazione. Utilizza size() e [] senza avere idea di che tipo offra la sua implementazione. Una classe che offre un'interfaccia a svariate altre classi è detta di ”tipo polimorfico”.

Come è comune per le classi astratte, Container non ha un costruttore, dopotutto non ha dati da inizializzare. Da un altro punto di vista, Container ha un distruttore e il distruttore è virtuale. Anche questo è normale in quanto solitamente passa un puntatore ad un Container senza conoscere a quale risorsa stia puntando.

Un container che implementa la funzione richiesta dall'interfaccia definita dalla classe astratta potrebbe essere la classe Vector:

```
class Vector_container : public Container { //
    ↪ Vector_container implements Container
```

```
    Vector v;
public:
    Vector_container(int s) : v(s) { } // Vector of s elements
    ~Vector_container() {}

    double& operator[](int i) { return v[i]; }
    int size() const { return v.size(); }
};
```

:public può essere letto come ”deriva da” o ”è un sottotipo di”. La classe Vector_Container deriva dalla classe Container e questa è detta classe base di Vector_container. Una terminologia alternativa è classe genitore e classe figlia. Questa proprietà è detta ereditarietà.

I membri operator[]() e size() fanno override dei membri corrispondenti nella classe base Container. Notare come il distruttore fa implicitamente riferimento a quello della classe base.

Per una funzione come use(Container&) usare un Container in completa ignoranza va bene, per altre funzioni dovremo creare un oggetto con il quale operare. Ad esempio:

```
void g(){
    Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    use(vc);
}
```

Poichè use() non è a conoscenza dei dettagli di Vector_container ma conosce solo l'interfaccia Container, funzionerà senza problemi con una diversa implementazione di Container. Ad esempio:

```
class List_container : public Container { // List_container
    ↪ implements Container
    std::list<double> ld; // (standard-library) list of doubles
        ↪ (§4.4.2)
public:
    List_container() { } // empty List
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}
```

```
        double& operator[](int i);
        int size() const { return ld.size(); }
};

double& List_container::operator[](int i)
{
        for (auto& x : ld) {
                if (i==0) return x;—
                i;
        }
        throw out_of_range("List container");
}
```

2.1.1 Funzioni virtuali

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Come viene risolte la chiamata a c[i] per operator[]()? Quando h() chiama use() viene chiamato operator[]() di List_container . Quando g() chiama use(), viene chiamato operator[]() di Vector_container. Per avere questa risoluzione, un oggetto Container deve contenere l'informazione necessaria per permettere di selezionare la giusta funzione a runtime.

La tecnica classica è che il compulatore converta il nome della funzione virtuale in un indice che verrà aggiunto alla tabella dei puntatori alle funzioni. Questa tabella è chiamata "virtual function table" o semplicemente vtbl. Ogni classe ha la sua vtbl che identifica le sue funzioni virtuali. Una rappresentazione grafica è questa:

Le funzioni nella vtbl permette agli oggetti di essere usati correttamente anche quando la grandezza dell'oggetto e il tipo di

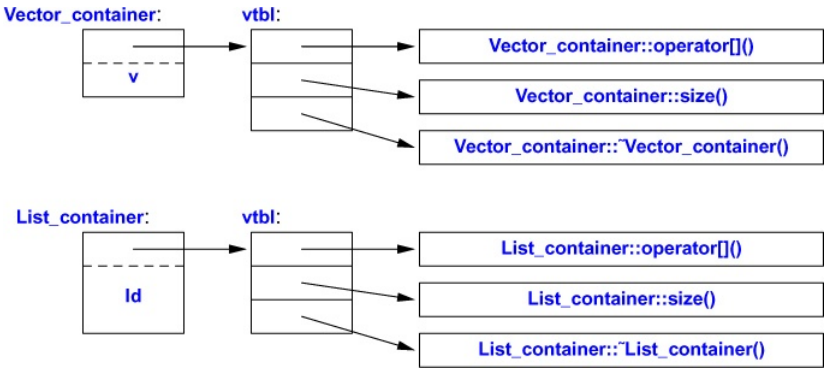

Figura 2.1: vtbl: virtual function table

dati sono sconosciuti alla funzione chiamante. All'implementazione della funzione chiamante è richiesta solo la conoscenza del puntatore in una vtbl nel Container e l'indice usato per la funzione virtuale. Questo tipo di chiamata è efficiente quasi quanto quella normale.

2.1.2 Gerarchia delle classi

L'esempio del Container è un caso molto semplice di gerarchia delle classi. Le classi sono ordinate tramite rapporti di derivazione. Ad esempio: Un esempio è vedere come vengono chiamate e

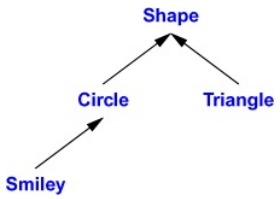

Figura 2.2: Gerarchia delle classi

implementate le varie classi:

```
class Shape {
public:
      virtual Point center() const =0; // pure virtual
```

```
    virtual void move(Point to) =0;

    virtual void draw() const = 0; // draw on current
        ↪ "Canvas"
    virtual void rotate(int angle) = 0;

    virtual ~Shape() {} // destructor
    // ...
};
```

Questa è la definizione della classe base, classe puramente virtuale.

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's
    ↪ elements by angle degrees
{
    for (auto p : v)–
        p>rotate(angle);
}
```

Possiamo quindi chiamare in maniera generica la classe astratta shape.

```
class Circle : public Shape {
public:
    Circle(Point p, int rr); // constructor

    Point center() const { return x; }
    void move(Point to) { x=to; }

    void draw() const;
    void rotate(int) {} // nice simple algorithm
private:
    Point x; // center
    int r; // radius
};
```

Qua vediamo l'implementazione di Circle, classe figlia di Shape.

```
class Smiley : public Circle { // use the circle as the base for a
    ↪ face
```

```
public:
    Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes) delete p;
    }
    void move(Point to);

    void draw() const;
    void rotate(int);

    void add_eye(Shape* s) { eyes.push_back(s); }
    void set_mouth(Shape* s);
    virtual void wink(int i); // wink eye number i

    // ...

private:
    vector<Shape*> eyes; // usually two eyes
    Shape* mouth;
};
```

Qua vediamo l'implementazione di Smiley, classe figlia di Circle.

```
void Smiley::draw()
{
    Circle::draw();
    for (auto p : eyes)–
        p>draw();–
    mouth>draw();
}
```

Abbiamo finalmente definito la funzione Smiley::draw(). Al suo interno vediamo che chiamiamo la classe genitore Circle come primo passaggio.

Vediamo anche che Smiley fa un override del distruttore della classe genitore. Un distruttore virtuale è essenziale per una classe astratta perchè un oggetto di una classe derivata è solitamente

manipolato tramite la sua interfaccia offerta dalla classe astratta. In particolare potrebbe decidere ci cancellare il puntatore alla classe base. Quindi, il meccanismo di chiamata alla funzione virtuale assicura che il distruttore corretto venga chiamato.

Una gerarchia di classi offre due benefici:

- Ereditarietà dell'interfaccia: un oggetto di una classe derivata può essere usato per qualunque oggetto venga richiesto. In quesot modo la classe base offre un'interfaccia alla classe derivata.
- La classe base offre una funzione o un dato che semplifica l'implementazione.

Le classi concrete sono usate esattamente come i tipi built in. Le classi in gerarchie vengono allocate tramite new e vi accediamo tramite puntatori o riferimenti. Per esempio consideriamo una funzione che legge dei dati che descrivono una forma da un input stream e costruisce l'appropriato oggetto Shape:

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is) // read shape descriptions
    ↪ from input stream is
{
    // ... read shape header from is and find its Kind k ...

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return new Circle{p,r};
    case Kind::triangle:
        // read triangle data {Point,Point,Point} into p1, p2,
            ↪ and p3
        return new Triangle{p1,p2,p3};
    case Kind::smiley:
        // read smiley data {Point,int,Shape,Shape,Shape}
            ↪ into p, r, e1 ,e2, and m
        Smiley* ps = new Smiley{p,r};–
        ps>add_eye(e1);–
```

```
        ps->add_eye(e2);-
        ps->set_mouth(m);
        return ps;
    }
}
```

```
void user()
{
    std::vector<Shape*>v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v); // call draw() for each element
    rotate_all(v,45); // call rotate(45) for each element
    for (auto p : v) delete p; // remember to delete elements
}
```

Ovviamente questo caso è estremamente semplificato. Ad esempio potremmo notare che il proprietario di un puntatore a Shape potrebbe non cancellarlo, in questo caso un puntatore libero potrebbe essere rischioso. Sarebbe quindi utile rutirnare un unique_ptr invece di un puntatore normale e immagazzinare il puntatore unico nel container:

```
unique_ptr<Shape> read_shape(istream& is) // read shape
    ↪ descriptions from input stream is
{
    // read shape header from is and find its Kind k

    switch (k) {
    case Kind::circle:
        // read circle data {Point,int} into p and r
        return unique_ptr<Shape>{new Circle{p,r}}; //
            ↪ §5.2.1
    // ...
}

void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
```

```
    draw_all(v); // call draw() for each element
    rotate_all(v,45); // call rotate(45) for each element
} // all Shapes implicitly destroyed
```

Ora l'oggetto è posseduto tramite il puntatore unico che verrà cancellato quando non sarà più utile, ovvero quando si uscirà dallo scope che l'ha creato. Per questa nuova versione dobbiamo implementare nuove versioni di draw_all() e rotate_all() che accettano un vector<unique_ptr<Shape». Ci sono alternative al dover scrivere tutto che vedremo poi.

2.2 Copia e sposta

Copiare un oggetto complesso significa farne una copia 1 a 1 tramite funzioni ben definite. Se si usasse il semplice simbolo si assegnazione non sarebbe copiato il valore ma il puntatore alla risorsa.

2.2.1 Copia

La copia è definita da due membri: un costruttore di copia e un assegnatore:

```
class Vector {
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
public:
    Vector(int s); // constructor: establish invariant, acquire
        ↪ resources
    ~Vector() { delete[] elem; } // destructor: release resources

    Vector(const Vector& a); // copy constructor
    Vector& operator=(const Vector& a); // copy assignment

    double& operator[](int i);
    const double& operator[](int i) const;
```

```
    int size() const;
};
```

2.2.2 Sposta

```
class Vector {
    // ...

    Vector(const Vector& a); // copy constructor
    Vector& operator=(const Vector& a); // copy assignment

    Vector(Vector&& a); // move constructor
    Vector& operator=(Vector&& a); // move assignment
};
```

Invece di dover copiare ogni signolo valore possiamo copiare il riferimento ai vettori e concatenarli, in questo modo non dobbiamo sprecare risorse aggiuntive.

& è detto lvalue che più o meno è intendibile come ”ciò che andrebbe a sinistra di un assegnamento” mentre && è detto rvalue ovvero ”ciò che andrebbe a destra di un assegnamento”

2.2.3 Gestione delle risorse

Offrendo costruttore, copia, spostamento e distruzione, un programma offre una gestione completa della vita di una risorsa.

2.3 Template

Quando qualcuno vuole un vettore non per forza è di double. Usiamo i template per descrivere concetti che è meglio tenere altamente generici.

2.3.1 Tipi parametrizzati

```
template<typename T>
class Vector {
private:
      T* elem; // elem points to an array of sz elements of type
          ↪ T
      int sz;
public:
      Vector(int s); // constructor: establish invariant, acquire
          ↪ resources
      ~Vector() { delete[] elem; } // destructor: release resources

      // ... copy and move operations ...

      T& operator[](int i);
      const T& operator[](int i) const;
      int size() const { return sz; }
};
```

Il prefisso template<typename T> rende T un parametro della dichiarazione. É il corrispettivo in C++ di ”per ogni T”. I membri della funzione possono essere definiti similarmente:

```
template<typename T>
Vector<T>::Vector(int s)
{
     if (s<0) throw Negative_size{};
     elem = new T[s];
     sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
     if (i<0 || size()<=i)
           throw out_of_range{”Vector::operator[]”};
     return elem[i];
}
```

Date queste definizioni possiamo creare ora un vettore così:

```
Vector<char> vc(200); // vector of 200 characters
Vector<string> vs(17); // vector of 17 strings
Vector<list<int>> vli(45); // vector of 45 lists of integers
```

E così lo possiamo usare:

```
void write(const Vector<string>& vs) // Vector of some strings
{
      for (int i = 0; i!=vs.size(); ++i)
            cout << vs[i] << '\n';
}
```

Se vogliamo dare la possibilità di usare un ciclo loop dobbiamo dare un inizio e una fine adatti:

```
template<typename T>
T* begin(Vector<T>& x)
{
    return &x[0]; // pointer to first element
}

template<typename T>
T* end(Vector<T>& x)
{
    return x.begin()+x.size(); // pointer to one-past-last
        ↪ element
}
```

Date queste definizioni possiamo scrivere:

```
void f2(const Vector<string>& vs) // Vector of some strings
{
      for (auto& s : vs)
            cout << s << '\n';
}
```

2.3.2 Funzioni template

Possiamo scrivere comodamente funzioni con i template. Vediamo la parola chiave auto, usata proprio per inferire automaticamente il tipo di x:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

2.3.3 Funzioni oggetto

Vengono chiamati anche funtori e sono oggetti che possono essere chiamati come funzioni. Perchè? Per programmare funzionalmente.

```
template<typename T>
class Less_than {
    const T val; // value to compare against
public:
    Less_than(const T& v) :val(v) { }
    bool operator()(const T& x) const { return x<val; } //
        ↪ call operator
};
```

Ad esempio questa funzione chiamata operator() implementa l'operatore (). Possiamo definire variabili di tipo Less_then tipo:

```
Less_than<int> lti {42}; // lti(i) will compare i to 42 using <
    ↪ (i<42)
Less_than<string> lts {"Backus"}; // lts(s) will compare s to
    ↪ "Backus" using < (s<"Backus")
```

Possiamo chiamare questi oggetti esattamente come chiamiamo le funzioni:

```
void fct(int n, const string & s)
{
      bool b1 = lti(n); // true if n<42
      bool b2 = lts(s); // true if s<"Backus"
      // ...
}
```

1.1.2