Programmazione C++

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

Jacopo De Angelis

2 luglio 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

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-

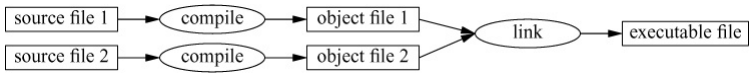

Figura 1.1: Processo di compilazione e unione

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".

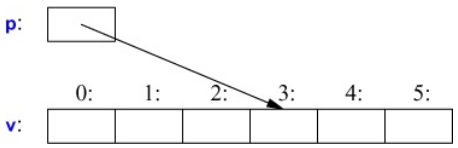

Figura 1.2: Puntatori e indirizzi

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:

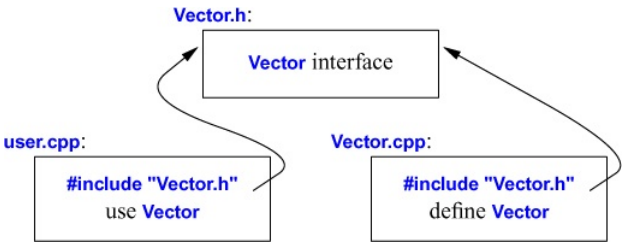

Figura 1.3: Uso dell'header

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
    // and return the result

    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 dati sono sconosciuti alla funzione chiamante. All'implementa-

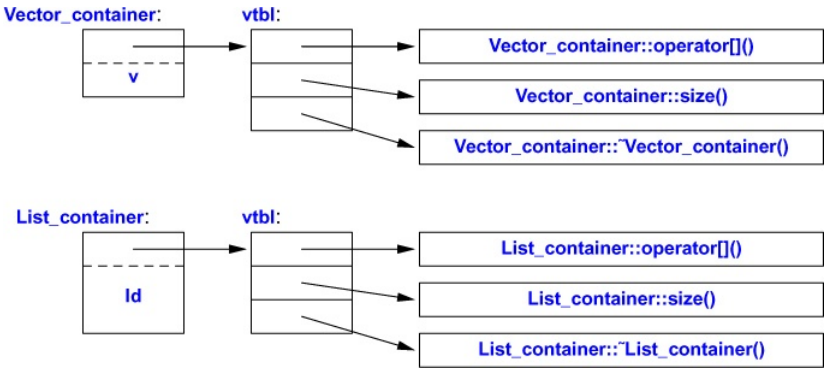

Figura 2.1: vtbl: virtual function table

zione 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

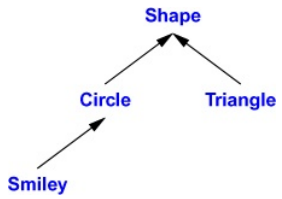

Figura 2.2: Gerarchia delle classi

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 automatica-

mente 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"
```

```
    // ...
}
```

2.3.4 Template variabile

Un template che può prendere un numero imprecisato di argomenti è detto ”variadic template”. Ad esempio:

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head); // do something to head
    f(tail...); // try again with tail
}

void f() { } // do nothing
```

La chiave è separare il primo argomento dal resto e trattarli in maniera ricorsiva.[1]

2.3.5 Alias

É molto comunque per i template dare ”un secondo nome” alle variabili in modo da rendere più comprensibile il corpo della funzione. Ad esempio:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};

template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
```

[1] Prolog FTW

```
void algo(Container& c)
{
    Vector<Element_type<Container>> vec; // keep results
        ↪ here
    // ...
}
```

Capitolo 3

Contenitori e algoritmi

3.1 Librerie

3.1.1 Headers e namespace della libreria standard

Ogni libreria standard è offerta tramite qualche header standardizzato, ad esempio:

- #include<string>
- #include<list>

La libreria standard è definita all'interno del namespace std. Per usare le sue libreria possiamo usare il prefisso std::.

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is peace", "Freedom is
    ↪ Slavery", "Ignorance is Strength"};
```

Ad esempio, per permettere l'utilizzo delle stringhe, all'inizio del file dobbiamo inserire:

```
#include<string> // make the standard string facilities
    ↪ accessible
using namespace std; // make std names available without
    ↪ std:: prefix
```

```
string s {"C++ is a –generalpurpose programming language"};
    ↪ // OK: string is std::string
```

Generalmente è una pessima scelta includere ogni nome da un namespace.

Selected Standard Library Headers	
<algorithm>	copy(), find(), sort()
<array>	array
<chrono>	duration, time_point
<cmath>	sqrt(), pow()
<complex>	complex, sqrt(), pow()
<fstream>	fstream, ifstream, ofstream
<future>	future, promise
<iostream>	istream, ostream, cin, cout
<map>	map, multimap
<memory>	unique_ptr, shared_ptr, allocator
<random>	default_random_engine, normal_distribution
<regex>	regex, smatch
<string>	string, basic_string
<set>	set, multiset
<sstream>	istrstream, ostrstream
<thread>	thread
<unordered_map>	unordered_map, unordered_multimap
<utility>	move(), swap(), pair
<vector>	vector

Figura 3.1: Lista ridotta degli header standard

Standard Container Summary	
vector<T>	A variable-size vector (§31.4)
list<T>	A doubly-linked list (§31.4.2)
forward_list<T>	A singly-linked list (§31.4.2)
deque<T>	A double-ended queue (§31.2)
set<T>	A set (§31.4.3)
multiset<T>	A set in which a value can occur many times (§31.4.3)
map<K,V>	An associative array (§31.4.3)
multimap<K,V>	A map in which a key can occur many times (§31.4.3)
unordered_map<K,V>	A map using a hashed lookup (§31.4.3.2)
unordered_multimap<K,V>	A multimap using a hashed lookup (§31.4.3.2)
unordered_set<T>	A set using a hashed lookup (§31.4.3.2)
unordered_multiset<T>	A multiset using a hashed lookup (§31.4.3.2)

Figura 3.2: Container

Capitolo 4

Funzioni base

4.1 Tipi e dichiarazioni

Per massimizzare la portabilità del codice è saggio essere espliciti in cosa è definito di base e cosa nell'implementazione corrente. Utilizzare solo tipi definiti dallo standard è il prezzo da pagare per la portabilità del codice. Un tipico esempio è di presentare tutte le dipendenze basate sull'hardware in forma di costanti e definire i timi negli header. Per supportare queste tecniche la libreria standard offre numeric_limits. Molti controlli sulle feature derivanti dall'implementazione può essere fatta tramite asserzioni statiche. Ad esempio

```
static_assert(4<=sizeof(int),"sizeof(int) too small");
```

I comportamenti non definiti sono peggiori, in quanto possono comportare in modi non desiderati. Ad esempio:

```
const int size = 4*1024;
char page[size];

void f()
{
        page[size+size] = 7; // undefined
}
```

L'esito più prevedibile è che si vada a sovrascrivere dei dati in aree di memoria non appartenenti all'array.

4.1.1 Implementazione

Un'implementazione di C++ può essere hostata o essere a sè stante. Un'implementazione hostata include tutte le librerie standard. Un'implementazione a sè stante può offrire meno librerie standard finchè le seguenti sono offerte:

Freestanding Implementation Headers		
Types	<cstddef>	§10.3.1
Implementation properties	<cfloat> <limits> <climits>	§40.2
Integer types	<cstdint>	§43.7
Start and termination	<cstdlib>	§43.7
Dynamic memory management	<new>	§11.2.3
Type identification	<typeinfo>	§22.5
Exception handling	<exception>	§30.4.1.1
Initializer lists	<initializer_list>	§30.3.1
Other run-time support	<cstdalign> <cstdarg> <cstdbool>	§12.2.4, §44.3.4
Type traits	<type_traits>	§35.4.1
Atomics	<atomic>	§41.3

Figura 4.1: Librerie obbligatorie

Le implementazioni freestanding sono pensate per codice che possa girare con il minimo delle risorse necessarie.

4.1.2 Caratteri di base

Il set di base si basa su ASCII-7. Per estenderlo un ambiente può mappare il set esteso in più modi, ad esempio usando nomi universali.

4.2 Tipi

4.2.1 Tipi fondamentali

- booleani (bool)
- Caratteri (char, wchar_t)
- Interi (int, short, long, long int, long long...)
- decimali (float, double, long double)
- void (assenza di informazione)

Da qua si possono costruire altri tipi come:

- puntatori (int*)
- Array (char[])
- riferimenti (double& e vettori<int>&&)

In può si possono definire tipi definiti dall'utente:

- strutture dati e classi
- enumerazioni

4.2.2 Caratteri

- char: 8 bit
- signed char: come har ma può contenere valori positivi e negativi
- unsigned char: come char ma garantisce il suo essere senza segno
- wchar_t: offre set estesi come unicode e la sua grandezza è derivante dalla sicurezza
- char16_t: UTF-16
- char32_t: UTF-32

Name	ASCII Name	C++ Name
Newline	NL (LF)	\n
Horizontal tab	HT	\t
Vertical tab	VT	\v
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Alert	BEL	\a
Backslash	\	\\
Question mark	?	\?
Single quote	'	\'
Double quote	"	\"
Octal number	*ooo*	*ooo*
Hexadecimal number	*hhh*	\x*hhh* ...

Figura 4.2: Caratteri speciali

4.2.3 Integrali

Possono essere decimali, ottali o esadecimali. Se inizia con 0x allora è un esadecimale, se inizia solo con 0 è un ottale.

Decimal	Octal	Hexadecimal
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

Figura 4.3: Tipi di intero

I numeri vengono distinti in base a forma, valore e suffisso:

4.2.4 Dimensioni

Alcuni tipi fondamentali di C++, ad esempio int, sono basati sull'implementazione hardware del sistema. Cosa vuol dire ciò? Che se vuoi fare un sistema portabile bisogna prendersi particolarmente cura di questi piccoli dettagli in modo da non generare problemi.

Forma	Valore	Suffisso	Contenitori adatti
	numero		int, long int, long long int
0, 0x	numero		int, unsigned int, long int, unsigned long int, long long int, unsigned long long int
	numero	u, U	unsigned int, unsigned long int, unsigned long long int
	numero	l, L	long int, long long int
0, 0x	numero	l, L	long int, unsigned long int, long long int, unsigned long long int
	numero	ul, lu, uL, Lu, Ul, lU, UL, LU	unsigned long int, unsigned long long int
	numero	ll, LL	long long int
0, 0x	numero	ll, LL	long long int, unsigned long long int
	numero	llu, llU, ull, Ull, LLu, LLU, uLL, ULL	unsigned long long int

Limitare l'impatto delle caratteristiche basate sull'implementazione è semplice, limitare l'impatto di caratteristiche basate sull'hardware è decisamente più difficile. Usare le librerie standard quando possibile è sempre un'ottima strategia.

La ragione per l'implementare più tipi di intero e più tipi di float è proprio questa, il dare agli sviluppatori più agio. Ci sono pochi assiomi riguardo le dimensioni:

- 1 ≡sizeof(char) ≤sizeof(short) ≤sizeof(int) ≤sizeof(long) ≤sizeof(long long)
- 1 ≤sizeof(bool) ≤sizeof(long)
- sizeof(char) ≤sizeof(wchar_t) ≤sizeof(long)
- sizeof(float) ≤sizeof(double) ≤sizeof(long double)
- sizeof(N) ≡sizeof(signed N) ≡sizeof(unsigned N)

Dove N può essere char, short, int, long, long long. In più è garantito che char ha minimo 8 bit, short 16, long 32.

Alcune implementazioni possono essere trovate tramite un semplice sizeof, altrimenti altre possono essere trovate in <limits>. Ad esempio:

```
#include <limits> // §40.2
#include <iostream>

int main()
{
      cout << "size of long " << sizeof(1L) << '\n';
      cout << "size of long long " << sizeof(1LL) << '\n';

      cout << "largest float == " <<
          ↪ std::numeric_limits<float>::max() << '\n';
      cout << "char is signed == " <<
          ↪ std::numeric_limits<char>::is_signed << '\n';
}
```

Le funzioni in <limits> sono constexpr, quindi possono essere usate senza overhead a runtime. Se si ha bisogno di tipi particolari si può includere la libreria <cstdint> che include, ad esempio:

```
int16_t x {0xaabb}; // 2 bytes
int64_t xxxx {0xaaaabbbbccccdddd}; // 8 bytes
int_least16_t y; // at least 2 bytes (just like int)
int_least32_t yy // at least 4 bytes (just like long)
int_fast32_t z; // the fastest int type with at least 4 bytes
```

L'header standard <cstddef> definisce un alias usato molto nelle librerie standard e nel codice utente: size_t. É un tipo implementato di unsigned int che può contenere la grandezza in byte di qualsiasi oggetto. Quindi è usato quando si vuole avere la grandezza di un oggetto. Ad esempio:

```
void* allocate(size_t n); // get n bytes
```

4.2.5 Allineamento

L'allineamento dei dati è come i dati vengono disposti in memoria. In certi casi questo dettaglio può permettere di risparmiare molto spazio, ad esempio nelle struct.

Per fare un esempio:

```
struct s1 { char a; short a1; char b1; float b; int c; char e;
    ↪ double f; };
// Dimensione: 32 byte
```

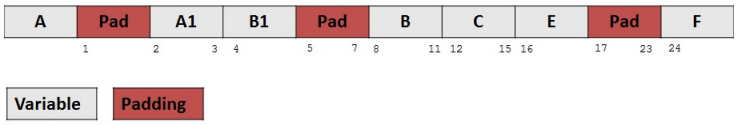

Figura 4.4: Struct ”non efficiente”

```
struct s1 { char a; short a1; char b1; float b; int c; char e;
    ↪ double f; };
// Dimensione: 32 byte
```

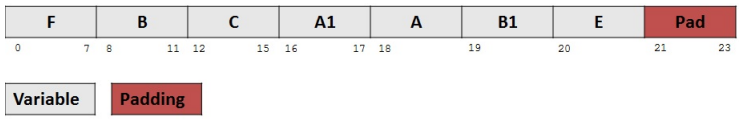

Figura 4.5: Struct ”efficiente”

La maggior parte delle volte questa cosa non interessa minimamente, si può programmare per anni senza nemmeno pensarci.

L’operatore alignof() può ritornare l’allineamento di un’espressione. Ad esempio:

4.3 Dichiarazione

Deve esserci sempre al massimo una dichiarazione per nome in C. Nel caso ci siano variabili derivanti da librerie esterne ci si può riferire con extern.

4.3.1 La struttura di una dichiarazione

Le dichiarazioni possono essere fatte in cinque blocchi:

Declarator Operators		
prefix	*	pointer
prefix	*const	constant pointer
prefix	*volatile	volatile pointer
prefix	&	lvalue reference (§7.7.1)
prefix	&&	rvalue reference (§7.7.2)
prefix	auto	function (using suffix return type)
postfix	[]	array
postfix	()	function
postfix	->	returns from function

Figura 4.6: Operatori di dichiarazione

1. prefisso opzionale (static o virtual)
2. tipo base
3. dichiaratore opzionale che include un nome (ad esempio p[8], n, *)
4. Suffisso opzionale per le funzioni (const, noexcept)
5. Inizializzatore opzionale ocorpo di funzione

4.3.2 Naming convention

File

- .cpp è l'estensione di C++, .c di C e gli header sono .h
- Per i file sorente usare file.cpp. Per gli header riferiti usare file.h. Si può anche usare file-impl.h per la dichiarazione di classi che non sono accessibili all'esterno del modulo. Se il file intero dichiara solo simboli interni al modulo, allora -impl è omesso
- usare il suffisso -doc.h per i file che contengono solo documentazione di Doxygen
- Per C++ cercare di dare al file lo stesso nome della classe che contengono. I file sono tutti all-lowercase
- Cercare di evitare classi con lo stesso nome ma in posti differenti

Guideline comuni tra C e C++

- Le macro per il preprocessore sono all-uppercase
- i nomi includono le guardie come GMX_DIRNAME_HEADERNAME_H
- i boolean hanno sempre prefisso b, seguiti da PascalCase
- le variabili enum hanno sempre prefisso e. Generalmente per enumerazioni usate tra più librerie la e è seguita da un altro prefisso, generalmente all-lowercase, per evitare coincidenze (ad esempio epbcNONE).
- Evitare abbreviazioni non comuni
- Se si usano acronimi seguire la policy di Microsoft:
  - Due lettere: all-uppercase
  - Tre o più lettere: Normalcase. Nel caso la prima lettera sia generalmente minuscola nell'uso dell'acronimo, allora usare all-lowercase

Codice

- PascalCase per classi, struct e typedef
- camelCase per funzioni e variabili
- si può usare un nome lo snake_case per variabili che sono fortemente rassomiglianti dei corrispettivi nella libreria standard-
- Le interfacce hanno suffisso Interface
- Le classi astratte hanno prefisso Abstract
- Per variabili associate ad un oggetto ha un _ finale
- Gli accessori per una variabile foo_ sono chiamati foo() e setFoo()
- Le variabili globali hanno prefisso g_
- Le variabili statiche hanno prefisso s_
- Le costanti globali hanno prefisso c_

4.3.3 Parole chiave

C++ Keywords					
alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
extern	false	float	for	friend	goto
if	inline	int	long	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template	this
thread_local	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

Figura 4.7: Parole chiave

4.3.4 auto e decltype()

Solitamente usiamo auto se vogliamo che l'interprete inferisca automaticamente il tipo del dato. In certi casi vogliamo solo dedurne il tipo senza inizializzare una variabile. Quindi possiamo usare decltype(expr) è il tipo di expr. Ad esempio, se sommiamo due matrici di tipo diverso, quale sarà il risultato? Con decltype possiamo saperlo.

```
template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b)
    ↪ -> Matrix<decltype(T{}+U{})>;
```

Quindi:

- auto: lo usiamo per inferire il tipo della variaile dal dato
- decltype: lo usiamo solo per conoscere il tipo di una funzione o di una variabile

Questi due strumenti sono importantissimi per quanto riguarda la programmazione generica.

4.3.5 Scope

- Local scope
- Class scope
- Namespace scope
- Global scope
- Statement scope
- Function scope

4.3.6 Inizializzazione

Può essere fatta in più modi:

- Tramite assegnamento
- Tramite lista di valori
- Tramite graffe (rischio di narrowing, ovvero di arrotondamento se necessario)

```
int a[] = { 1, 2 }; // array initializer
struct S { int x, string s };
S s = { 1, "Helios" }; // struct initializer
complex<double> z = { 0, pi }; // use constructor
vector<double> v = { 0.0, 1.1, 2.2, 3.3 }; // use list
    ↪ constructor
```

Quando inizializziamo dobbiamo considerare due fattori:

- il tipo dell'oggetto
- il tipo di inizializzazione

```
char v1 = 12345; // 12345 is an int
int v2 = 'c'; // 'c' is a char
T v3 = f();
```

Ad esempio usando l'inizializzatore riduciamo il rischio di conversioni

```
char v1 {12345}; // error: narrowing
int v2 {'c'}; // fine: implicit char->int conversion
T v3 {f()}; // works if and only if the type of f() can be
    ↪ implicitly converted to a T
```

Quando usiamo auto interviene solo il tipo dell'inizializzatore, quindi possiamo usare tranquillamente =

```
auto v1 = 12345; // v1 is an int
auto v2 = 'c'; // v2 is a char
auto v3 = f(); // v3 is of some appropriate type
```

In effetti, possiamo sfruttare a nostro vantaggio = con auto, poichè l'inizializzatore potrebbe creare sorprese inaspettate

```
auto v1 {12345}; // v1 is a list of int
auto v2 {'c'}; // v2 is a list of char
auto v3 {f()}; // v3 is a list of some appropriate type
```

Il che è abbastanza logico, basti pensare che

```
auto x0 {}; // error: cannot deduce a type
auto x1 {1}; // list of int with one element
auto x2 {1,2}; // list of int with two elements
auto x3 {1,2,3}; // list of int with three elements
```

4.4 Oggetti e valori

4.4.1 Durata degli oggetti

- Automatica: a meno che un programmatore specifichi altrimenti, un oggetto dichiarato in una funzione è creato con la sua definizione e distrutto quando il suo nome esce dallo scope.
- Statica: gli oggetti dichiarati globalmente o in un namespace e i valori static dichiarati in funzioni o in classi sono

creati e inizializzati solo una volta e vivono fino al termine del programma. Possono creare problemi in multithread, per questo bisogna implementare un sistema di lock

- Free store: usando new e delete possiamo creare oggetti la cui gestione è esplicita
- Oggetti temporanei: durano solo durante il loro uso. Ad esempio una varibile che viene valutata durante l'assegnamento avrà la durata della variabile ma il valore a destra dell'=, invece, avrà durata solo utile all'assegnamento
- Locali al thread: un oggetto dichiarato thread_local dura quanto il thread

4.4.2 Alias

In certi casi magari il nome di una variabile è troppo lungo, complicato, brutto. Oppure differenti tipi hanno lo stesso nome nel contesto.

```
using Pchar = char*; // pointer to character
using PF = int(*)(double); // pointer to function taking a
    ↪ double and returning an int

template<class T>
class vector {
    using value_type = T; // every container has a
        ↪ value_type
    // ...
};
template<class T>
class list {
    using value_type = T; // every container has a
        ↪ value_type
    // ...
};
```

Una vecchia notazione usava typedef, per motivi di consistenza è stata portata avanti

```
typedef int int32_t; // equivalent to "using int32_t = int;"
typedef short int16_t; // equivalent to "using int16_t =
    ↪ short;"
typedef void(*PtoF)(int); // equivalent to "using PtoF =
    ↪ void(*)(int);"
```

Ad esempio qua definire int come int32_t ci permette di comprendere subito che il programma mira ad un uso di int a 32 bit, quindi possiamo usare in maniere differenti le variabili, ad esempio

```
using int32_t = long;
```

il suffisso _t è convenzionale per gli alias.

Gli alias possono essere usati anche per introdurre un template, ad esempio:

```
template<typename T>
    using Vector = std::vector<T, My_allocator<T>>;
```

Non possiamo applicare modificatori come unsigned agli alias, ad esempio:

```
using Char = char;
using Uchar = unsigned Char; // error
using Uchar = unsigned char; // OK
```

Capitolo 5

Puntatori, array e reference

5.1 Puntatori

Per un qualsiasi tipo T, T* è un "puntatore a T". I puntatori possono contenere indirizzi di oggetti. Ad esempio

```
char c = 'a';
char* p = &c; // p holds the address of c; & is the address-of
    ↪ operator
```

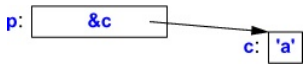

Figura 5.1: Puntatore

La funzione fondamentale dei puntatori è quella dello scollegamento del riferimento ("dereferencing"), ovvero riferirsi ad un oggetto tramite il suo puntatore. Questa operazione è chiamata indirizzamento ("indirection"). L'operatore di dereferenziamento unario è *. Ad esempio:

```
char c = 'a';
char* p = &c; // p holds the address of c; & is the address-of
    ↪ operator
char c2 = *p; // c2 == 'a'; * is the dereference operator
```

L'oggetto puntato da p è c, e il valore di c è 'a', quindi il valore di *p assegnato a c2 è 'a'.

I puntatori puntano a unità di memoria minima, il valore più piccolo che possono puntare è un char (notare che i bool hanno la stessa grandezza minimo).

Se si volesse valutare a livello di bit si potrebbe usare un operatore bitwish, come vedremo poi, o un bitset.

Sfortunatamente puntatori ad array e puntatori a funzioni hanno una notazione più complicata:

```
int* pi; // pointer to int
char* ppc; // pointer to pointer to char
int* ap[15]; // array of 15 pointers to ints
int (*fp)(char*); // pointer to function taking a char*
    ↪ argument; returns an int
int* f(char*); // function taking a char* argument; returns a
    ↪ pointer to int
```

5.1.1 void*

void* è un puntatore ad un oggetto di cui non si conosce la variabile restituita. Può essere usato solo per puntare a variabili, non a funzioni o ad un membro interno di una classe. In più void* può essere associato ad un altro void*, può essere usato per comparazioni e può essere convertito esplicitamente ad un altro tipo. Altre operazioni non sono sicure in quanto il compilatore non potrebbe sapere che variabile è puntata. Per poter usare un void* dobbiamo convertirlo esplicitamente ad un puntatore di un tipo conosciuto. Per esempio:

```
void f(int* pi)
{
    void* pv = pi; // ok: implicit conversion of int* to void*
    *pv; // error: can't dereference void*
    ++pv; // error: can't increment void* (the size of the
        ↪ object pointed to is unknown)
```

```
    int* pi2 = static_cast<int* >(pv); // explicit conversion
        ↪ back to int*

    double* pd1 = pv; // error
    double* pd2 = pi; // error
    double* pd3 = static_cast<double*>(pv); // unsafe
        ↪ (§11.5.2)
}
```

5.1.2 nullptr

nullptr rappresenta un puntatore a null, ovvero un puntatore che per ora inutilizzato. Può essere assegnato ad ogni puntatore ma non a tipi built-in:

```
int* pi = nullptr;
double* pd = nullptr;
int i = nullptr; // error: i is not a pointer
```

5.2 Array

Per un tipo T, T[size] è un ”array di size elementi di tipo T”. Gli elementi sono indicizzati [0, size-1].

Si può accedere agli elementi dell'array con la notazione array[n] dove n è l'indice dell'elemento. Gli accessi out of bound sono pericolosi e soprattutto spesso non sono controllati.

Il numero di elementi deve essere un'espressione costante, ovvero o un valore fisso o il risultato di una constexpr.

Array multidimensionali possono essere espressi come array di array.

Se si ha visogno che i dati siano contigui un array è ideale, altrimenti può creare problemi.

```
float v[3]; // an array of three floats: v[0], v[1], v[2]
```

```
char* a[32]; // an array of 32 pointers to char: a[0] .. a[31]

void f()
{
      int aa[10];
      aa[6] = 9; // assign to aa's 7th element
      int x = aa[99]; // undefined behavior
}

void f(int n)
{
      int v1[n]; // error: array size not a constant expression
      vector<int> v2(n); // OK: vector with n int elements
}
```

Un array può essere allocato staticamente, sullo stack o nel free store:

```
int a1[10]; // 10 ints in static storage

void f()
{
      int a2 [20]; // 20 ints on the stack
      int*p = new int[40]; // 40 ints on the free store
      // ...
}
```

Non esiste un assegnamento di un array e il nome di un array viene implicitamente convertito ad un puntatore verso il suo primo elemento.

In particolare, è meglio evitare di usare puntatori nelle interfacce perchè la loro conversione automatica a puntatori provoca molti problemi generalmente. Se si alloca un array sul free store allora ci si deve assicurare di usare delete[] sul suo puntatore solo dopo il suo ultimo utilizzo.

La maniera più sicura per avere un array sul free store è quello di avere un riferimento come una stringa, un vettore o un unique_ptr. Se viene allocato staticamente allora bisogna essere sicuri di non cancellarlo mai.

Uno degli usi più comuni di un array terminato da 0 sono le stringhe in C. In C++ alcune librerie standard si basano su di ciò.

5.2.1 Inizializzazione di array

Un array può essere inizializzato con una serie di valori. Nel caso venga dichiarato così, la grandezza dell'array verrà adattata al numero di elementi della lista. Se invece viene esplicitata la dimensione e si eccede con i dati in ingresso allora ci sarà un errore. Se invece se ne danno meno ci sarà un padding di 0 nelle celle non esplicitate:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };

char v3[2] = { 'a', 'b', 0 }; // error: too many initializers
char v4[3] = { 'a', 'b', 0 }; // OK


int v5[8] = { 1, 2, 3, 4 };
// is equivalent to
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

Non c'è una copia built-in per gli array. Non puoi inizializzare un array con un altro, nemmeno se dello stesso tipo, e non c'è un assegnamento di array.

```
int v6[8] = v5; // error: can't copy an array (cannot assign an
    ↪ int* to an array)
v6 = v5; // error: no array assignment
```

Quando si ha bisogno di un assegnamento ad una collezione di oggetti, meglio usare un vettore, un array o un valarray.

5.2.2 Stringhe

Attenzione ad alcune peculiarità delle stringhe:

- le stringhe generalmente vengono considerate come terminate da \0, quindi nel caso sia presente uno \0 a metà frase difficilmente il compilatore penserà che ci sia altro dopo.
- il compilatore considera un array di stringhe come una stringa continua

Raw string

In certi casi un backslash è solo un backslash ~~e un sigaro è solo un sigaro~~.

Per quei casi si può usare una raw string, indicata con un prefisso R alla dichiarazione della stringa:

```
string s = "\\w\\\\w"; // I hope I got that right

string s = R"(\w\\w)"; // I'm pretty sure I got that right

R"("quoted string")" // the string is "quoted string"
```

E se volessi includere nella stringa "(? In questo caso possiamo usare un delimitatore interno alla stringa, l'importante è che sia simmetrico:

```
R"***("quoted string containing the usual terminator
    ↪ (")")"***"
     // "quoted string containing the usual terminator (")"
```

E le regex? Poter usare raw string in una regex è molto utile, soprattutto per casi complicati. In compenso non esistono caratteri di escape.

Set di caratteri estesi

Nel caso si abbia bisogno di caratteri speciali, si possono sfruttare le wchar_t implicitamente:

```
"folder\\file" // implementation character set string
```

```
R”(folder\file)” // implementation character raw set string
u8”folder\\file” // UTF-8 string
u8R”(folder\file)” // UTF-8 raw string
u”folder\\file” // UTF-16 string
uR”(folder\file)” // UTF-16 raw string
U”folder\\file” // UTF-32 string
UR”(folder\file)” // UTF-32 raw string
```

5.3 Puntatori negli array

Il nome di un array può essere usato come puntatore per il primo elemento, ad esempio:

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v; // pointer to initial element (implicit conversion)
int* p2 = &v[0]; // pointer to initial element
int* p3 = v+4; // pointer to one-beyond-last element
```

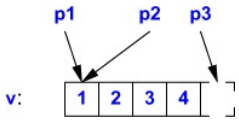

Figura 5.2: Puntatori negli array

Mai puntare a prima o dopo l'array.

La conversione implicita di un array al suo puntatore significa che la dimensione dell'array si perde. Proprio per questo motivo è preferibile usare vector, array e string della libreria standard, perchè superano questa difficoltà con size().

5.3.1 Navigare tra gli array

Si può accedere ad un array in due modi:

1. direttamente iterando su di esso

2. sfruttando i puntatori

```
void fi(char v[])
{
        for (int i = 0; v[i]!=0; ++i)
                use(v[i]);
}

void fp(char v[])
{
        for (char* p = v; *p!=0; ++p)
                use(*p);
}
```

Il prefisso * dereferenzia un puntatore, in questo modo *p è il carattere puntato da p e ++ incrementa il puntatore. Le due versioni sono identiche per il compilatore.

Per ogni array a e intero j, dove j è interno al range di a:

```
a[j] == *(&a[0]+j) == *(a+j) == *(j+a) == j[a]
```

5.3.2 Passare un array

Gli array non vengono passati per valore ma per puntatore al primo elemento.

```
void comp(double arg[10]) // arg is a double*
{
        for (int i=0; i!=10; ++i)
                arg[i]+=99;
}
void f()
{
        double a1[10];
        double a2[5];
        double a3[100];

        comp(a1);
        comp(a2); // disaster!
```

```
    comp(a3); // uses only the first 10 elements
};
```

A cosa può portare questo? Che gli array vengano passati per valore e non per indirizzo alle funzioni, rischiando di creare effetti collaterali.

5.4 Puntatori e costanti

C++ offre due significati a costante:

- constexpr: valutata durante la compilazione
- const: non modificata all'interno dello scope

Il ruolo di consexpr è di garantire la valutazione a compilazione, mentre const è di essere immutabile per le interfacce.

Le costanti hanno una grande utilità, in più bisogna tenere conto del fatto che:

- Avere delle costanti simboliche invece di stringhe all'interno del codice è spesso utile
- Molti puntatori vengono spesso letti ma non scritti
- In molte funzioni i parametri vengono letti ma non scritti

Poichè una costante non può essere mutata deve essere inizializzata durante la sua dichiarazione.

```
void f()
{
    model = 200; // error
    v[2] = 3; // error
}

void g(const X* p)
{
```

```
        // can't modify *p here
}
void h()
{
        X val; // val can be modified here
        g(&val);
        // ...
}
```

Quando viene usato un puntatore, due oggetti sono coivolti: il puntatore e l'oggetto a cui punta. Mettere come prefisso di un puntatore const rende l'oggetto, ma non il puntatore, una costante. Per dichiarare un puntatore costante dobbiamo usare *const invece del solo *. Ad esempio:

```
void f1(char* p)
{
        char s[] = "Gorm";

        const char* pc = s; // pointer to constant
        pc[3] = 'g'; // error: pc points to constant
        pc = p; // OK

        char *const cp = s; // constant pointer
        cp[3] = 'a'; // OK
        cp = p; // error: cp is constant

        const char *const cpc = s; // const pointer to const
        cpc[3] = 'a'; // error: cpc points to constant
        cpc = p; // error: cpc is constant
}

char *const cp; // const pointer to char
char const* pc; // pointer to const char
const char* pc2; // pointer to const char
```

Un modo per capire meglio questa scrittura è leggere da destra a sinistra, ad esempio "cp è una costante puntatore a un char" e "pc2 è un puntatore ad un char costante"

Un oggetto che è costante quando viene acceduto tramite un

puntatore potrebbe essere variabile quando acceduto tramite altre vie. Dichiarando un puntatore costante la funzione impedisce di modificare l'oggetto puntato. Per esempio:

```
const char* strchr(const char* p, char c); // find first
    ↪ occurrence of c in p
char* strchr(char* p, char c); // find first occurrence of c in p
```

La prima versione è usata per le strighe quando l'elemento non deve essere modificato e ritorna un puntatore ad una costante che non permette modifiche. La seconda versione è usata per stringhe modificabili.

5.5 Riferimenti

Un puntatore ci permette di passare anche grandi quantità di dati con il minimo sforzo. Il tipo di puntatore determina cosa si possa fare con i dati attraverso il puntatore. Usare un puntatore è differente dall'usare il nome di un oggetto in alcuni modi:

- Possiamo usare una sintassi differente, ad esempio *p invece di obj e p->m invece di obj.m
- Possiamo creare un puntatore che punti a diversi oggetti in momenti differenti
- Dobbiamo essere attendi quando usiamo puntatori in quando un puntatore potrebbe essere nullptr o puntare ad un oggetto che non era quello che ci aspettavamo

Queste differenze possono essere pesanti, ad esempio il controllo che non punti a nullptr. Molti problemi possono essere risolti tramite riferimenti (reference) che sono alias per un oggetto, solitamente implementato per contenere l'indirizzo di un oggetto e non impongono overhead di prestazioni comparati ai puntatori.

Le differenze con i puntatori sono:

- Puoi accedere ad un riferimento con esattamente la stessa sintassi del nome di un oggetto
- un riferimento si riferisce sempre all'oggetto con il quale è stato inizializzato
- non è un riferimento nullo e possiamo assumere che un riferimento si riferisca sempre ad un oggetto

```
template<class T>
class vector {
      T* elem;
      // ...
public:
      T& operator[](int i) { return elem[i]; } // return reference
          ↪ to element
      const T& operator[](int i) const { return elem[i]; } //
          ↪ return reference to const element

      void push_back(const T& a); // pass element to be
          ↪ added by reference
      // ...
};

void f(const vector<double>& v)
{
      double d1 = v[1]; // copy the value of the double referred
          ↪ to by v.operator[](1) into d1
      v[2] = 7; // place 7 in the double referred to by the result
          ↪ of v.operator[](2)

      v.push_back(d1); // give push_back() a reference to d1
          ↪ to work with
}
```

L'idea di passare per riferimento un valore ad una funzione è vecchissima. Ci sono tre tipi di riferimenti:

1. lvalue: si riferiscono ad oggetti che vogliamo cambiare
2. const: si riferiscono ad oggetti immutabili

3. rvalue: si riferiscono ad oggetti che non vogliamo preservare dopo l'uso

5.5.1 Riferimenti lvalue

Nel tipo della variabile una notazione X& significa "riferimento a X". É usata per riferirsi ad un lvalue

```
void f()
{
      int var = 1;
      int& r {var}; // r and var now refer to the same int
      int x = r; // x becomes 1

      r = 2; // var becomes 2
}
```

Per assicurare che il riferimento sia un nome di qualcosa dobbiamo inizializzare il riferimento

```
int var = 1;
int& r1 {var}; // OK: r1 initialized
int& r2; // error: initializer missing
extern int& r3; // OK: r3 initialized elsewhere
```

L'inizializzazione di un riferimento è differente dall'assegnamento.

```
void g()
{
      int var = 0;
      int& rr {var};
      ++rr; // var is incremented to 1
      int* pp = &rr; // pp points to var
}
```

Qua ++rr non incrementa il riferimento di rr, è invece applicato all'intero al quale si riferisce rr, ovvero var. Conseguentemente il valore di un riferimento non può essere cambiato dopo l'ini-

zializzazione. Per ottenere un puntatore ad un oggetto possiamo scrivere &rr. Non possiamo avere un puntatore ad un riferimento.

Notare una cosa:

```
void increment(int& aa)
{
      ++aa;
}

void f()
{
      int x = 1;
      increment(x); // x = 2
}

int next(int p) { return p+1; }

void g()
{
      int x = 1;
      increment(x); // x = 2
      x = next(x); // x = 3
}
```

Risultato? Uguale. Leggibilità? Decisamente migliore quella di next. Tendenzialmente è meglio evitare funzioni che modifichino le variabili passate.

5.5.2 Riferimenti rvalue

L'idea di avere più di un tipo di reference è per supportare differenti tipi di uso degli oggetti:

- un lvalue non costante si riferisce ad un oggetto sul quale l'utente può scrivere
- un lvalue costante è immutabile

- un rvalue si riferisce ad un oggetto temporaneo che l'utente può modificare, assunto che l'oggetto non verrà mai più usato

Il dichiaratore && significa "riferimento a rvalue". Non usiamo const in questo caso. Molti dei benefici risultanti dall'usare un riferimento rvalue sono legati alla scrittura dell'oggetto al quale si riferiscono. Sia un lvalue costante che un rvalue possono essere legati ad un rvalue, ciò che cambia è lo scopo:

- Possiamo usare un riferimento rvalue per implementare una lettura distruttiva per ottimizzare quello che altrimenti sarebbe una copia
- possiamo usare un riferimento ad un lvalue costante per prevenire modifiche degli argomenti

5.5.3 Riferimenti a riferimenti

Se si ha un riferimento ad riferimento si ottiene semplicemente un riferimento alla variabile del primo. Ma che tipo di riferimento?

```
using rr_i = int&&;
using lr_i = int&;
using rr_rr_i = rr_i&&; // "int && &&" is an int&&
using lr_rr_i = rr_i&; // "int && &" is an int&
using rr_lr_i = lr_i&&; // "int & &&" is an int&
using lr_lr_i = lr_i&; // "int & &" is an int&
```

In altre parole vince l'lvalue.

I riferimenti a riferimenti possono accadere come risultato di un alias o di un argomento per un template.

Capitolo 6

Strutture, unioni ed enumerazioni

- struct: sequenza di elementi di tipo arbitrario
- union: una struct che mantiene solo uno dei suoi elementi alla volta
- enum: set di nomi costante
- enum class: è una enum dove gli enumeratori sono all'interno dello scope dell'enumerazione e non ci sono conversioni implicite

6.1 Strutture

Un array è un'aggregazione di elementi dello stesso tipo. Nella sua forma più semplice, una struct è un aggreato di elementi di tipo arbitrario. Ad esempio:

```
struct Address {
    const char* name; // "Jim Dandy"
    int number; // 61
    const char* street; // "South St"
    const char* town; // "New Providence"
    char state[2]; // 'N' 'J'
```

```
    const char* zip; // ”07974”
};
```

Ciò definisce un tipo chiamato Indirizzo, consistente degli oggetti di cui si ha bisogno per inviare mail a qualcuno all'interno degli Stati Uniti.

Le variabili di tipo Indirizzo possono essere dichiarate come ogni altra variabile, e i membri individuali possono essere acceduti tramite dot notation (struct.elemento). Ad esempio:

```
void f()
{
    Address jd;
    jd.name = ”Jim Dandy”;
    jd.number = 61;
}

Address jd = {
    ”Jim Dandy”,
    61, ”South St”,
    ”New Providence”,
    {'N','J'}, ”07974”
};
```

notare che jd.state potrebbe non essere inizializzato dalla stringa ”NJ”. Le stringhe terminano con \0m quindi ”NJ” ha tre caratteri. Uno in più rispetto a quanto contenibile da jd.state.

Spesso le strutture sono accedute tramite puntatore usando l'operatore ->. Ad esempio>:

```
void print_addr(Address* p)
{
    cout << –p>name << '\n'
        << –p>number << ' ' << –p>street << '\n'
        << –p>town << '\n'
        << –p>state[0] << –p>state[1] << ' ' << –p>zip
            ↪ << '\n';
}
```

Quando p è un puntatore, p -> è equivalente e (*p).m

Alternativamente, una struct può essere passata per riferimento e acceduta tramite .:

```
void print_addr2(const Address& r)
{
      cout << r.name << '\n'
           << r.number << ' ' << r.street << '\n'
           << r.town << '\n'
           << r.state[0] << r.state[1] << ' ' << r.zip << '\n';
}
```

6.1.1 Layout delle struct

Un oggetto di una struct contiene i suoi membri nell'ordine di dichiarazione. Ciò può portare a un problema di allineamento (vedere pagina 54).

6.1.2 nomi delle struct

Il nome di una struct è subito utilizzabile tramite puntatore:

```
struct Link {
      Link* previous;
      Link* successor;
};
```

Mai istanziare direttamente una variabile dello stesso tipo:

```
struct No_good {
      No_good member; // error: recursive definition
};
```

Per permettere a due struct di fare riferimento a vicenda basta anche solo dichiararne il nome prima:

```
struct List; // struct name declaration: List to be defined later
```

```
struct Link {
        Link* pre;
        Link* suc;
        List* member_of;
        int data;
};

struct List {
        Link* head;
};
```

6.1.3 Strutture e classi

Una struct è semplicemente una classe dove i membri sono pubblici di default. Una struct quindi può avere funzioni e costruttori, ad esempio:

```
struct Points {
        vector<Point> elem;// must contain at least one Point
        Points(Point p0) { elem.push_back(p0);}
        Points(Point p0, Point p1) { elem.push_back(p0);
            ↪ elem.push_back(p1); }
        // ...
};

Points x0; // error: no default constructor
Points x1{ {100,200} }; // one Point
Points x1{ {100,200}, {300,400} }; // two Points
```

Non devi scrivere un costruttore solo per inizializzare le variabili, serve anche per ordinare gli argomenti, convalidarli, modificarli, stabilire invarianti, ecc.. Ad esempio:

```
struct Point {
        int x, y;
};

Point p0; // danger: uninitialized if in local scope (§6.3.5.1)
Point p1 {}; // default construction: {{},{}}; that is {0.0}
```

```
Point p2 {1}; // the second member is default constructed:
    ↪ {1,{}}; that is {1,0}
Point p3 {1,2}; // {1,2}

struct Address {
      string name; // "Jim Dandy"
      int number; // 61
      string street; // "South St"
      string town; // "New Providence"
      char state[2]; // 'N' 'J'
      char zip[5]; // 07974

      Address(const string n, int nu, const string& s, const
          ↪ string& t, const string& st, int z);
};
```

6.1.4 Equivalenza e struct

Due struct sono differenti tipi che possono avere anche gli stessi elementi. Quindi l'uguaglianza o l'assegnamento tra struct diverse non restituisce niente.

6.2 Unioni

Una union è un struct nella quale tutti i membri sono allocati allo stesso indirizzo, in questo modo la union oocupa sempre e solo tanto spazio quanto l'elemento più grande. Ovviamente, una union può contenere un solo valore alla volta.

```
enum Type { str, num };

struct Entry {
      char* name;
      Type t;
      char* s; // use s if t==str
      int i; // use i if t==num
};
```

```
void f(Entry* p)
{
      if (–p>t == str)
            cout << –p>s;
      // ...
}
```

I membri s ed i non possono mai essere usati allo stesso momento, si potrebbe rimediare specificando che entrambi dovrebbero essere un'unione.

```
union Value {
      char* s;
      int i;
};
```

Il linguaggio non tiene traccia di quale valore è contenuto nella union, quindi il programmatore deve fare ciò:

```
struct Entry {
      char* name;
      Type t;
      Value v; // use v.s if t==str; use v.i if t==num
};

void f(Entry* p)
{
      if (–p>t == str)
            cout << –p>v.s;
      // ...
}
```

Per evitare errori, uno può incapsulare una union in modo che la corrispondenza tra tipo e accesso all'unione possa essere garantita.

La union può certe volte essere usata male per "conversione di tipi". Questo uso errato è praticato soprattutto da programmatori che sono esperti in linguaggi che non hanno una conversione

esplicita, quindi devono barare. Ad esempio ”convertire” int in int* può essere assunto tramite equivalenza bitwise:

```
union Fudge {
      int i;
      int* p;
};

int* cheat(int i)
{
      Fudge a;
      a.i = i;
      return a.p; // bad use
}
```

Questa non è una conversione. Le unioni possono essere essenziali da un punto di vista prestazionale. Bisogna tenere da conto che sono molto prone ad errori, quindi è meglio considerare di usarle il meno possibile.

6.2.1 Union e classi

Tecnicamente, una union è un tipo di struct che è un tipo di classe. Le differenze sono:

- una union non può avere funzioni virtuali
- una union non può avere membri che siano reference
- una union non può avere una classe base
- una union ha un membro con un costruttore definito dall'utente, un'operazione di copia, una di spostamento o un decostruttore, e offre una speciale funzione delete per l'unione
- al massimo un membro della union può avere un inizializzaore
- una union non può essere usata come classe base

6.2.2 Union anonime

Per vedere come possiamo scrivere una classe che superi i problemi di una union consideriamo una viariante di Entry:

```
class Entry2 { // two alternative representations represented
    ↪ as a union
private:
    enum class Tag { number, text };
    Tag type; // discriminant

    union { // representation
        int i;
        string s; // string has default constructor, copy
            ↪ operations, and destructor
    };
public:
    struct Bad_entry { }; // used for exceptions

    string name;

    ~Entry2();
    Entry2& operator=(const Entry2&); // necessary because
        ↪ of the string variant
    Entry2(const Entry2&);
    // ...

    int number() const;
    string text() const;

    void set_number(int n);
    void set_text(const string&);
    // ...
};
```

La funzione di accesso può essere definita come:

```
int Entry2::number() const
{
    if (type!=Tag::number) throw Bad_entry{};
    return i;
```

```
};

string Entry2::text() const
{
      if (type!=Tag::text) throw Bad_entry{};
      return s;
};
```

Queste funzioni controllano il tag type e se una di queste corrisponde all'accesso desiderato, ritorna un riferimento al valore, altrimenti lancia un'eccezione. Una union implementata così è chiamata tagged union o discrimination union.

Queste funzioni di accesso in scrittura fanno lo stesso typechecking ma notare come tengono in considerazione il valore precedente.:

```
void Entry2::set_number(int n)
{
      if (type==Tag::text) {
            s.~string(); // explicitly destroy string (§11.2.4)
            type = Tag::number;
      }
      i = n;
}

void Entry2::set_text(const string& ss)
{
      if (type==Tag::text)
            s = ss;
      else {
            new(&s) string{ss}; // placement new: explicitly
                ↪ construct string (§11.2.4)
            type = Tag::text;
      }
}
```

L'uso di una union forza l'uso di alcune funzioni di basso livello per gestire la vita degli elementi della union. Questa è un'altra ragione per non prendere sottogamba le union.

Notare che la union nella dichiarazione di Entry2 non ha nome. Questo crea una union anonima. Una union anonima è un oggetto, non un tipo, e i suoi membri possono essere acceduti senza dover anteporre il suo nome. Questo significa che possiamo usare i membri di una union anonima esattamente come qualsiasi altro membro della classe, finchè ci ricordiamo che i membri della union possono essere utilizzati uno alla volta.

Entry2 ha un membro di un tipo con un operatore di assegnamento definito dall'utente, string, in questo modo l'operatore di assegnamento alla union comprende la cancellazione. Se vogliamo assegnare Entry2, dobiamo definire Entry2::operator=().

```
Entry2& Entry2::operator=(const Entry2& e) // necessary
    ↪ because of the string variant
{
    if (type==Tag::text && e.type==Tag::text) {
        s = e.s; // usual string assignment
        return *this;
    }

    if (type==Tag::text) s.~string(); // explicit destroy
        ↪ (§11.2.4)

    switch (e.type) {
    case Tag::number:
        i = e.i;
        break;
    case Tag::text:
        new(&s)(e.s); // placement new: explicit construct
            ↪ (§11.2.4)
        type = e.type;
    }

    return *this;
}
```

I costruttori e gli spostamenti possono essere definiti in maniera simile. Dobbiamo avere almeno uno o due costruttori per stabilire la corrispondenza tra type tag e valore. Il decostruttore deve gestire il caso string:

```
Entry2::~Entry2()
{
    if (type==Tag::text) s.~string(); // explicit destroy
        ↪ (§11.2.4)
}
```

6.3 Enumerazioni

Una enumeration (enum) è un tipo che può ontenere un set di interi specificati dall'utente.

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

Ci sono due tipi di enum:

- enum class: i nomi degli enumeratori sono locali all'enum e i loro valori non sono implicitamente convertiti
- plain enum: i nomi degli enumeratori sono nello stesso scope di enum e il loro valore è implicitamente convertito in integer

In generale è meglio preferire le prime.

6.3.1 enum classes

Una enum class è un'enumerazione fortemente tipizzata e all'interno dello scope. Ad esempio:

```
enum class Traffic_light { red, yellow, green };
enum class Warning { green, yellow, orange, red };// fire alert
    ↪ levels

Warning a1 = 7; // error: no int->Warning conversion
int a2 = green; // error: green not in scope
int a3 = Warning::green; // error: no Warning->int conversion
```

```
Warning a4 = Warning::green; // OK

void f(Traffic_light x)
{
      if (x == 9) { /* ... */ } // error: 9 is not a Traffic_light
      if (x == red) { /* ... */ } // error: no red in scope
      if (x == Warning::red) { /* ... */ } // error: x is not a
          ↪ Warning
      if (x == Traffic_light::red) { /* ... */ } // OK
}
```

Notare che gli enumeratori presenti in entrambe le enumerazioni non hanno conflitti grazie al loro scope ridotto.

Quando si dice che gli elementi dell'enumerazione sono converitit automaticamente in integer ci si riferisce al loro indice:

```
enum class Warning : int { green, yellow, orange, red }; //
    ↪ sizeof(Warning)==sizeof(int)

// If we considered that too wasteful of space, we could
    ↪ instead use a char

enum class Warning : char { green, yellow, orange, red }; //
    ↪ sizeof(Warning)==1


static_cast<int>(Warning::green)==0
static_cast<int>(Warning::yellow)==1
static_cast<int>(Warning::orange)==2
static_cast<int>(Warning::red)==3
```

Dichiarare una variabile Warning non come un int ma come una enum può indicare all'utente e al compilatore come gestirla, ad esempio:

```
void f(Warning key)
{
      switch (key) {
      case Warning::green:
          // do something
```

```
            break;
        case Warning::orange:
            // do something
            break;
        case Warning::red:
            // do something
            break;
        }
}
```

Poichè il giallo non è gestito il compilatore potrebbe avvisare della cosa.

Un enumeratore può essere inizializzato da una constexpr di tipo integer, ad esempio:

```
enum class Printer_flags {
    acknowledge=1,
    paper_empty=2,
    busy=4,
    out_of_black=8,
    out_of_color=16,
    //
};
```

É possibile dichiarare delle enum classe senza definirle, ad esempio:

```
enum class Color_code : char; // declaration
void foobar(Color_code* p); // use of declaration
// ...
enum class Color_code : char { // definition
    red, yellow, green, blue
};
```

Il valore di un tipo intero può essere esplicitamente convertito in un tipo enumerazione Il risultato di questa conversione è indefinito a meno che il valore non sia nel range del tipo dell'enumerazione, ad esempio:

```
enum class Flag : char{ x=1, y=2, z=4, e=8 };
```

```
Flag f0 {}; // f0 gets the default value 0
Flag f1 = 5; // type error: 5 is not of type Flag
Flag f2 = Flag{5}; // error: no narrowing conversion to an
    ↪ enum class
Flag f3 = static_cast<Flag>(5); // brute force
Flag f4 = static_cast<Flag>(999); // error: 999 is not a char
    ↪ value (maybe not caught)
```

Capitolo 7

Dichiarazioni

7.0.1 Riassunto delle dichiarazioni

- Dichiazione
  - declaration
  - $expression_{opt}$;
  - $statement - list_{opt}$
  - try $statement - list_{opt}$ handler-list

  - case constant-expression : statement
  - default : statement
  - break ;
  - continue ;

  - return $expression_{opt}$;
  - goto identifier ;
  - identifier : statement

  - selection-statement
  - iteration-statement
- selezione

  - if (condition) statement
  - if (condition) statement else statement
  - switch (condition) statement

- iterazione
  - while (condition) statement
  - do statement while (expression) ;
  - for (for-init-statement $condition_{opt}$; $expression_{opt}$) statement
  - for (for-init-declaration : expression) statement

- dichiarazione di lista
  - statement $statement - list_{opt}$

- condizioni
  - expression
  - type-specifier declarator = expression
  - type-specifier declarator expression

- lista di handler
  - handler $handler - list_{opt}$

- handler
  - catch (exception-declaration) $statement - list_{opt}$

7.1 Dichiarazione come inizializzazione

A meno che una variabile non sia dichiarata statica, l'inizializzatore è lanciato quando il thread passa attraverso la sua dichiarazione

Capitolo 8

Espressioni

8.1 Una semplice calcolatrice

Consideriamo una semplice calcolatrice. Essa consiste in:

- parser
- input
- tavola simbolica
- driver

8.1.1 Parser

```
program:
        end // end is end-of-input
        expr_list end

expr_list:
        expression print // print is newline or semicolon
        expression print expr_list

expression:
        expression + term
```

```
        expression –term
        term

term:
        term / primary
        term * primary
        primary

primary:
        number // number is a floating-point literal
        name // name is an identifier
        name = expression–
         primary
        ( expression )
```

In altre parole un programma una sequenza di espressioni separate da punto e virgola.

In altre parole il programma è definibile tramite un DFA che contiene alcuni nodi base. I nomi non sono da definire prima dell'uso. Questa sintassi, abbastanza complicata alla prima vista, è chiamata "discesa ricorsiva" (recursive descent). É un modo abbastanza popolare[1] per avere una tecnica top-down. In un linguaggio come C++, dove chiamare una funzione è abbastanza economico, è anche un metodo di programmazione efficiente.

I simboli terminali (ad esempio end, number, + e -) sono riconosciuti da un analizzatore lessicale, e i simboli non terminali da una funzione di analisi sintattica, expr(), term() e prim().

Come input, il parser usa un Token_stream che incapsula la lettura dei caratteri e la loro composizione in token. Questa è la funzione di un Token_stream: "tokenizza", ovvero trasforma lo stream di caratteri in token. Un token è una coppia kind-of-token, valore, come number, 123.45 dove 123.45 è stato trasformato in un float. In più il Token_stream nasconde la fonte del carattere. Vedremo poi come possono essere presi direttamente da una linea di comando o da altri input di sistema. La definizione di un token è come segue:

[1] ma non immediato come dice il libro

```
enum class Kind : char {
        name, number, end,
        plus='+', minus='-', mul='*', div='/', print=';',
            ↪ assign='=', lp='(', rp=')'
};

struct Token {
        Kind kind;
        string string_value;
        double number_value;
};
```

Rappresentare un token tramite il valore intero del suo carattere è conveniente ed efficiente e può aiutare le persone che usano i debugger. L'interfaccia di Token_stream è simile a questa:

```
class Token_stream {
public:
        Token get(); // read and return next token
        const Token& current(); // most recently read token
        // ...
};
```

L'implementazione è presentata dopo.

Ogni funzione del parser prende un bool in ingresso, chiamato get indicante se la funzione richiede la chiamata a Token_stream::get() per il prossimo token. Ogni funzione valuta la propria espressione e ritorna il valore. La funzione expr() gestisce addizione e sottrazione. Consiste di un singolo loop che cerca termini da aggiungere o sottrarre:

```
double expr(bool get) // add and subtract
{
    double left = term(get);

    for (;;) { // "forever"
            switch (ts.current().kind) {
            case Kind::plus:
                left += term(true);
                break;
```

```
            case Kind::minus:
                left -= term(true);
                break;
            default:
                return left;
            }
    }
}
```

La funzione prim() gestisce un valore base come expr() e term(), tranne per il fatto che essendo al livello più basso non ci serve un loop e va fatto il vero lavoro, non solo la gestione:

```
double prim(bool get) // handle primaries
{
    if (get) ts.get(); // read next token

    switch (ts.current().kind) {
    case Kind::number: // floating-point constant
    { double v = ts.current().number_value;
        ts.get();
        return v;
    }
    case Kind::name:
    { double& v = table[ts.current().string_value]; // find the
        ↪ corresponding
        if (ts.get().kind == Kind::assign) v = expr(true); //
            ↪ '=' seen: assignment
        return v;
    }
    case Kind::minus: // unary minus
        return -prim(true);
    case Kind::lp:
    { auto e = expr(true);
        if (ts.current().kind != Kind::rp) return error("')'
            ↪ expected");
        ts.get(); // eat ')'
        return e;
    }
    default:
        return error("primary expected");
```

```
    }
}
```

Quando un token è un numero, il suo valore è assegnato a number_value. Similarmente quando è un nome viene assegnato a string_value.

Notare che prim() legge sempre un token in più che utilizza per analizzare la sua espressione primaria. Ad esempio per capire se un nome va assegnato o semplicemente letto guarda il token successivo per capire.

8.1.2 Input

Leggere l'input è la parte più difficile spesso. Iniziamo definendo Token_stream:

```
class Token_stream {
public:
    Token_stream(istream& s) : ip{&s}, owns{false} { }
    Token_stream(istream* p) : ip{p}, owns{true} { }

    Token_stream() { close(); }

    Token get(); // read and return next token
    Token& current(); // most recently read token

    void set_input(istream& s) { close(); ip = &s;
        ↪ owns=false; }
    void set_input(istream* p) { close(); ip = p; owns = true;
        ↪ }

private:
    void close() { if (owns) delete ip; }

    istream* ip; // pointer to an input stream
    bool owns; // does the Token_stream own the istream?
    Token ct {Kind::end} ; // current token
};
```

Inizializziamo un Token_stream con un input stream dal quale ottenere i suoi caratteri. Il Token_stream implementa la convenzione di possedere un istream passato tramite un puntatore ma non tramite riferimento.

Un Token_stream contiene tre valori: un puntatore ad un input stream (ip), un Boolean (owns), indicante il possesso dell'input stream, e il token corrente (ct).

ct ha un valore di default. Le persone non dovrebbero chiamare current() prima di get(), ma se lo fanno almeno avranno un Token definito. Il valore iniziale per ct è Kind::end, così un programma che accede a current() in maniera errata non otterrà un valore che non sia quello di input.

Token_stream::get() viene presentato in due fasi:

1. viene offerta una versione semplice che impone un peso sull'utente
2. successivamente viene modificato una versione più complicata ma più facile da usare

L'idea è che get() legga un carattere, usi il carattere per decidere quale tipo di token richieda di essere composto, legga più caratteri di quanto sia necessario, e infine ritorna il Token rappresentante i caratteri letti.

La dichiarazione iniziale legge il primo carattere che non sia uno spazio da *ip in ch e controlla che l'operazione di lettura abbia successo:

```
Token Token_stream::get()
{

    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end}; // assign and return
```

Di default l'operatore » salta gli spazi bianchi (compresi newline, tabs ecc.) e lascia il valore di ch non cambiato se l'input fallisce. Conseguentemente ch==0 indica la fine dell'input.

Notare come venga usata la notazione nell'assegnamento, questo per evitare di dover scrivere in due stringhe

```
ct.kind = Kind::end; // assign
return ct; // return
```

Gli operatori vengono gestiti semplicemente facendoli passare con il loro valore:

```
case ';': // end of expression; print
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
    return ct={static_cast<Kind>(ch)};
```

I numeri sono gestiti in maniera simile:

```
case '0': case '1': case '2': case '3': case '4': case '5': case '6':
    ↪ case '7': case '8': case '9':
case '.':-
ip>putback(ch); // put the first digit (or .) back into the input
    ↪ stream
*ip >> ct.number_value; // read the number into ct
ct.kind=Kind::number;
return ct;
```

(per inciso, questo è un pessimo modo di scrivere uno switch).

Se un token non è la fine dell'input, un operatore, un carattere di punteggiatura o un numero, allora deve essere un nome. Un nome è trattato similarmente:

```
default: // name, name =, or error
    if (isalpha(ch)) {-
```

```
            ip>putback(ch); // put the first character back into
                ↪ the input stream
            *ip>>ct.string_value; // read the string into ct
            ct.kind=Kind::name;
            return ct;
    }
```

Infine possiamo avere un errore:

```
error("bad token");
return ct={Kind::print};
```

Quindi, ecco la funzione completa:

```
Token Token_stream::get()
{
    char ch = 0;
    *ip>>ch;

    switch (ch) {
    case 0:
        return ct={Kind::end}; // assign and return
    case ';': // end of expression; print
    case '*':
    case '/':
    case '+':
    case '-':
    case '(':
    case ')':
    case '=':
        return ct=={static_cast<Kind>(ch)};
    case '0': case '1': case '2': case '3': case '4': case '5': case
        ↪ '6': case '7': case '8': case '9':
    case '.':-
        ip>putback(ch); // put the first digit (or .) back into
            ↪ the input stream
        *ip >> ct.number_value; // read number into ct
        ct.kind=Kind::number;
        return ct;
    default: // name, name =, or error
        if (isalpha(ch)) {-
```

```
                ip>putback(ch); // put the first character back
                    ↪ into the input stream
                *ip>>ct.string_value; // read string into ct
                ct.kind=Kind::name;
                return ct;
            }

            error("bad token");
            return ct={Kind::print};
        }
}
```

8.1.3 Input di basso livello

Per rendere il programma più utilizzabile da un essere umano possiamo modificare cosa il programma si aspetti. Ad esempio, possiamo pensare ad un sostituto per il punto e virgola:

```
Token Token_stream::get()
{

    char ch;

    do { // skip whitespace except '\n'
        if (!ip>get(ch)) return ct={Kind::end};
    } while (ch!='\n' && isspace(ch));

    switch (ch) {
    case ';':
    case '\n':
        return ct={Kind::print};
```

La chiamata a ip->get(ch) legge un singolo carattere dall'input *ip. Di default get() non salta gli spazi allo stesso modo di ». Il test if (!ip–>get(ch)) ha successo se nessun carattere può essere letto da cin, in questo caso Kind::end viene restituito per terminare la sessione di calcolo. L'operatore ! è usato perchè get() restituisce true in caso di successo.

La funzione della libreria standard isspace() offre un test per gli spazi bianchi restituendo 0 se il carattere non è uno spazio bianco.

8.1.4 Gestione degli errori