Jaco's Fantastic Adventures

A C++ Referenza

2023

Contents

Chapter 1

Introduzione

1.1 Siti utili

- `https://godbolt.org` Compiler Explorer
- `https://explainshell.com/` per comandi shell unix
- `https://en.cppreference.com/w/` Reference
- `https://github.com/Programmazione-per-la-Fisica` repository lezioni + tutorial + labo
- `https://hackingcpp.com/` Risorse varie utili

1.2 Shell

- `<command> --help` guida rapida
- `cd <nome directory>` `cd /<directory path>`
  (per nomi con spazi `Pinco Pallino` o `'Pinco Pallino'`)
  `cd ..` cartella parente, `cd .` cartella attuale, `cd ~` directory home
- `pwd` posizione directory
- `ls` `ls /<dir path>` elenca contenuti directory
  `ls -l` lunga, `ls -a` con file nascosti, `ls -la` con file che iniziano per '.', `ls -A` senza file impliciti con '.' e '..'
- `mkdir <dir name>` crea cartella, `mkdir -p <dir1>/<dir2>/<dir3>` crea cartelle nidificate `rmdir <dir name>` rimuove dir (se vuota!)
- `rm <path>` rimuove file / cartella(e), `rm -f <...>` forza rimozione (ignora file e argomenti non esistenti e PROTEZIONI FILE) [1], `rm -d <...>` per directory vuote, `rm -r <...>` `rm -R <...>` ricorsivo: per eliminare cartelle, `rm -i <..>` chiede di confermare ogni rimozione, `rm -I <...>` chiede solo una volta per ≥ 3 argomenti, `rm -v <...>` 'verbose' : restituisce info su cosa si elimina
- `file <file path>` tipo file
- `cp <source> <destination>` copia file sorgente specificato nella destinazione: questa può essere un file (nome della copia) o una cartella (vi crea copia con nome del source) `cp -r <...> <...>` per copiare cartelle
- `mv <source> <destination>` sposta file (o cartella) : sintassi analoga alla copia. Può essere usato per **rinominare** spostando in stessa posizione di memoria ma con nome diverso file/cartella di destinazione

[1] NON USARE MAI `rm -rf` sulla cartella di root: si perde tutto!!

- **Aprire VS Code** `code .`
- **Compilare :**
  `g++ -Wall -Wextra name.cpp - o <compiledname>` → `./<compiledname>`
  Per usare sanitizer (controllare memory leak) `g++ -Wall -Wextra -fsanitize=address <...> -o <...>`
  Per produrre file senza linking `g++ ... -c <name>.cpp`
  Per salvare tutti i file intermedi generati nel processo di compilazione `g++ ... -save-temps <name>.cpp`
- **Formattazione :**
  `clang-format --dump-config -style=google > .clang-format` genera file di form.
  `clang-format -i <file name>` formatta file (`-i` impone modifiche direttamente sul file).
  Oppure da VS Code combinazione di tasti: Alt + ↑ + F
- **Scaricare file** `curl <link> -o <filename>`
- **Scaricare repository da GitHub** `git clone <link file .git>`
- **Installare pacchetti** `sudo apt install <name(s)>`
- **Aggiornamenti** verificare versione installata `cat /etc/os-release`, aggiornare distro `sudo do-release-upgrade`.
  Per aggiornamento periodicamente catalogo pacchetti `sudo apt update && sudo apt upgrade`
- **Problemi con utenti** se Ubuntu si apre come utente root, aprire Powershell Windows e digitare
  `ubuntu.exe config --default-user <USERNAME>`

Accesso a cartelle: WSL & Windows da Windows a WSL digitare nella barra indirizzi

`\\wsl$\Ubuntu\`

oppure `explorer.exe /home/user` `explorer.exe .`
da WSL a Windows digitare nella shell `cd /mnt/C/` (disco)

1.3 Commenti

```
int main() // commento singola linea
{
     /* commento
     multilinea */
}
```

Chapter 2

Basics

2.1 Variabili

Identificatori che assegnano nome a oggetti

```
int i; // dichiarazione
i = 23; // assegnazione
int j{40}; // dichiarazione e inizializzazione

// inizializzazione con braces {} : universale ma previene narrowing (conversione implicita):

int l{1.}; // error!

int const m{23}; // non modificabile

auto h{...}; // compiler deduces type from initializer (expression)
             // preserves const-ness of references (see later)!
```

2.1.1 Stringhe

Tipo non primitivo, da Standard Library

```
std::string ciao{"Letterale"};
ciao = ciao + " diverso"; // concatenazione
ciao.size(); // capacity
ciao.empty() // boolean: true if string is empty, otherwise false
ciao = "Pesce";
bool b{ciao == "Domani" || ciao != "Qui" || ciao > "a" || ciao < "a" || ciao <= "a" || ciao >= "a" };
// comparison
char c = ciao[2]; char d = ciao.back(); char e = ciao.front() // access to carachter
ciao.insert(); ciao.append(), ciao.erase() // insertion, removal
ciao.find(); // search

// PER CHAR:
#include <cctype>
...
std::isalpha(<char>); // verifica se carattere è alfabetico: 0 se non lo è,
                      // =/= 0 altrimenti
std::tolower(<char>); // converte in minuscolo
std::toupper(<char>); // converte in maiuscolo
```

2.2 Operatori

2.2.1 Aritmetici

```
+a; -a; a + b; a - b; a * b; a / b; a % b; ~a; a & b; a | b; a ^ b; a << b; a >> b;
```

2.2.2 Logici

```
!a; // not
a && b; // and
a || b; // or
```

2.2.3 Confronto

```
a == b; a != b; a < b; a > b; a >= b; a <= b;
```

2.2.4 Incremento

```
++a; --a; a++; a--;
```

2.2.5 Assegnazione

```
a = b; a += b; a -= b; a *= b; a /= b;
a %= b; // resto
a &= b; a |= b; a ^= b; a <<= b; a >>= b;
```

2.2.6 Accesso

```
a[b]; // subscript
*a; // dereference
&a; // address-of
a->b; // structure dereference
a.b; // access member
```

2.2.7 Altri

```
a(...); a, b; ? : ;
```

Chapter 3

Flow control

3.1 Statements / enunciati

```
a + 4; // expression statement
; // empty statement
{int b = 3; a += b;} // compound statement (block)
```

3.2 If, then, else

```
if ( <boolean condition> ) {
   ... // cond true
} else {
   ... // statement
} // else is optional
if (i < 1) <statement>; else <statement2>; // senza graffe
```

3.3 While

```
while ( <boolean condition> ) <statement>
// executes repeatedly until condition becomes false
```

3.4 For

```
for (<initial statement> ; <condition-expression> ; <expression>) <statement>
// alla fine di ogni iterazione: expression eseguita, poi condition-expr valutata

int i{1}; // inizializzata fuori dal loop
for ( ; i < 20; ++i) ; // senza graffe
```

3.4.1 Range-for

```
for( <range-declaration> : <range-expression> ) <statement>
// iterate on all elements of <range-expression>
// <range-declaration> declares variable of SAME TYPE of elements of range : CONST REFERENCE
```

3.5 Break, continue

```
for(...){
    ...
    continue; // jump to end of current iteration
    ...
    break; // terminate the loop
}
```

3.6 Numeri e input

```
#include <iomanip>
...
4. ; -1.4e7; 13.25E-2; // double
4.f ; -1.4e7f; 13.25E-2F; // float
int n
std::cout << std::setprecision(n) << 32.; // imposta precisione numero (inclusione header richiesta)
```

3.7 Funzioni matematiche standard

```
#include <cmath>
. . .
double x{. . .};
std::sqrt(x);
std::pow(x, .5); // (<base>, <esponente>)
std::sin(x);
std::log(x);
std::abs(x);
```

3.8 Conversione

```
int a = 1 + 2.34; // implicita
a += static_cast<int>(2.34 + 3.21); // esplicita
```

Chapter 4

Functions

```
<return-type> function-name( <parameter1>, ...); // declaration

<return-type> function-name( <parameter1>, ...){
    ... // statement
    ... function-name( ... ); // recursion: function can call itself
    return <expression>;      // expr must be convertible to ret type
                              //more return stat. in block (all of same type!!)
} // definition
// parameter type must be specified, name is optional

void ciao(){ // nothing returned + no parameter
    ...
    return;
}

int func(...);
auto func(...) -> int; // equivalenti
```

4.0.1 Overloading

```
int func(int);
int func(char);

func('a');
func(23);

// segnatura determinata SOLO DA PARAMETRI (no return-type)
```

4.0.2 Main

```
int main(){
return 0; // = successo (opzionale)
}
// altre sintassi:
#include<cstdlib>
...
    return EXIT_SUCCESS;
    return EXIT_FAILURE;
```

Per valore restituito, da shell `$?`

4.1 Doctest

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
...
```

```
// no main!!

// dopo funzioni:
TEST_CASE("Testing isqrt") {
CHECK(isqrt(0) == 0);
CHECK(isqrt(9) == 3);
CHECK(isqrt(10) == 3);
CHECK(isqrt(-1) == 0);
. . .
}
```

4.1.1 Per avere subcases

Per ogni subcase il TEST CASE è eseguito da capo.

```
TEST_CASE("vectors can be sized and resized") {
    std::vector<int> v(5);

    REQUIRE(v.size() == 5);
    REQUIRE(v.capacity() >= 5);
    // opzionale, necessario se si impongono requirements comuni per tutti i subcase

    SUBCASE("adding to the vector increases its size") {
        v.push_back(1);

        CHECK(v.size() == 6);
        CHECK(v.capacity() >= 6);
    }
    SUBCASE("reserving increases just the capacity") {
        v.reserve(6);

        CHECK(v.size() == 5);
        CHECK(v.capacity() >= 6);
    }
}
```

4.2 Conditionary expressions

```
<condition-expression> ? <expression-true> : <expression-false>;
// condition-expr must be (convertible to) bool
// <expr-true> and <expr-false> have to be of the same type
```

Chapter 5

Pointers & References

5.1 Pointers

```
int i{5};
int* p{&i}; // <type>* = pointer to type (different object type!), & = address-of operator
int** pp{&p}; // <type>** = pointer to pointer to type

int k{*p}; // * = dereference operator : restituisce reference, non oggetto!
q = nullptr; // null pointer
auto a = *q; // ERRORE!

struct S{
  int n;
  ...
  void f();
}
S ogg{...};
S* pt = &ogg;
pt->n;
pt->f();
// structure-dereference operator - equivale a (*<pointer>).<member>
```

5.1.1 Pass by pointer

```
int conta(std::string* s){
    ...
    ... *s ...;
    ...
}
std::string ciao{"bu"};
conta(&ciao);
```

5.2 References

```
int i{25};
int& ri{i}; // <type>& = reference to type (different type!)
int& rx; // ERRORE: dichiarazione sempre con inizializzazione!

int p{3};
ri{p}; // ERRORE: no rebinding (riassociazione)
```

5.2.1 Pass by reference

Parametri input: consigliata per tipi non primitivi, altrimenti by value. **Output:** returnare valore oppure passare by non const reference . **I/O:** non-const ref. Occhio con il `const` !

```
void func(int& n){
    ...
    ++n;
    ...
}

void func(std::string const& ciao){
    // cannot modify ciao (see later)
}
```

5.2.2 Const ref

```
std::string text{ · · · };
std::string& rtext{text}; // can read/modify text via rtext
std::string const& crtext{text}; // cannot modify, read-only!

std::string const text{ · · · };
std::string& rtext{text}; // ERROR: else could modify text via rtext
std::string const& crtext{text}; // ok, can only read text via crtext
```

5.2.3 Returning references

Mai per variabili locali della funzione o in generale oggetti che non sopravvivono alla fine dell'enunciato (finisce scope: ERRORE)

5.3 Enumeration

Tipo distinto con costanti dette enumerators (underlying type = int di default - modificabile). Tutti i valori dell'underlying type sono validi per l'enumeration.

```
enum class Operator { Plus, Minus, Multiplies, Divides };
auto op{Operator::Plus}; // op is of type Operator (:: scope resolution operator)
// valore di default = precedente en. + 1 (valore del primo è 0)
enum class Operator { Plus = -2, Minus, Multiplies = 42, Divides };
// possono essere assegnati valori

Operator op{55};

enum class byte : unsigned char { }; // select different underlying type
auto i{static_cast<int>(Operator::Plus)}; // conversioni all'und. type DEVONO ESSERE ESPLICITE
```

5.3.1 Unscoped enum

No need to use `Operator::`, conversion to the underlying type is implicit. (SCONSIGLIATO)

```
enum Operator { Plus, Minus, Multiplies, Divides }; // NB no class
```

5.4 Switch

Trasferisce controllo a uno tra enunciati multipli secondo valore espressione

```
double compute(char op, double left, double right)
{
    double result;
    switch (op) { // condition = only integral (int, char, bool) or enumeration value
      case '+':
        result = left + right;
        break;
      · · ·
      case '/':
        result = (right != 0.) ? left / right : 0.;
```

```
        break; // if not inserted, control 'falls through' next instruction!
      default: // at most one (not necessarily at the end!) - handles all other cases
        result = 0.;
    }

    return result;
}
```

In caso di `fall through` segnalazione del compilatore: se si intende silenziare warning aggiungere come attributo `[[fallthrough]]`.

Chapter 6

Data abstraction

6.1 Struct

Default = `public` (can be omitted)

```
struct <newtype> {
    <type> <name>; // data member, instance variable
}; // occhio!

<newtype> ad{};
auto b = ad.<name>; // operatore di accesso ai membri
```

6.2 Class

default = `private` (can be omitted)

```
class <newtype> {
  private:
      <type1> <name1>;
      ...
      <typeN> <nameN>;
  public:
    <newtype>(<type1> x, ..., <typeN> z) : <name1>{x}, ..., <nameN>{z} {}
      // CONSTRUCTOR : inizialization order is important!
    <newtype>(<type1> x) : <name1>{x}, ..., <nameN>{<default-valueN>} {}
      // multiple constructors!
    <newtype>(<type1> x, <type2> y) : <newtype>{x, <def-value2>, ..., <def-valueN>} {} // delegating
    <newtype>() : <name1>{<def-value1>}, ..., <nameN>{<def-valueN>} {}
    // DEFAULT CONSTRUCTOR (MUST be declared if there are other constr, otherwise automatically
    // generated)
    void func(){
      ... <name1> ...;
    } // member functions - methods (only f. with access to private memb.)
      // OVERLOADING is possible
    auto name1 const {
      return <name1>;
    } // methods that don't modify object MUST BE DECLARED CONST
};

<newtype>& manipulate(<newtype>& a){
    ... a.name1 ... ; // no access to private
} // implemented as FREE FUNCTION - suggested (better for manteniance)


class Complex {
    double r_{0.};
    double i_{0.};
```

```
  public:
    Complex(double x, double y) : r_{x}, i_{y} {}
    Complex() = default; // default implementation generated by compiler:
                         // (r_{0.} i_{0.}) - do not use {} !
};
```

6.2.1 Costruttori

```
Complex(double x = 0., y = 0.) : r_{x}, i_{y} {}
// default arguments: assumed if corresponding arg is absent (works for any function!)

explicit Complex(...) : ... {} // explicit constructor : prevents implicit conversion in
// initialization + impl. construction from list between braces:
double norm(Complex consy& c) {...}
norm(2.); // ERROR!
Complex foo(...){
  ...
  return {a, b};
} // ERROR!
```

6.2.2 This

```
struct T{
   void func(){
     ... this ...;
   } // POINTER of type T* to object for which method called
   void fit() const{
     ... this ...; // in const methods, is of type T const*
   }
};
```

6.3 Overloading

```
class <newtype>{
  ...
};

<newtype> operator@(<newtype> const& c, <newtype> const& d){
    ...
}
<newtype> ad{};
<newtype> bc{};

ad @ bc; // overloading di operatori (NB: l'associatività non può essere modificata)
```

Tipicamente si implementa `operator@` in termini di `operator@=`. Quest'ultimo restituisce (di solito) **reference** all'oggetto su cui si opera. Metodo non const, ma passare secondo addendo come const ref:

```
class <newtype>{
   ...
  public:
   ...
   <newtype>& operator+=(<newtype> const& rhu){
     ...
     return *this;
   }
};
```

6.3.1 Classi nidificate

Nested class **può accedere ai membri privati dell'outer c.** (no viceversa)

```
class Regression {
  . . .
  public:
  // can also be private, but in case cannot be named outside (e.g. one must use 'auto')
  class Result; // can be forward declared and later defined
  class Result { . . . }; // nested class definition
  Result fit() const { . . . }
};

Regression reg;
. . .
Regression::Result result{ reg.fit() }; // or auto result{. . .}
// necessario scope operator ::

class Tizio{
  ...
  class Caio; // can be declared inside and defined outside
};

class Tizio::Caio{ // don't forget scope operator
  using tiz = Tizio; // can use type alias for outer class
  tiz k_;
  ...
};
```

6.4 Assert

Espressione booleana il cui soddisfacimento è verificato **al runtime**: se fallisce il programma muore. **Abbondare nel codice!**.

```
#include <cassert>
...
assert(<boolean-cond>);
```

Per disabilitare: quando si compila da shell `g++ -DNDEBUG ...`
Possibile anche dalla Standard Library, per verificare condizione **AL COMPILE TIME**:

```
static_assert ( <bool-constexpr> , <message> );
// messaggio opzionale: DEVE ESSERE string literal
// (neanche una constexpr valutata al compile!!)

static_assert ( <bool-constexpr> );

// bool-constexpr deve essere costante bool o constexpr valutata al compile che
// viene convertita in bool
```

6.5 Eccezioni

Sono oggetti. Utilizzare **solo in contesti specifici** dove non è possibile comunicare errore in altro modo.

```
struct E{};

auto function3() {
   ... // executed
   throw E{}; // sollevare (lanciare) eccezione
   ... // NOT executed : flow control is transferred
}

auto function2(){
   ... // executed
   function3();
```

```
   ... // not executed
}

auto function1(){
   try{
     ... // executed
     function2();
     ... // not executed
   } catch (E const& e) { // HANDLER
     ... e ... // use e
   }
}
```

Si sollevano **by value** e si catchano **by reference (o const ref)**.
Eccezione si propaga fino a handler (`catch`) adatto (suitable, compatibile con il tipo di eccezione lanciata). Se non viene trovato programma viene `terminated`.

Catch multipli Viene scelto il primo che corrisponde a tipo eccezione sollevata: **l'ordine conta!**

```
auto read_from(std::filesystem::path const& p) {
  std::ifstream is(p);
  if (!is) {
    throw std::filesystem::filesystem_error{
    "read_from", p, std::make_error_code(std::errc::invalid_argument)
    };
  }
  . . .
}

auto g() {
  try {
    read_from("/tmp/data");
    . . .
  } catch (std::filesystem::filesystem_error const& e) {
    std::cerr << e.path1();
  } catch (std::exception const& e) {
    std::cerr << e.what();
  } catch (...) { // it's really three dots!!! (specific syntax)
    std::cerr << "unknown exception";
  }
}
```

6.5.1 Eccezioni nei costruttori

utili nel caso non sia possibile inizializzazione corretta (impossibile stabilire / soddisfare invariante di classe).
Possibile utilizzare un tipo di eccezione dalla Standard:

```
#include <stdexept>
. . .
class Rational {
  . . .
  Rational(int num = 0, int den = 1) : n{num}, d{den} { // constructor
    if (d == 0) {
      throw std::runtime_error{"denominator is zero"};
      // costruito con stringa o string literal
    }
    . . .
  }
};

auto do_computation() {
  . . .
```

```
  Rational{n,m} // m here happens to be 0
  . . .
}

try {
  do_computation();
  . . .
} catch (std::runtime_error const& e) {
  std::cerr << e.what() << '\n';
  // metodo what() per recuperare stringa
}
```

Nota: eccezioni nella standard implementate tramite gerarchia polimorfica. E.g. classe derivata (indirettamente) da `std::runtime_error` (a sua volta derivata dalla base `std::exception`) è `std::filesystem::filesystem_error`, che si lancia in caso di problemi nella gestione dei `path`. Il catch permette di recuperare i percorsi in memoria coinvolti.

6.6 Verificare tipo

Funzioni con return type booleano. Vedi C++ Ref per specifici

```
#include <type_traits>

is_void(...);
...
is_floating_point(...);
...
is_pointer(...);
...
is_fundamental(...);
...
is_const(...);
...
is_signed(...); is_unsigned(...);
...
is_polymorphic(...);
...
```

6.7 Type aliases

Nome alternativo per tipi esistenti. **NON** introduce nuovo tipo (nessun overloading per funzioni!)

```
using Length = double; // using = keyword!
typedef double Length; // equivalent, old alternative
```

Aliases utilizzati dentro classi per dichiarare tipi

```
class FitResult { . . . };

class Regression {
    . . .
  public:
    using Result = FitResult;
    Result fit() const { . . . }
};

Regression::Result result{ reg.fit() }; // result is of type FitResult
```

6.8 Structured binding

Dichiarare più variabili inizializzandole a valori di membri di una struct, secondo **ordine!**
Anche come const reference ai membri:

```
struct Point {
double x;
double y;
};
Point p{1.,2.};
auto [a, b] = p;
std::cout << a << ' ' << b; // print 1 2

// COME REFERENCE
Point p{1.,2.};
auto& [a, b] = p; // a is a ref to p.x, b is a ref to p.y
a = 3.;
b = 4.;
std::cout << p.x << ' ' << p.y; // print 3 4
```

Chapter 7

Templates

Modello di classe o funzione con tipi (e non solo) come parametri.
Concetto di base nell'approccio definito **generic programming**
`Template` è keyword.

7.1 Class Template

NON costituisce type di per sè!

```
// NB: < ... > sonp sintassi specifica!

template<typename FP> // or, template<class FP>
class Complex {
   static_assert(std::is_floating_point_v<FP>); // (*)
   FP r;
   FP i;
  public:
   Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
   FP real() const { return r; }
   FP imag() const { return i; }
};

// (*) compile-time check + type introspection
// possibile definire tipi accettabili come parametri di template

Complex<double> d; // instantiation of a Complex<double> type
Complex<float> f;  // DIFFERENT TYPE than d
d + f;             // possible ERROR
```

Per impostare valori di default dei parametri:

```
template<typename FP = <default type> >
```

7.1.1 Type aliases as templates

Possibile ad esempio bloccare parametri di template:

```
// array of 3 T's
template<class T> using Array3 = std::array<T, 3>;

Array3<double> a; // std::array<double, 3>
```

7.2 Function template

```
template<typename FP> // or template<class FP>
auto norm2(Complex<FP> const& c) {
   return c.real() * c.real() + c.imag() * c.imag();
```

```
}
auto nf = norm2(f); // nf is of type float
auto nd = norm2(d); // nd is of type double
```

Possibile anche

```
template<typename FP>
auto norm2(Complex<FP> const& c)
{ return c.real() * c.real() + c.imag() * c.imag(); }
```

Per l'istanziazione del template devono essere noti **tutti gli argomenti**. Possono essere dedotti **dagli argomenti della chiamata della funzione**:

```
template<class F> F norm2(Complex<F> const& c) {· · ·}
Complex<float> f;
norm2(f); // no need to specify norm2<float>(f)
norm2<float>(f); // ok, be explicit
```

L'importante è che nel caso si possano dedurre tutti quelli necessari!
Analogo per template di classe con la chiamata del constructor:

```
template<class FP> class Complex {· · ·};

Complex d; // error, cannot deduce FP
Complex<double> e; // ok
Complex f{1.}; // ok, Complex<double>
```

Oltre ai tipi, possono esserci altri parametri. Se valori, devono essere noti **necessariamente al compile time** ed essere di tipo integral, enumeration, pointer, reference, $std :: nullptr_t$, floating-point, literal (con specifiche caratteristiche, vd C++Ref)

```
template<class T, int N> struct array { · · · };
template<class T, T v> struct integral_constant { · · · };
// possibile anche mischiare type e non-type!
```

Chapter 8

Standard Library

Perché sprecare tempo a imparare quando l'ignoranza è immediata? (Hobbes)

8.1 Namespace

Per partizionare spazio nomi e evitare conflitto tra identificatori. **Richiesto per progetto!** Si consiglia di mettere tutte le entità di un componente di software (classi, funzioni, etc.) dentro stesso namespace.

```
// in <vector>
namespace std {
template<class T> vector { · · · };
}
```

Possono essere riaperti, anche in altri file (tranne il `namespace std` !) e nidificati.

8.1.1 Namespace alias

```
namespace ch = std::chrono;
auto t0 = ch::system_clock::now(); // std::chrono::system_clock::now()
// it's the same namespace, not a new one!
```

8.1.2 Using declaration + directive

```
using std::string; // DECLARATION: rende visibile simbolo di un namespace
string s; // possibile accedere a membro senza scope operator

using namespace std; // DIRECTIVE: visibili TUTTI i simboli
string s;
```

NOTA BENE `using directive` sconsigliato: non si può chiudere e si rischia conflitto di identificatori, specie in scope globale e header file

8.2 Iteratori

Range sono semiaperti a dx e rappresentati da 2 iteratori: `first` corrisponde al primo elemento, `last` all'estremo superiore $\notin$ range, ovvero la posizione di memoria immediatamente adiacente (successiva) all'ultimo elemento.
Un range è vuoto quanto `first == last`

8.2.1 Operazioni su iteratori

Sintatticamente analoghe a quelle sui pointer

```
std::vector<int> v {1,2,3};
auto it = v.begin();
auto its = v.end();
*its; // UNDEFINED BEHAVIOUR: no elemento del vettore!
```

```
// DEREFERENCE:
std::cout << *it; // print 1
*it = 4; // v is now {4,2,3}
// ACCESS TO MEMBERS:
struct Point {
double x;
double y;
};
...
std::vector<Point> vect {Point{1,2}, Point{3,4}};
auto its = vect.begin();
std::cout << (*itv).x; // print 1
std::cout << itv->x; // equivalent
// INCREMENTO (Spostamento di posizione):
++itv;
std::cout << itv->x; // print 3

// CONFRONTO
auto itn = vect.begin();
itv == itn; // verifica se puntano a stesso elemento
// NOTA: funziona anche per iteratori su diversi vettori con stesso tipo di elementi
// (diverso tipo: ERRORE), ovviamente sempre falso
```

Nota: possibile creare vettore di stringhe (implementate nella standard come container di caratteri) e accedere a metodi analoghi

```
std::vector<std::string> v {"hello", "world"};
auto itv = v.begin(); // itv points to the first string in the vector
auto its = itv->begin(); // its points to the first character
// of the first string ('h');
// a string is a container of characters
```

8.2.2 Gerarchia degli iteratori, operazioni aggiuntive

In ordine decrescente di potere (operazioni supportate, versatilità) con operazioni `Lettura / Accesso / Scrittura / Iterazione / Confronto`

1 RandomAccessIterator

`=*p / -> [] / *p= / ++ -- + - += -= / == != < > <= >=`

2 BidirectionalIterator

`=*p / -> / *p= / ++ -- / == !=`

3 ForwardIterator

`=*p / -> / *p= / ++ / == !=`

4a OutputIterator

`=*p / -> / / ++ / == !=`

4b InputIterator

`/ / *p= / ++ /`

8.3 Vector

Contenitore dinamico: dimensione varia al runtime, layout di memoria contiguo, **da utilizzare come default**

```
#include <vector>

std::vector<int> a; // empty vector of ints
std::vector<int> b{2}; // one element, initialized to 2
std::vector<int> c(2); // two elements (!), value-initialized (to default, 0 for int)
std::vector<int> d{2,1}; // two elements, initialized to 2 and 1
std::vector<int> e(2,1); // two elements, both initialized to 1

auto f = b; // crea copia del vettore
f == b; // operatore supportato: verifica corrispondano gli elementi (qui true)
```

Occhio alla sintassi nell'inizializzazione: se vi è nel programma un costruttore che accetta

```
std::initializer_list
```

8.3.1 Operazioni / metodi

```
std::vector<int> vec{};

vec.size(); // dimensione (NB è unsigned int!)
            // per farci operazioni bene forzare prima conversione implicita in int
vec.empty(); // verifica se è vuoto o no (booleano)
// NB diverso da vettore con tutti valori di default!
vec[<n>]; // accesso all' n-esimo elemento (int)
vec[0]; // primo elemento: IL CONTEGGIO PARTE DA 0
vec[vec.size()]; // ERRORE: l'ultimo elemento corrisponde a vec.size()-1

vec.push_back(5); // aggiunge elemento alla fine

vec.begin(); // iteratore corrispondente al first
vec.end(); // iteratore corr. al last
// (vd. dopo)

vec.insert( <iterator>, <value>); // inserisce elemento (inizializzato a <value>) nella posizione
                                  // PRECEDENTE a quella indicata da <iterator>
vec.insert( vec.end(), <value> ); // analogo a push_back
vec.insert( <it>, <count>, <value>); // inserisce <count> elementi con valore <value> prima di <it>
                                     // NOTA: <count> è size_type (unsigned)
vec.insert( <it>, <it_first>, <it_last>); // ins. RANGE di elementi da <it_first> a <it_last>
                                          // prima di <it> NOTA BENE: first e last non possono
                                          // essere iteratori su vec!
std::initializer_list<int> l{1,2,3,4};
vec.insert( <it>, l); // inserisce elementi della lista nel vettore
                      // NB tipo el. deve essere lo stesso di vec

vec.erase( <iterator> ); // rimuove elemento *<iterator>
vec.erase( <it_first>, <it_last> ); // rimuove range: <it_first> a primo elemento, <it_last> a
                                    // estremo superiore

vec.capacity(); // capacità complessiva allocata per il vettore
                // (non solo slot inizializzati)
vec.reserve(size_type <newcap>); // se <newcap> > capacità, rialloca spazio
                                 // NB nel caso INVALIDA tutti ITERATORI!
                                 // (non modifica size)
vec.resize(size_type <count>); // se <count> = size, nulla
                               // se <count> < size, colma differenza
                               // inserendo di default elementi
                               // se <count> > size, riduce ai primi
                               // <count> elementi
vec.resize(size_type <count>, const int <value>); // analogo, ma nel secondo
                                                  // caso colma inserisce <value>
```

Nota bene: dopo `erase`, iteratori a elementi rimossi non sono più validi: utilizzo o operazioni su di essi danno `undefined behaviour`. Nel caso dei `vector`, **anche iteratori a elementi successivi, fino a** `end` sono invalidati. Analogamente nel caso di riallocazione di memoria, **tutti gli iteratori** che puntano al vettore sono invalidati!

8.4 Array

Dimensione stabilita al compile, layout contiguo

```
#include <array>

std::array< <Type>, <Size> > arr{}; // Size deve essere specificato al compile!
std::array<int,2> c{}; // 2 ints, value-initialized (0 for int)
std::array<int,2> d{1}; // 2 ints, initialized to 1 and 0
                        // (following order!)

auto e = d; // copia tutto l'array, tipo di e: std::array<int,2>

d.at( <n> ); // n è size_type: restituisce n-esimo elemento
             // e SOLLEVA std::out_of_range se n >= size
d[ <n> ]: // n-esimo elemento, senza bound checking

d.begin(); d.end(); d.empty(); // analogo a vettori, per empty sempre falso se N > 0
```

8.5 Algoritmi

Si consiglia utilizzo: efficienti, Funzioni generiche che operano su range, implementati come template ('duale' dei container nella STL). Necessario

```
#include <algorithm>
```

Nota: algoritmi che trovano / modificano specifici elementi restituiscono iteratori, non valori!
Varie categorie:

8.5.1 Non modifying

```
all_of
any_of
none_of
// verifica se predicato vero per tutti, almeno uno o nessun elemento di range

for_each
for_each_n
// applica funzione a tutti / primi n elem di range

count
count_if
// numero elem. che soddisfano criteri

mismatch
// prima posizione in cui differiscono 2 range

find
find_if
find_if_not
// primo

find_end
// ultima occorrenza di determinata sequenza in un range

find_first_of
// occorrenza qualsiasi di data sequenza in range
```

```
adjacent_find
// primi due elementi adiacenti che soddisfano predicato (default: sono uguali)

search
// cerca prima occorrenza di dato range in altro range

search_n
// cerca n copie consecutive di elemento in range
```

8.5.2 Modifying

```
copy
copy_if
// copies a range of elements to a new location

copy_n
// copies a number of elements to a new location

copy_backward
// copies a range of elements in backwards order

move
// moves a range of elements to a new location

move_backward
// moves a range of elements to a new location in backwards order

fill
// copy-assigns the given value to every element in a range

fill_n
// copy-assigns the given value to N elements in a range

transform
// applies a function to a range of elements, storing results in a destination range

generate
// assigns the results of successive function calls to every element
// in a range

generate_n
// assigns the results of successive function calls to N elements
// in a range

remove
remove_if
// removes elements satisfying specific criteria

remove_copy
remove_copy_if
// copies a range of elements omitting those that satisfy specific criteria

replace
replace_if
// replaces all values satisfying specific criteria with another value

replace_copy
replace_copy_if
//copies a range, replacing elements satisfying
//specific criteria with another value
```

```
swap
// swaps the values of two objects

swap_ranges
// swaps two ranges of elements

iter_swap
// swaps the elements pointed to by two iterators

reverse
// reverses the order of elements in a range

reverse_copy
// creates a copy of a range that is reversed

rotate
// rotates the order of elements in a range

rotate_copy
// copies and rotate a range of elements

shift_left
shift_right
// shifts elements in a range

random_shuffle
shuffle
// randomly re-orders elements in a range

sample
// selects n random elements from a sequence

unique
// removes consecutive duplicate elements in a range

unique_copy
// creates a copy of some range of elements that contains
// no consecutive duplicates
```

8.5.3 Partitioning

```
is_partitioned
// determines if the range is partitioned by the given predicate

partition
// divides a range of elements into two groups

partition_copy
// copies a range dividing the elements into two groups

stable_partition
// divides elements into two groups while preserving their relative order

partition_point
// locates the partition point of a partitioned range
```

8.5.4 Sorting

```
is_sorted
// checks whether a range is sorted into ascending order
```

```
is_sorted_until
// finds the largest sorted subrange

sort
// sorts a range into ascending order

partial_sort
// sorts the first N elements of a range

partial_sort_copy
// copies and partially sorts a range of elements

stable_sort
// sorts a range of elements while preserving order between equal elements

nth_element
// partially sorts the given range making sure that it is partitioned by
// the given element
```

Per range ordinati:

8.5.5 Binary search

```
lower_bound
// returns an iterator to the first element not less than the given value

upper_bound
// returns an iterator to the first element greater than a certain value

binary_search
// determines if an element exists in a partially-ordered range

equal_range
// returns range of elements matching a specific key
```

8.5.6 Set

```
includes
// returns true if one sequence is a subsequence of another

set_difference
// computes the difference between two sets

set_intersection
// computes the intersection of two sets

set_symmetric_difference
// computes the symmetric difference between two sets

set_union
// computes the union of two sets
```

8.5.7 Altro

```
merge
// merges two sorted ranges

inplace_merge
// merges two ordered ranges in-place: second one attached to the end of
// the first
```

Senza requisiti di ordine:

8.5.8 Min / Max

```
max
// returns the greater of the given values

max_element
// returns the largest element in a range

min
// returns the smaller of the given values

min_element
// returns the smallest element in a range

minmax
// returns the smaller and larger of two elements

minmax_element
// returns the smallest and the largest elements in a range

clamp
// clamps a value between a pair of boundary values: returns upper or lower
// bound if out of the interval, otherwise returns value
```

8.5.9 Comparison

```
equal
// determines if two sets of elements are the same

lexicographical_compare
// returns true if one range is lexicographically less than another

lexicographical_compare_three_way
// compares two ranges using three-way comparison: < = >
```

8.5.10 Numeric

Parallelismo per esecuzione in parallelo (con più cores processore)

```
#include <execution>
...
std::<alg1-name>( std::execution::par, <alg1 arguments> );
std::<alg2-name>( std::execution::par, <alg2 arguments> );
```

8.6 Concepts

Insieme di requisiti che un tipo deve soddisfare al compile, limitano tipi utilizzabili come argomenti di template; introdotti esplicitamente da C++20.

```
template<class T>
concept Incrementable = requires(T t) { ++t; };

template<Incrementable T>
auto advance(T& t) { ++t; }

int i {42};
advance(i); // ok, int is a model of Incrementable

struct S {};
```

```
S s;
advance(s); // error, S is not a model of Incrementable
```

8.7 Function objects

Possibile costruire oggetti funzione tramite overloading dell'operatore di chiamata, da passare come argomenti ad algoritmi.

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42 // or directly: LessThan42{}
); // *it == 32
```

Possibile passare parametri come variabili private di classe (o pubbliche di struct)

```
class LessThan {
   int m_;
  public:
   explicit LessThan(int m) : m_{m} {}
   auto operator()(int n) const {
     return n < m_;
   }
};

LessThan lt42 {42};
auto b1 = lt42(32); // true
// or: auto b1 = LessThan{42}(32);
```

8.8 Lambda expression

Metodo rapido per creare function object senza nome (usa e getta), definiti `closures`.

```
[ <captures> ]( <params> ) <specs requires> { <body> }
```

<captures> è il prefisso: possibili sintassi:

```
std::find_if(..., [](int n) {
                     return n < 42;
                   }
); // non cattura nulla

... [=] ... // cattura tutte le variabili necessarie (tra quelle dichiarate
            // e inizializzate prima) BY VALUE

auto m = ...;
... [m] ... // cattura m (in genere tutte le var. indicate) BY VALUE

... [m = <value> ] ... // inizializza e cattura m BY VALUE

... [&] ... // cattura tutte le variabili necessarie (tra quelle dichiarate
```

```
            // e inizializzate prima) BY REFERENCE

auto m = ...;
... [&m] ... // cattura m (in genere tutte le var. indicate) BY REFERENCE

... [=, &k] ... // cattura tutto il necessario BY VALUE e k BY REFERENCE

// NB se il primo termine nella cattura è = (&), eventuali successivi non possono essere
// catturati by value (by ref) !

... [&, k] ... // tutto BY REF e k BY VALUE
```

Nota 1: le variabili globali sono disponibili senza cattura!
Note 2: possibile catturare `current object`: (possibili `[&, ..., this, ...]` , `[=, ..., this, ...]` (da C++20), `[=, ..., *this, ...]` (da C++17))
Nota 3: catturare solo variabili che si utilizzano (altrimenti viene comunque allocato spazio dal costruttore)

<params> sono i parametri dell'operatore () sovraccaricato e implementato come template (parametri = tipi argomenti), che vengono passati alla lambda nell'esecuzione dell'algoritmo.
Ogni lambda crea implicitamente una classe differente con metodo overloaded e variabili catturate come private.
Per <specs required> le sintassi:

```
[...](...) {...} // nulla: operator() implementato come const:
                 // non può modificare variabili catturate (!)

[...](...) mutable {...} // può modificare variabili catturate by value
// NOTA BENE: catturate by reference sono modificabili DI DEFAULT,
// impossibile catturare by const& !

[...](...) -> <type> {...} // specifica return type (default: auto - 'generic lambda')

[...](...) mutable -> <type> {...} // ordine se presenti entrambi
```

8.8.1 Std::function

Function wrapper polimorfico che può prendere qualsiasi entità chiamabile con determinata segnatura (funzioni, metodi ma anche oggetti funzione).

```
#include <functional>
...
using Function = std::function<int(int,int)>; // signature
```

Chapter 9

Compilation model

One Definition Rule: ogni entità può essere definita una volta sola per translation unit! Class, templates e funzioni e variabili *inline* possono essere definite in più TU **se le definizioni sono identiche** (token-by-token nel source code)
Nota bene: violazioni ODR non sempre diagnosticate da compiler ma possono causare malfunzionamenti eseguibile.

Strutturazione componente software

- *Header file / file di intestazione, interfaccia* (`.hpp`) **dichiarazioni** delle free functions, **definizioni** di classi con **dichiarazioni** di metodi e **definizioni** template
- *Source file / file di implementazione* (`.cpp`) **definizioni** di free functions e metodi e di qualsiasi altra entità necessaria per l'implementazione
- *File di test* (`.cpp`)

9.0.1 Header file

L'header è incluso tramite `#include "name.hpp"` nel file di implementazione e in tutti gli altri file di progetto che abbiano necessità di accedere a funzioni e classi ivi presenti.
Ogni funzione definita in un header file deve essere `inline`:

- per le free functions deve essere esplicitato
- per i metodi delle classi e i template è implicito

Metodi possono essere **dichiarati** nella classe e **definiti** nel source file, utilizzando l'operatore di scope.
Se il metodo è definito fuori dalla definizione della classe ma è in un header file, va dichiarato `inline` !

```
// in statistics.hpp

class Sample{
   ...
  public:
   void add(double); // declaration
   ...
};
...
inline void Sample::add(double d){
  ... // definition
}
```

9.0.2 Include guards

Per evitare che un header file venga incluso più volte nella stessa translation unit (e.g. se viene incluso anche un altro h.f. che a sua volta include il primo).

```
// at the beginning of EVERY header file!

// e.g. need to include result.hpp and sample.hpp with the latter
// including the former

// file result.hpp
#ifndef RESULT_HPP
#define RESULT_HPP
. . .
// classes, functions, templates, ...
. . .
#endif
```

9.1 Build systems (CMAKE)

File di configurazione `CMakeLists.txt`. Esempio dal mock project del prof (commenti secondo sintassi)

```
cmake_minimum_required(VERSION 3.16)
project(mandelbrot_sfml VERSION 0.1.0)
# nome e versione progetto

# abilita il supporto per i test, tra cui l'opzione BUILD_TESTING usata sotto
include(CTest)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# richiedi l'uso di C++17, senza estensioni non-standard offerte dal compilatore usato
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS ON)

# abilita warning
string(APPEND CMAKE_CXX_FLAGS
      " -Wall -Wextra -Wpedantic -Wconversion -Wsign-conversion -Wcast-qual -Wformat=2"
      " -Wundef -Wshadow -Wcast-align -Wunused -Wnull-dereference"
      " -Wdouble-promotion -Wimplicit-fallthrough -Wextra-semi -Woverloaded-virtual"
      " -Wnon-virtual-dtor -Wold-style-cast")

# abilita l'address sanitizer e l'undefined-behaviour sanitizer in debug mode
string(APPEND CMAKE_CXX_FLAGS_DEBUG " -fsanitize=address,undefined -fno-omit-frame-pointer")
string(APPEND CMAKE_EXE_LINKER_FLAGS_DEBUG " -fsanitize=address,undefined -fno-omit-frame-pointer")

# richiedi il componente graphics della libreria SFML, versione 2.5
# le dipendenze vengono identificate automaticamente
find_package(SFML 2.5 COMPONENTS graphics REQUIRED)

add_executable(mandelbrot_sfml main.cpp)
target_link_libraries(mandelbrot_sfml PRIVATE sfml-graphics)

# se il testing e' abilitato...
# per disabilitare il testing, passare -DBUILD_TESTING=OFF a cmake durante la fase di configurazione
if (BUILD_TESTING)

  # aggiungi l'eseguibile all.t
  add_executable(all.t all.t.cpp complex.t.cpp)
  # aggiungi l'eseguibile all.t alla lista dei test
  add_test(NAME all.t COMMAND all.t)

endif()
```

Comandi da shell

- **Installazione (con SFML)** `sudo apt install cmake libsfml-dev`
- **Configurazione area di compilazione** all'interno di cartella con file progetto eseguire `cmake -S . -B <newdir>` : viene creata sottocartella `<newdir>` (nome a scelta, di solito `build`) dove avverrà compilazione. Nota: se il file `CMakeLists.txt` non viene modificato, non è necessario eseguire questo passaggio più di una volta.
- **Conf. in debug** `cmake -S . -B <newdir>/debug -DCMAKE_BUILD_TYPE=Debug`
- **Conf. in release mode** `cmake -S . -B <newdir>/debug -DCMAKE_BUILD_TYPE=Release`
- **Compilazione** `cmake --build <newdir>`
  Se fallisce **basta** modificare file sorgente .hpp e .cpp (senza riconfigurazione)
- **Compilare con test** `cmake --build <newdir> --target test`

Possibile, nel caso si aggiunga il testing, eseguire direttamente l'eseguibile dei test (vd. file `.txt`)

Chapter 10

Explicit Memory Management

Costruire oggetti sullo heap. Gestione esplicita: costruzione implica responsabilità su gestione e **distruzione**! (Sullo stack è implicita, automatica)

```
new []; // creates array: allocates memory + runs constructor
delete []; // delete array: run destructor + deallocate memory

int p* = new int{5}; // new restituisce pointer
delete p; // chiamato sul pointer
// p non modificato: UNICA possibile azione è riassegnazione
p = ...;

int* q = nullptr;
delete q; // well defined: does nothing

void func(){
   int* pt = new int{45};
   ... // no delete
} // MEMORY LEAK : pt's scope ended (implicitly destructed)

// DOUBLE DELETE : trying to call destructor more than one time on same
// object!
```

Nota bene: pointer è unico riferimento a oggetto creato sullo *heap*: se il suo scope finisce prima della distruzione dell'oggetto creato questo è perduto!

```
Sample* create()
{
  auto rc = new Sample{};
  rc->add(· · ·);
  · · ·
  return rc;
}

auto use()
{
  auto ru = create();
  ru->stats();
  · · ·
  delete ru;
} // ok!
```

Sconsigliato utilizzo raw pointer in caso di eccezioni: se porzione di codice con `delete` non viene eseguita, si rischia leak!

10.1 Array nativi

Sequenza contigua di oggetti in memoria. Usare molta precauzione nell'utilizzo!

```
int a[3] = {123, 456, 789}; // int[3], the size must be a constant
                            // and can be deduced from the initializer
++a[0];
a[3]; // undefined behavior (from 0 to size-1)

// "arrays decay to pointers at the slightest provocation" (!!!)
auto b = a; // int*, size information lost
            // restituisce pointer al primo elemento !
            // anche quando passato come parametro funzione!

++b; // increase by sizeof(int) - incrementabile
assert(b == &a[1]); // true

*b = 654; // dereferenziabile
b += 2; // increase by 2 * sizeof(int)
if (b == a + 3) { ... } // ok, but not more than this
```

Allocazione dinamica array: quando dimensione nota solo al runtime.

```
int*p = new int[3] {14, 56, 144};
...
delete [] p; // sintassi specifica: altrimenti per decadimento pointer
             // non elimina array!
```

Difficoltà gestione: per passare correttamente a funzione necessario

- Passare separatamente anche dimensione
- Da C++20 utilizzare `std::span` (header `<span>`). É template di classe che contiene tipo elementi e estensione (funziona con qualsiasi sequenza di oggetti contigua in memoria)

Cmq **sconsigliato**.

10.2 Null Terminated Byte Strings (NTBS)

Array di caratteri non nulli seguiti dal carattere nullo (`char0` oppure `0`), noto anche come *C-Strings*. É di tipo `char[]` / `char*` se modificabili o `char const[]` / `char const*` se non.
Per la lunghezza `std::strlen` (legge array fino a trovare il null carachter)
Il tipo degli `string literals` è `char const[N]` con `N` costante.
É possibile inizializzare una `std::string` da una NTBS; inoltre tramite il metodo `c_str` si può ottenere la rappresentazione NTBS da una stringa. Returna un pointer all'array (`char const*`), che viene invalidato se si chiamano metodi `non const` sulla stringa di partenza. **Permette di estrarre i singoli caratteri**.

```
std::string stan{"Domani"};
char const* pt2 = stan.c_str();
std::cout << pt2[0]; // prints D
```

10.3 Main

Può avere due forme

```
int main() {
   ...
   return 0; // successo (implicito)
             // qualsiasi altro valore = failure
}

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

argc corrisponde al numero di argomenti della riga di comando, `argv[]` è un array di C-stringhe (`char*` in quanto C-stringhe sono a loro volta array di caratteri) che rappresentano gli argomenti.

- `argv[0]` è il nome del programma (di solito)
- `argv[argc]` è `nullptr`

Esistono librerie per gestire e interpretare command line.

10.4 Allocazione stile C

Strumenti ereditati da C, **SCONSIGLIATISSIMI**: gestiscono memoria non inizializzata!

```
#include <cstdlib>
...
void* malloc(size_t s); // alloca s byte
void* calloc(size_t n, size_t s); // alloca n volte s byte inizializzati a 0

void free(void* p); // libera spazio allocato in precedenza
void* realloc(void* p, size_t s); // cambia dimensione array puntato da p,
                                  // oppure crea copia allocando s byte e
                                  // poi libera p
```

`size_t` è tipo standard per dimensioni (unsigned int)

10.5 Considerazioni globali su allocazione dinamica

Quando non necessario, non utilizzarla (usare stack)!
Maggiore responsabilità, utilizzo di tempo e spazio in memoria: oltre a quanto visto necessario allocare su stack per pointer + per conformazione *heap* ('groviera') tempo non determinabile mappatura disponibilità di memoria da parte degli allocatori.
Dove sono necessarie **garanzie sul tempo di esecuzione** è proibito allocare dinamicamente!
Per Address Sanitizer (ASan) vedi istruzioni compilazione, **NB** non trova sempre tutti i problemi di memoria!

10.6 Destructor

- **Ordine di distruzione è opposto a quello di costruzione / definizione** (da tenere a mente quando si ordinano variabili private di classe!)
- Sotto-oggetti distrutti ricorsivamente

```
{
  S s{· · ·};
  · · ·
  T t{s};
  · · ·
} // t is destroyed first, then s
```

Una classe può avere **un solo distruttore**

```
class Array {
    int* m_data{};
    int m_n{};
   public:
    Array(int n) : m_data{new int[n]}, m_n{n} {}
    //non è possibile sostituire m_data{new int[m_n]}:
    //conta l'ordine di costruzione per il compiler
    //stabilito dalla private e non dal constructor
    ~Array() {  // destructor, declared as ~ClassName()
        delete[] m_data;
    }
    int& operator[](int i){
        assert(i >= 0);
        assert(i < m_n);
        return m_data[i];
    }
};
```

RAII = Resource Acquisition Is Initialization

10.7 Controlling copying

```
DynamicArray original{· · ·};

// COPY CONSTRUCTOR: (new object created as copy)
auto copy{original}; // auto copy = original

DynamicArray other{...};
// COPY ASSIGNMENT OPERATOR: (change value of existing obj as copy)
other = original;
```

Per gestire operazioni in caso di risorse da gestire (e.g. allocate dinamicamente):

```
class DynamicArray {
   ...
  public:
   ...
   DynamicArray(DynamicArray const& other) : ... {...} // copy constructor
                // takes object of same class by const reference
   DynamicArray& operator=(DynamicArray const& other) {
                // copy assignment operator: takes const ref, returns ref
                // (*this modified)
     if (this != &other) { // checks for auto-assignment
       ...
     }
     return *this;
   }
   ...
};
```

10.8 Smart pointer

Si comporta come pointer ma gestisce il lifetime dell'oggetto cui punta. Ogni volta che si alloca dinamicamente passare raw pointer a smart p. **il prima possibile!**
Esempio di implementazione:

```
template<typename Pointee>
class SmartPointer {
   Pointee* m_p;
  public:
   explicit SmartPointer(Pointee* p): m_p{p} {}
   ~SmartPointer() { delete m_p; }
   Pointee* operator->() { return m_p; }
   Pointee& operator*() { return *m_p; }
};

class Sample { · · · };

{
  SmartPointer<Sample> sp{new Sample{}};
  sp->add(· · ·);
  (*sp).stats();
} // sp destroyed at the end of the scope!
```

Possibile anche (anzi) utilizzare smart pointers forniti dalla standard tramite

```
#include <memory>
```

- `std::unique_ptr<T>`

```
class Sample { · · · };
```

```
void take(std::unique_ptr<Sample> q); // by value

std::unique_ptr<Sample> p{new Sample{}}; // explicit new
auto p = std::make_unique<Sample>(); // better (*)
auto r = p; // error, non-copyable
take(p); // error, non-copyable
auto r = std::move(p); // ok, movable
take(std::move(r)); // ownership is moved (no copy!)
```

(*) costruisce un nuovo oggetto Sample e restituisce valore puntatore ad esso; argomenti passati a `make_unique` vengono inoltrati al costruttore di Sample!
Consigliato salvo quando necessario:

- `std::shared_ptr<T>` Permette shared ownership (contando anche reference). Per condividere own. con altri `shared_ptr` necessario copy construction o copy assignment: se si fa invece tramite raw pointer sottostante si ha UB!

  ```
  void take(std::shared_ptr<Sample> q); // by value

  std::shared_ptr<Sample> p{new Sample{}}; // explicit new
  auto p = std::make_shared<Sample>(); // better (*)
  auto s = p; // ok, copyable
  take(p); // ok, copyable
  auto s = std::move(p); // ok, movable
  take(std::move(s)); // ok, movable
  ```

Nota 1: sempre possibile passare da `unique` a `shared`, ma non viceversa!
Nota 2: per accedere al raw pointer:

```
<smart>_ptr<T>::get(); // any kind of smart ptr, returns non-owning T*
unique_ptr<T>::release(); // returns owning T* : must be explicitly
                          // managed!!
```

Nota 3: array sono supportati

Passare a funzioni passare smart ptr solo se funzione necessita di utilizzare lo smart pt stesso! Passando `unique` by value possibile trasferire ownership, `shared` by value per 'mantenere in vita' risorsa puntata, by const reference per chiamare metodi / utilizzare il valore dello s.p. stesso.

Returnare se funzione passa al chiamante qualcosa allocato dinamicamente, returnare smart ptr!

10.8.1 Disabilitare operazioni di copia

Per sopprimere le operazioni di copia si segnano il copy constructor e il copy assignment operator con `= delete`:

```
template<typename Pointee>
class UniquePtr {
   Pointee* m_p;
  public:
   explicit UniquePtr(Pointee* p): m_p{p} {}
   ~UniquePtr() { delete m_p; }
   UniquePtr(UniquePtr const&) = delete;
   UniquePtr& operator=(UniquePtr const&) = delete;
   Pointee* operator->() { return m_p; }
   Pointee& operator*() { return *m_p; }
};
```

Nota: il copy constructor, anche con `= delete` resta un costruttore, dunque impedisce lo stesso la generazione del default constructor!
In genere `= delete` può applicarsi a qualsiasi funzione, a patto che sia specificato nella **prima dichiarazione** della funzione in una translation unit. L'utilizzo di una funzione *deleted* causa errore di compilazione

10.9 Special Member Functions

Per ogni classe (in aggiunta al costruttore di default). Si seguono:

- **RULE OF ZERO** se non necessario, non definirne nessuna (generate automaticamente dal compilatore, comportamento dipende da quello dei data members)
- **RULE OF FIVE** se si definisce almeno una, meglio definirle tutte: nel caso possibile anche usare `= default` e `= delete`

```
class MyClass {
   MyClass(MyClass const&); // copy constructor
   MyClass& operator=(MyClass const&); // copy assignment
   MyClass(MyClass&&); // move constructor
   MyClass& operator=(MyClass&&); // move assignment
   ~MyClass(); // destructor
};
```

10.10 Tassonomia Container della STL

Sequence client stabilisce dove posizionare elementi (`array, deque, forward_list, list, vector`)

Associative decide il container:

- **Ordered** ordine sulla base di *key* (funzione)
  coppie key-valore: `map, multimap` // valori ordinati: `set, multiset`
- **Unordered** posizione determinata da *hash* della *key* dell'elemento, ovvero dal valore numerico di una funzione che prende in input la *key*
  `unordered_map, unordered_multimap, unordered_set, unordered_multiset`

10.10.1 Std::list

Dimensione dinamica, layout di memoria **non contiguo**, stabilità degli iteratori **[aggiunta, rimozione e spostamento elementi non invalida iteratori o reference - solo con delete!]**.
Implementata come double-linked list: ogni nodo comprende elemento, puntatore al precedente ed al successivo. Consente dunque iterazione bidirezionale, a differenza di `std::forward_list` (solo in avanti) al prezzo di efficienza di spazio.

```
#include <list>
...
std::list<int> lista = {7,6,8,9};

lista.front(); // primo elemento (TIPO int)
lista.back(); // ultimo el (TIPO int)
lista.begin(); // consueto (TIPO int*)
lista.end(); // consueto (TIPO int*)
lista.empty();
lista.size();
lista.max_size(); // limite per ragioni di sistema o implementazione

lista.erase(<iterator>);
lista.push_back(...);
lista.push_front(...); // inserisce all'inizio
                       // NON INVALIDA ITERATORI
lista.insert(<iterator>, <value>);

lista.sort(); // 6,7,8,9
std::list<int> lista2 = {10,11,13,12};
lista2.sort(); // both must be ordered
lista.merge(lista2); // lista2 becomes empty, lista iterators NOT altered
std::cout << lista 1; // 6 7 8 9 10 11 12 13
                    // for equivalent elements, the one from lista precedes
                    // the other from lista2 !
```

Per iteratori: supporta solo `++it`, `--it` (no random access)

Chapter 11

Static data and functions

Alcuni oggetti possono essere creati fuori da qualsiasi blocco di funzione (incluso `main`), come *globali*. Hanno `static` storage duration, ovvero lifetime corrisponde all'intera durata del programma: sono inizializzati prima della chiamata di `main` e distrutti dopo la fine della sua esecuzione. Sono posizionati nel segmento di memoria indicato con *Static Data*.

- Possono essere inizializzati a valore costante al compile o dinamicamente (e.g. con chiamata di funzione)
- L'ordine di inizializzazione e distruzione è **deterministico** solo nella medesima Translation Unit: in caso di dipendenza tra oggetti in varie TU possibili problemi!
- **Meglio evitare di utilizzarli, specie se non costanti**

Applicazione: definire costanti (e.g. fisiche), possibilmente dentro un namespace che non sia quello globale

```
namespace std::numbers {
  inline constexpr double e = . . . ;
  inline constexpr double pi = . . . ;
  . . .
} // specify inline if, as probable, definitions are in a header file
```

Constexpr garantisce che l'inizializzazione può essere effettuata al compile time. Utilizzato nella dichiarazione di un oggetto o di un metodo non `static` implica `const`, in quella di una funzione o di uno `static data member` implica `inline`.
Una variabile `constexpr` deve essere *LiteralType*, essere immediatamente inizializzata e l'espressione che la inizializza (compresa di chiamate a costruttori e conversioni implicite) deve essere una `constant expression` valutabile al compile time.

11.1 Membri static

(è keyword) Un membro di una classe dichiarato `static` **non è parte** di alcun oggetto della classe. Dunque

- Esiste anche se $\nexists$ oggetti della classe
- Ha static storage duration
- É dichiarato dentro la classe **ma definito fuori**, fatto salvo sia dichiarato `inline` o `constexpr` (vd. sopra per implicazione) o ancora sia `const` integral type.

Per accedervi si utilizza scope operator; tuttavia se sono stati dichiarati oggetti del tipo si può anche usare operatore di accesso ai metodi (`.`).

```
struct X {
  static int n; // declaration
};
// probably in a .cpp file
int X::n = 42; // definition

//Alternativamente:
struct X {
```

```
  ...
  inline static int n = 42; // or constexpr static ...
};

X xx;
std::cout << xx.n; // prints 42
```

I metodi dichiarati `static` possono accedere agli altri membri della classe, **ma solo quelli** `static`. Non possono inoltre essere dichiarati `const` (si applica solo a metodi non `static`)

Sospendere processo

```
#include <chrono>
...
using namespace std::chrono_literals;
std::this_thread::sleep_until(std::chrono::system_clock::now() + 15ms);
// now() è uno static member di tipo std::chrono::time_point
```

Chapter 12

Dynamic polymorphism

Concetto centrale nell'**Object-Oriented Programming**, permette di implementare un'unica interfaccia (polimorfica) per entità di diversi tipi. Dinamico = al runtime!

12.1 Inheritance

Una classe può essere dichiarata `derived` da una o più classi `base` (iterando → gerarchia).

- Membri della classe base lo sono anche delle derivate.
- La costruzione di un oggetto `Derived` comporta (anche solo implicitamente) quella di un sotto-oggetto `Base`
- Puntatori `Derived*` possono essere convertiti implicitamente in `Base*`
- É possibile legare una referenza `Base&` a un oggetto `Derived`

```
struct Base {
  int a;
  Base(int a) : a{a} {}
  void f();
  int operator()() const;
};
struct Derived : Base {
  double d;
  Derived(int i, double d)
    : Base{i+1}, d{d} {}
  int h() const;
};

Derived de{42, 3.14};
de.d;
de.h();
de.a; // Base::a
de.f(); // Base::f
de(); // Base::operator()
Base* b1 = &de;
Base& b2 = de;
```

12.2 Abstract classes & Virtual functions

Permette di realizzare interfaccia comune per classi derivate e loro metodi. **Non esiste di per sè:** non è possibile creare oggetti di tipo `Base`!

```
struct Shape { // abstract base class
  virtual ~Shape(); // virtual destructor, no '= 0' here
  virtual Point where() const = 0; // pure virtual function
};
```

```
struct Circle : Shape { // derived
  Point c;
  int r;
  ~Circle();
  Point where() const override;
};

struct Rectangle : Shape {
  Point ul;
  Point lr;
  ~Rectangle();
  Point where() const override;
};

std::unique_ptr<Shape> create_shape(); // use a smart pointer *
auto s = create_shape();
s->where(); // call redirected to corresponding function at RUNTIME
// * automatically deleted at end of scope
```

Un metodo non `static` è una funzione virtuale se è dichiarata per la prima volta con la keyword o se compie `override` di una funzione virtuale di una classe base. Una classe è **polymorphic** se ha almeno un metodo virtuale. **Funzioni overridden devono avere la stessa segnatura!** altrimenti si ha **HIDING** : sufficiente varino tipi argomenti oppure si aggiunga attributo `const`
Una funzione è **pura** se la dichiarazione termina con `= 0`; salvo eccezione (vd. dopo) **non può essere definita**.
Una classe è **abstract** se ha almeno un metodo virtuale puro; nel caso non ne possono essere creati oggetti.
Possibile anche la classe base non sia astratta:

```
struct Shape {
  Point p;
  Shape(Point p) : p{p} {}
  virtual ~Shape();
  virtual Point where() const { return p; } // not pure, has default implementation
};

struct Circle : Shape {
  int r;
  Circle(Point p, int d) : Shape{p}, r{d} {}
  ~Circle();
  // where() is inherited from Shape
};

struct Rectangle : Shape {
  Point lr;
  Rectangle(Point p1, Point p2) : Shape{p1}, lr{p2} {}
  ~Rectangle();
  Point where() const override { return (p + lr) / 2; }
  // implementata diversamente con override (p ereditato da Shape)
};
```

non consigliato, specie per la gestione delle variabili private: in questo caso, ad esempio, p stesso rappresenta un'interfaccia e non è possibile modificarne il nome.

12.3 Slicing

In generale, per classi base non astratte **è possibile l'istanziazione e la copia di oggetti**. Nel caso è però consigliato di porre i metodi speciali di copy/move = `delete`
Quando si passa un oggetto di classe derivata a una funzione che prende un parametro di classe base **by value**, **VIENE PASSATO SOLO IL SOTTO-OGGETTO DI CLASSE BASE**!
In generale si consiglia comunque di **dichiarare il distruttore come pure virtual** ma di definirlo (in modo che le classi derivate siano distrutte in modo adeguato)

```
struct Shape {
```

```
  Point ul;
  Shape(Point p): p{p} {}
  virtual ~Shape() = 0;
  virtual Point where() const; // non-pure virtual function
};

inline Shape::~Shape() = default; // or any other implementation
```

In generale **Si consiglia di tenere le classi base astratte**

12.4 Final

Una funzione virtuale può avere uno solo degli attributi `virtual, override, final`. Una funzione `final` non può essere overridden in classi derivate, mentre una **classe** `final` non può avere derivate!

```
struct Derived final { · · · };
```

12.5 Access control

Un membro di classe può essere

- `public` nome può essere utilizzato ovunque
- `private` solo dai membri e dai friend della classe
- `protected` solo dai membri, dai friend della classe **e dalle derivate da tale classe, oltre che i loro friend**

Ordine dentro classe: `private:` → `protected:` → `public:`
Anche la derivazione in sè può avere tali attributi

```
class Derived : public|private|protected Base {};
```

Si sconsigliano gli ultimi due! Con derivazione `public`:

- `public` in Base → `public` in Derived
- `protected` in Base → `protected` in Derived

Sub-typing(relazione *is-a* (*è una*))
CONSIGLIO: tenere i data members comunque in `private`!

12.6 Distruttore e copy/move

In caso di inheritance polimorfica, il distruttore di classe base **deve essere**

- `public` & `virtual` oppure
- `protected` & `non-virtual`

Le operazioni di copy/move per la classe base devono essere accessibili alle derivate ma non al public: **si dichiarano** `protected`!

```
class Base
{
   · · ·
  protected:
   Base(Base const&);
   Base& operator=(Base const&);
   Base(Base&&);
   Base& operator=(Base&&); // FIVE RULE!
  public:
   · · ·
};
```

Possibile altrimenti implementare un metodo virtuale di *cloning*

```
class Base
{
  public:
   virtual Base* clone() const = 0;
};

class Derived: public Base
{
  public:
   Derived* clone() const override
   {
     return new Derived{*this}; // call the copy ctor of Derived
   }
};
```

12.7 I/O Streams

La libreria Input/Output della SL è implementata tramite gerarchie di classi e basata sul concetto di *streams*. `std::cin` e `std::cout` sono riferite al terminale, ma altre classi permettono lettura e scrittura da/su file, stringhe, rete (networking).
Grazie alla gerarchia polimorfica oggetti di tali classi possono essere passati a funzioni implementate partendo dalla classe base.
Alias specifici del namespace:

```
namespace std {
  using istream = basic_istream<char>;
  using ostream = basic_ostream<char>;
  using istringstream = basic_istringstream<char>;
  using ostringstream = basic_ostringstream<char>;
  using ifstream = basic_ifstream<char>;
  using ofstream = basic_ofstream<char>;
  . . .
  istream cin;
  ostream cout;
}
```

Leggere e scrivere file

```
#include <fstream>
...
std::ifstream is{"/tmp/in"}; // open /tmp/in for reading
std::ofstream os{"/tmp/out"}; // open /tmp/out for writing
if (!is || !os) { // supporta conversione a booleano: good = true
                  // fail = false (nell'apertura del file)
// manage error
}
double d;
while (is >> d) {
  os << std::setw(12) << d * 2 << '\n';
}
```

Il costruttore e il distruttore di `fstream` aprono e chiudono implicitamente il file. Vi sono anche metodi espliciti `open()` e `close()` (sconsigliati)

Leggere e scrivere stringhe

```
#include <sstream>
...
std::istringstream is{"12. 14. -42."};
std::ostringstream os;
double d;
```

```
while (is >> d) {
os << std::setw(12) << d * 2 << '\n';

std::cout << os.str(); // member func str() allows to get & set string contents
// " 24\n 28\n -84\n"
```

Per stampare su stringa da qualsiasi stream in ingresso:

```
std::getline(<istream>, <string>);
std::getline(<istream>, <string>, <char>); // stampa fino a quando
                                           // incontra <char>
```

Con `./a.out < testo.txt` da shell possibile dare file di testo in input all'eseguibile (ovviamente dopo compilazione).

12.7.1 Operatore minore-minore

`operator<<` implementato come *free function* (oggetto da streammare è il secondo parametro, non l'oggetto di classe `*this` su cui si chiamerebbe altrimenti! - si deve stampare sullo stream)

```
class Complex {
  ...
 public:
  ...
};

inline std::ostream& operator<<(std::ostream& os, Complex const& c)
{
  os << ... ;
  return os;
}
```

In generale, implementato come **template**, accetta e returna `std::basic_ostream<...>`

12.7.2 Friend functions

Free functions implementate in modo da essere esposte in modo adeguato attraverso l'interfaccia pubblica e da accedere ai membri privati di classe. Può essere sia in parte privata che pubblica.

```
class Complex {
   double r; double i;
  public:
   friend std::ostream& operator<<(std::ostream& os, Complex const& c) {
     os << '(' << c.r << ',' << c.i << ')'; // access to private
     return os;
   }
   . . .
};
```

Essendo definite dentro la definizione di classi, le free function sono automaticamente `inline`. Possono inoltre essere solo dichiarate all'interno e definite altrove
Un metodo di una classe può essere `friend` di un'altra. Se ciò riguarda **tutti** i metodi della prima classe, la si può in modo sintetico indicare come `friend` della seconda

Chapter 13

SFML

Stutturazione file di esempio (main!). Si nota la struttura del Game Cycle, che viene eseguito a ogni frame.

```
...
#include <SFML/Graphics.hpp>
#include <iostream>
...

auto to_color(int k)
{
  return k < 256 ? sf::Color{static_cast<sf::Uint8>(10 * k), 0, 0}
                 : sf::Color::Black; // header include anche colori
                                     // di default già implementati
                                     // come static const
}

// NB in header file definitions are inside namespace sf
// here one must use scope operator sf::

int main()
{
  auto const display_width  = 600u;
  auto const display_height = 600u; // display size
  ...
  sf::RenderWindow window(sf::VideoMode(display_width, display_height),
                          "Mandelbrot Set");
  window.setFramerateLimit(60);

  sf::Image image;
  image.create(window.getSize().x, window.getSize().y);
  // getSize() retrieves size of rendering region as a Vector2u
  // (type defined in library!)

  for (auto row = 0u; row != display_height; ++row) {
    for (auto column = 0u; column != display_width; ++column) {
      auto k = mandelbrot(top_left + complex{delta_x * column, delta_y * row});
      image.setPixel(column, row, to_color(k)); // change color of a pixel
    }
  }

  sf::Texture texture;
  texture.loadFromImage(image);
  sf::Sprite sprite;
  sprite.setTexture(texture);

  while (window.isOpen()) { // handles events happening during a frame
    sf::Event event;
    while (window.pollEvent(event)) {
```

```
      if (event.type == sf::Event::Closed) { // gestisce chiusura finestra
        window.close();
      }
    } // NB: tutti gli eventi che avvengono nella durata di un frame
      // (click, etc.) vengono raccolti indiscriminatamente

    window.clear(); // 'stampa' sfondo (colore)

    window.draw(sprite); // da chiamare per ogni forma etc. da visualizzare
                         // ANCHE ITERANDO SU VETTORE DI FORME

    window.display(); // termina frame

    using namespace std::chrono_literals; // possibile limitare framerate
    std::this_thread::sleep_for(15ms);
  }
}
```

Per lavorare con la classe Shape e le sue derivate, CircleShape, RectangleShape, ConvexShape:

```
sf::CircleShape circle;
circle.setFillColor(sf::Color::Blue); // or sf::Color( R,G, B);

circle.setPosition( <float_x>, <float_y>);
circle.setPosition( <const sf::Vector2f& >); // vettore posizione nello schermo

//NB l'asse x è orientato da sx a dx, y DALL'ALTO AL BASSO:
// il punto {0,0} è l'angolo in alto a sx

//ALTI METODI:
circle.setRotation( <angolo float>);
circle.setScale( <float_x>, <float_y>); // analogamente con Vector2f
circle.setOrigini(x, t); // imposta origine locale dell'oggetto

circle.move(x,y); circle.move( vec );
circle.rotate(x);
circle.scale(x,y); circle.scale( vec );

circle.getTransform(); // returns the combined tranform of the object
                       // (all the trans. done together) as const Transform&
circle.getInverseTransform(); // analogo ma per la transform combinata inversa

// PER TUTTI I METODI set E' IMPLEMENTATO ANCHE IL CORRISPONDENTE get
// (return type è l'argomento del set)
```

Le trasformazioni sono ottenute tramite matrici tridimensionali (trasformazioni affini, non solo lineari!), secondo metodi implementati nella classe base Transformable, da cui deriva la citata Shape ma anche Sprite e Text.
Lista colori di default:
Black, White, Red, Green, Blue, Yellow, Magenta, Cyan, Transparent

NOTA BENE: VI SONO MEMORY LEAKS INSITI IN SFML!
NELLA RELAZIONE DEL PROGETTO VANNO INDICATE EVENTUALI SEGNALAZIONI DEL SANITIZER