

---

# Lab di programmazione

---

## Tipo

Definisce una tipologia di contenuto, un range di valori e un insieme di operazioni per un oggetto. Ogni tipo ha il suo formato *literal*.

## Oggetto

Una regione di memoria con un tipo che specifica quale tipo di dato può essere inserito.

## Variabile

Un oggetto con un nome

## Valore

Elemento posto dentro la variabile: un insieme di bit in memoria interpretati in funzione del tipo.

## Type safety

Sono type safe le conversioni che preservano il valore: tutte le conversioni verso un tipo con maggior capacità. Il C++ è fortemente tipizzato, e la type safety deve essere sempre garantita.

`{}` è la notazione di inizializzazione che evita le narrowing conversions e lancia eccezione in caso contrario.

## Lvalue e Rvalue

Un *lvalue* è ciò che sta alla sinistra di un operatore di assegnamento. Ci sono oggetti che assumono il significato di come *lvalue* o *rvalue* a seconda dei contesti.

Un costruttore di copia richiede un *lvalue* come argomento. Una reference anche. Una const reference non richiede un *lvalue*. Il nome di un array è un *rvalue*.

---

## Ordine di valutazione

```
v[i] = ++i;
```

```
v[++i] = i;
```

```
int x = ++i + ++i;
```

```
cout << ++i << ' ' << i << '\n';
```

Tutti ordini di valutazione non definiti.

## Dichiarazione e definizione

*extern* indica che la dichiarazione non è una definizione, è poco usato e poco utile.

Per una variabile, la dichiarazione sopperisce al tipo ma solo la definizione sopperisce all'oggetto (la memoria).

Per una funzione, la dichiarazione sopperisce al tipo ma solo la definizione sopperisce il corpo (il codice eseguibile).

Una reference deve essere necessariamente inizializzata in fase di definizione.

Una variabile *const* deve essere inizializzata.

*A variable is declared when the compiler is informed that a variable exists (and this is its type); it does not allocate the storage for the variable at that point.*

*A variable is defined when the compiler allocates the storage for the variable.*

`vector<char> v` è una definizione (riserva memoria);

Sintassi di inizializzazione: `=`, `()`, `{}`

`new int(10)` crea un nuovo puntatore ad un oggetto intero che contiene il valore 10

`new int[10]` crea un nuovo puntatore ad un vettore di 10 interi

Il meccanismo che garantisce una inizializzazione di default è il costruttore di default.

Per poter inizializzare un vettore con una lista del tipo `{1,2,3...}` possiamo fare:

```
vector(initializer_list<double> lst)
```

```
:sz{lst.size()}, elem{new double[sz]}
```

```
{ copy(lst.begin(),lst.end(),elem) }
```

poi potremo scrivere: `vector v1={1,2,3}` oppure equivalentemente `vector v1{1,2,3}`

Attenzione: `vector v2(3)` inizializza tre elementi al valore `0.0`, `vector v2{3}` inizializza un elemento a valore `3.0`.

La distinzione tra dichiarazione e definizione è utile per poter usare entità prima di fornirne la definizione completa, es. Header. Riflette interfaccia vs implementazione.

## Identifier

Un *Identifier* è una sequenza di caratteri che inizia con una lettera o una “\_” seguita da zero o più lettere, numeri o “\_”.

“\$” non può essere usato come *identifier*. *Identifiers* che iniziano con una underscore o che contengono doppia underscore sono riservati per implementazione: non vanno usati.

## Preprocessore

*Header*: collezione di dichiarazioni. Siccome l'`#include` avviene prima di qualsiasi altra azione del compilatore, gestire gli `#include` è compito del *preprocessore*. Si può includere con `#include "file.h"`, `#include <file.h>`.

Il preprocessore *runs* prima del compilatore e trasforma il codice sorgente in quello che vede il compilatore. Ogni riga di codice che inizi con `#` è una direttiva del preprocessore. In realtà, questa azione è integrata nel compilatore e non particolarmente interessante a meno che non causi eccezioni. Come output genera un file `.i` a memoria, ovvero un file di testo (codice modificato).

Non sono istruzioni le direttive del preprocessore!

## Compilare il codice

```
g++ -o my_exec my_source.cpp -lmy_lib
```

## Scope

*global scope*: area di testo fuori da qualsiasi altro scope

*namespace scope*: scope nidificato in uno scope globale o in un altro namespace

*class scope*: area di testo dentro una classe

*local scope*: all'interno delle parentesi di un blocco o nella lista di argomenti di una funzione

*statement scope*: interno di un *for statement*, *while* *switch* o *if*.

Il principale scopo di uno *scope* è di mantenere i nomi locali, così che non possano interferire con nome dichiarati altrove.

*clash*: due definizioni incompatibili nello stesso *scope*

*member function*: funzione interna ad una classe

*member class*: classe interna ad una classe

*local class*: classe interna ad una funzione (da evitare)

*local functions*: funzioni interne a funzioni (NON LEGALE IN C++)

Il *return* è una forma di inizializzazione.

## Size of

*sizeof(x)*, *x* può essere un tipo o un'espressione. Se è un'espressione, il risultato è la dimensione dell'oggetto risultante, se è un tipo è la dimensione dell'oggetto di quel tipo. La dimensione è misurata in *bytes*.

## Cast

*dynamic\_cast<D\*>(p)*: potrebbe ritornare 0

*dynamic\_cast<D\*>(\*p)*: potrebbe throware *bad\_cast*

Utilizzati per navigare nella gerarchia di classi dove *p* è un puntatore ad una classe base e *D* una classe derivata dalla base.

*static\_cast<T>(v)*: "for reasonably well behaved conversions"

*reinterpret\_cast<T>(V)*: per reinterpretare un pattern di bit

## Passaggio per valore, reference, const reference e puntatore

*Passaggio per valore*: una copia dell'oggetto è passata al nuovo oggetto. Il suo costo è il costo di copia del valore. Lo usiamo per oggetti molto piccoli.

Una *reference* è così chiamata perché si riferisce (*refers*) ad un oggetto definito altrove. Solo quando è necessario modificare un oggetto grande.

Una *const reference* impedisce la modifica dell'oggetto passato. Lo usiamo per oggetti grandi che non dobbiamo modificare.

Una *constexpr*: esprime la nostra necessità di avere calcoli a tempo di compilazione. Una *constexpr* è una funzione normale, tranne quando la si usa dove è richiesta una costante. In questo caso è calcolata a tempo di compilazione solo se i suoi argomenti sono espressioni costanti e dà errore se non lo sono. Per fare in modo che accada, deve avere un corpo consistente di un singolo *return-statement*, ad esempio *constexpr Point scale(Point p) {return {xscale\*p.x,yscale\*p.y}; }*

*Pointer*: usare un puntatore come argomento avverte il programmatore che qualcosa potrebbe essere modificato (perché c'è bisogno di usare &), mentre una reference *looks innocent*. D'altro canto se si usa un puntatore si potrebbe passare come argomento un nullpointer. "*The choice depends on the nature of the function*". Per funzioni dove "nessun oggetto" (rappresentato da *nullptr*) è un argomento valido meglio usare un puntatore. Altrimenti, si usi una reference.

## La memoria

Quando startiamo un programma, il compilatore mette da parte memoria per il nostro codice (*code/text storage*), per le variabili globali che definiamo (*static storage*), per le variabili locali e gli argomenti delle funzioni (*stack storage*), per l'allocazione dinamica della memoria (*free store*).

Lo *stack* è anche detto *automatic storage*: variabili definite in funzioni vi sono inserite a meno che non siano esplicitamente dichiarate *static*. Allocatedo quando una funzione è chiamata e deallocatedo quando la funzione ritorna.

Variabili dichiarate nello scope globale o nello scope namespace si trovano nello stato storage. Il linker allora storage statico prima che il programma sia eseguito. Gli oggetti sono costruiti prima dell'esecuzione del main e distrutti dopo.

Oggetti creati con new sono allocati nel free store.

Ogni volta che chiamiamo una funzione si attiva lo *stack of activation records* (anche chiamato *stack*).

- Text segment – AKA code segment / text – Copiato in memoria dal file oggetto – Contiene le istruzioni eseguibili – Spesso read-only

- Initialized data segment – AKA data segment – Contiene variabili globali e variabili statiche – Read/write • Può contenere una parte read-only

- Uninitialized data segment – AKA BSS segment (Block Started by Symbol, ragioni storiche) – Inizializzato a 0 dal SisOp – Contiene variabili globali e statiche non inizializzate esplicitamente

- Stack – Contiene i RA delle funzioni • Quindi le variabili locali e i parametri

Heap – Formalmente è spesso visto come uno spazio che cresce in verso opposto allo stack  
 – La sua dimensione può essere modificata con le funzioni di sistema `brk` e `sbrk` – Può essere gestito usando blocchi di memoria non contigui (funzione di sistema `mmap`) – Gestito tramite `new` e `delete` • Allocazione dinamica della memoria!

## Variabili globali

Le variabili globali sono inizializzate prima che la prima riga di codice del `main()` sia eseguita. Vivono finché il programma non termina, poi sono distrutte. Sono costruite nell'ordine in cui sono definite e distrutte in ordine inverso.

La ragione principale per cui non le vogliamo è che è difficile sapere quale funzione le modifica. Non hanno nessun tipo di protezione/incapsulamento. Secondo Stroustrup: *a good program will have only very few (say, one or two), if any, global variables*. Un altro problema è che l'ordine di inizializzazione delle variabili globali in file diversi non è definito.

Sono statiche. Le variabili statiche sono inizializzate solo la prima volta, mantengono il valore anche al di fuori del loro scope, possono essere globali o locali (globali sempre static, locali static se così dichiarate).

Le globali hanno la memoria scritta a 0 prima delle inizializzazioni.

## Variabili locali automatiche

Variabili locali automatiche – A ogni chiamata di `f()`, si costruisce `s` • `s` inizializzata alla stringa vuota • `s` esiste fino all'uscita da `f()` – A ogni iterazione del loop, si costruiscono `stripped` e `not_letters` • Esistono fino alla fine del loop • Distrutte in ordine inverso alla creazione

## Namespace

Un *namespace* è un raggruppamento di dichiarazioni (classi, funzioni, dati e tipi). Un nome composto dal nome di un *namespace* (o il nome di una classe) e un nome di un membro combinato con `::` è detto *fully qualified name*.

Se in una classe importo due namespace che possiedono due membri con lo stesso nome si ha un errore:

*Reference to 'cout' is ambiguous // candidate found by name lookup is 'B::cout' // candidate found by name lookup is 'std::cout'*

Corretto definirlo come `namespace name{}` NON con `namespace name{;`

## Tipi built-in e UDT

Vector, string e ostream sono considerati UDT.

Un tipo è detto *built-in* se il compilatore sa come rappresentare oggetti di quel tipo e che operazioni possono essere fatte tra essi senza che il programmatore debba dare direttive a tal proposito nel *source code*.

Tipi non *built-in* sono detti *user-defined types* (UDT). Ci sono due modi per definire un nuovo tipo: come classe o come *enum*.

Una classe è un tipo definito dall'utente che specifica come gli oggetti di quel tipo possono essere rappresentati, creati, usati e distrutti.

Le parti usate per definire una classe sono detti *members*. Accediamo ai membri usando la notazione *object.member*.

Una funzione membro che ha lo stesso nome della sua classe è speciale: è detta *costruttore*.

Il valore di un oggetto è spesso detto *state*, e l'idea di un valore valido è spesso detta *valid state*. Una regola per ciò che fa di un valore un valore valido è detta *invariant*.

Una *initializer list* è una notazione con cui inizializziamo i membri: `:y(yy), m{mm}, d{dd} {...}`

Una funzione la cui definizione è contenuta nella dichiarazione della classe è detta *inline*.

## Enum

Un *enum* (*enumeration*) è un tipo definito dall'utente che specifica l'insieme dei suoi valori (*enumerators*) come costanti simboliche. Es. `enum class Month {jan = 1,...};`

Un *plain enum* esportano i loro valori allo scope dell'*enumeration* e consentono conversioni a *int*.

Può contenere membri private?

Un *plain enum* si converte implicitamente a un intero, ma non viceversa.

Una *Enum class* non si converte ad un intero, e non vale il viceversa.

## Funzione statica

Una funzione *static* è creata solo una volta.

## Non-member function

Una funzione che può essere semplicemente ed efficacemente implementata da sola dovrebbe essere implementata all'esterno di una classe. Queste sono dette *non-member functions*.

Una *helper function* è una *non-member function* che è una funzione ausiliaria alla classe, prendendo argomenti da essa.

## Costruttore di copia

per esempio del tipo `vector(const vector&)`

```
vector::vector(const vector& arg)
    :sz{arg.sz}, elem{new double[arg.sz]}
{ copy(arg,arg+sz,elem); }
```

**Costruttore di copia:** è un costruttore per inizializzare un oggetto con i valori di un altro oggetto. Corretto con reference non con puntatore: non vogliamo fare in modo che si possa modificare l'oggetto. Richiede un *lvalue come* argomento. Non è obbligatorio definirlo se si vuole solo una copia membro a membro tra due oggetti della stessa classe.

possiamo scrivere sia `vector v2=v` sia `vector v2{v}`, sono analoghe;

## Assegnamento di copia

Consiste nell'overloading dell'operatore =

```
vector& vector::operator=(const vector& arg)
{
    double* p = new double[a.sz];
    copy(a.elem,a.elem+a.sz,elem);
    delete[] elem;
    elem=p;
    sz=a.sz;
    return *this;
}
```

questo assegnamento di copia gestisce anche l'auto-assegnamento.



Costruttore di copia e assegnamento di copia sono forniti di default dal compilatore se non definiti dall'utente. Il significato sarà *memberwise copy*.

*Shallow copy* è dannosa per gli oggetti ma non per i tipi built-in.

I tipi che supportano shallow-copy sono detti avere una *pointer-semantics* o *reference semantics*, mentre quelli che supportano deep-copy sono detti avere una *value-semantics*.

Assegnamento della reference: copia dell'oggetto a cui si riferisce (deep copy)

## Move constructor

Le operazioni di move complementano quelle di copia, e servono a spostare la proprietà di risorse da un oggetto a un altro e a muovere un oggetto da uno scope a un altro.

con un costruttore di move (mai costante!)

```
vector::vector(vector&& arg)
    :sz{arg.sz}, elem{arg.elem}
```

```
{ arg.sz = 0; arg.elem=nullptr; }
```

questo si chiamerà **move constructor**, richiede un *rvalue* come argomento

```
vector& vector::operator=(vector&& arg)
```

```
{
    delete[] elem;
    elem = arg.elem;
    sz = arg.sz;
    arg.elem=nullptr;
    a.sz=0;
    return *this;
}
```

Di default, il compilatore genererà operazioni di move di default. Il significato sarà *memberwise move*.

## Costruttore e distruttore

Una classe ha bisogno di un distruttore se acquisisce risorse, per esempio la memoria dal *free store*. Una classe che ha un distruttore quasi certamente ha bisogno anche di tutte le operazioni di *copy* e di *move*, perché il significato originale di copia e di movimento sono quasi certamente sbagliati.

Un distruttore è chiamato quando un oggetto di una classe è distrutto. Avviene quando si esce dallo scope, il programma termina, o *delete* è usato su un puntatore a un oggetto.

Un costruttore è chiamato quando un oggetto della sua classe è creato: quando una variabile è inizializzata, un oggetto è creato usando *new*, quando un oggetto è copiato.

Un costruttore che accetta un singolo argomento definisce una conversione dal tipo del suo argomento alla classe. *vector v = 10* inizializza un vettore di 10 double, *v = 20* assegna un nuovo vettore di 20 double a *v*. Un costruttore definito *explicit* consente solo la normale costruzione semantica e non le conversioni implicite.

Una classe base generalmente ha bisogno di un distruttore virtual.

## Overloading di []

Overloading di [] in vector: *double& operator[](int n)*, *double operator[](int n) const*. Uno non *const* e uno *const*: il primo deve permettere la scrittura e quindi restituisce reference, il secondo non deve permettere la scrittura e quindi restituisce per valore. Avremmo potuto anche fare *const double& operator[](int n)* (non l'abbiamo fatto solo perché l'oggetto è piccolo).

## Come il programma elimina un oggetto allocato dinamicamente

L'entità che gestisce la dimensione dell'array passato a *delete[]* non è il sistema operativo con il suo sistema di gestione della memoria, bensì il codice generato dal compilatore. Quindi è il compilatore che crea le strutture che tengono traccia delle dimensioni dei blocchi di memoria allocati dinamicamente, e il relativo codice macchina è parte integrante dell'eseguibile che ottenete post compilazione e linking. Questo meccanismo può essere implementato in vari modi (dipende dal compilatore):

- *Over-allocation*: il numero di elementi dell'array è inserito prima del primo elemento dell'array. Quando chiamiamo *delete[] p* si genera uno pseudocodice del tipo:

```
// Get the number of elements in an array
size_t n = * (size_t*) ((char*)p - sizeof(size_t));

// Call the destructor for each of them
while (n-- != 0)
{
    p[n].~SomeClass();
}

// And finally cleaning up the memory
operator delete[] ((char*)p - sizeof(size_t));
```

Lo svantaggio è che se si chiama il *delete* in modo scorretto il programma crasherà con "Heap Corrupt" ed è molto difficile risalire alla causa del crash.

- *Associative array*: si ha il richiamo ad un contenitore globale nascosto che contiene il puntatore all'array e agli elementi che contiene. In questo caso non c'è nessun dato nascosto prima degli elementi dell'array.

```
// Getting the size of an array from the hidden global storage
size_t n = arrayLengthAssociation.lookup(p);

// Calling destructors for each element
while (n-- != 0)
{
    p[n].~SomeClass();
}

// Cleaning up the memory
operator delete[] (p);
```

Lo svantaggio è che cercare nello storage globale porta un rallentamento del programma. Ma compensa il fatto che il programma potrebbe essere più tollerante con la scelta del *delete* sbagliato.

## Parametri vs Argomenti

L'argomento di una funzione è il valore effettivo fornito a una funzione, mentre il parametro è la variabile all'interno della definizione della funzione. Un argomento passato a una funzione è convertito nel tipo del parametro.

## std::move

Equivalente ad uno *static\_cast* ad una *rvalue reference*. Utile quando passiamo un argomento ad una funzione e non vogliamo che il rispettivo parametro sia una copia ma una *move*.

Es. `void g(Object that_will_be_created_with_a_move) //the parameter won't be a copy`  
`g(std::move(obj));`

Non funziona (e in tal caso non si ha errore ma una copia più lenta del solito) se:

- è chiamato su un oggetto che non ha *move constructor*;
- è chiamato con un argomento *const*;
- quando il valore ritornato (nell'esempio il parametro della funzione) dev'essere un argomento *const reference* (`void g(const Object &that_wont_be_created_with_a_move)`).

L'idea di non effettuare nessuna copia nella costruzione dell'oggetto a partire da un argomento fornito al costruttore è irrealizzabile (a meno di particolari condizioni al contorno), perché la sorgente di questa copia è quasi sempre un oggetto locale automatico (o qualcosa di simile) che esiste nel contesto in cui il costruttore è chiamato. Tale oggetto non è movabile e non può essere spostato dentro l'oggetto che stiamo costruendo.

## Allocazione dinamica nell' *initialization list*?

L'*initialization list* andrebbe usata solo per copiare valori, spostando le allocazioni dinamiche all'interno del corpo della funzione.

La percezione di Ghidoni dell'*initialization list* è quella di una sezione di codice dove avvengono operazioni semplici e veloci, mentre un'allocazione dinamica è un'operazione piuttosto onerosa. Non c'è riscontro in fonti autorevoli: è una sua visione.

## *Initialization list* - ordine di creazione dei membri

I *data member* vengono inizializzati nell'ordine con cui sono stati dichiarati nella classe. Se dovessi inizializzare un *data member* tramite un altro *data member* che è posto dopo nell'ordine di dichiarazione risulterebbe errore.

## Aritmetica dei puntatori

Un puntatore può puntare ad un elemento di un array, e possiamo dereferenziare quel puntatore incrementandolo o meno: `double *p = &ad[5], p[2] = 6, *p = 7, p+=2` (si sposta due elementi avanti). Questa è detta aritmetica dei puntatori. Molto dannosa, ma non removibile per ragioni storiche.

Non possiamo assegnare qualcosa ad un nome di array: il puntatore al primo elemento (il nome dell'array) è un valore, non una variabile. `ac = new char[20]` da errore. Non si può neanche copiare un array tramite assegnamento.

`char ac[] = "Beorn"` (array di 6 elementi con 0 finale), `int ai[] = {1,2,3,4,5,6}`, `int ai2[100] = {1,2,3,4,5,6}` (gli altri 94 inizializzati a 0), `double ad[100] = {}` (tutti gli elementi inizializzati a 0), `char chars[] = {'a','b','c'}` (no 0 alla fine)

Mai accedere attraverso un *delete pointer*.

## Void\*

`void*` salta qualsiasi controllo: è un puntatore a memoria raw, devo solo fornire indicazioni su come dev'essere. Possiamo assegnargli qualsiasi puntatore. Rappresenta il concetto puro di indirizzo di memoria senza indicazioni su come usarla.

Le uniche operazioni consentite sono copia (assegnamento o inizializzazione), cast e comparazione (`==`, `!=`, `<`, `>`, `<=`, `>=`).

## Default arguments

Argomento di default per gestire il caso in cui una chiamata alla funzione non specifichi un argomento.

`void f(int, int=0, int=0) -> f(1,2), f(1) legali`

`f(1,,1) non legale`

## Unspecified arguments

`void printf(const char* format...);` // significa argomento `const char*` e forse più argomenti...

## Overloading

Nell'overloading degli operatori deve essere presente almeno un argomento UDT (user defined Type): non è possibile definire `int operator+(int, int)`.

Non è possibile cambiare il significato degli operatori per i tipi built-in o introdurre nuovi operatori. Tutti gli operatori possono essere definiti tranne i seguenti:

`?: . .* :: sizeof typeid alignas noexcept`

Si possono overloadare sia `"*" che ","`

Funzioni che definiscono i seguenti operatori devono essere membri di una classe:

`= [] () ->`

`Vector operator+=(const Vector&, int);` è invece legale

– `operator+`, `operator-`, `operator*`, `operator/`: due argomenti • Almeno uno UDT, l'altro può essere anche built-in

– `operator <<`: un argomento `ostream&`, ritorna `ostream&`

– `operator >>`: un argomento `istream&`, ritorna `istream&`

• Operatori implementati con funzione membro – Un argomento in meno

Nell'overloading di data member il primo argomento è sostituito dall'oggetto su cui è invocata la funzione. Lo UDT deve essere il primo argomento `int + T` non è accettato. Lo è solo `T + int`. Meno flessibile!

`T& operator++(T& t); // preincremento T operator++(T& t, int); // postincremento:  
argomento //int dummy`

In C++ è consentito il sovraccarico di operatori unari e binari come friend functions.

Un operatore unario esegue un'azione su un singolo operando, un binario su due e così via.

## Friend

In alcune circostanze, è utile che una classe **conceda l'accesso a livello di membro a funzioni che non sono membri** della classe **o a tutti i membri di una classe separata**. Queste funzioni e classi libere sono contrassegnati dalla parola chiave **friend**.

Una funzione o una classe non può dichiararsi amica di nessuna classe. Nella definizione di una classe, si può usare la parola chiave friend e il nome di una funzione non membro o di un'altra classe per garantirle l'accesso ai membri privati e protetti della classe. Nella definizione di un modello, un parametro di tipo può essere dichiarato come amico.

L'esempio seguente mostra una classe Point e una funzione amica, ChangePrivate. La funzione amica ha accesso ai dati privati dell'oggetto Point che riceve come parametro.

Una funzione dichiarate *friend* non può essere anche dichiara *virtual*.

Una funzione che non è membro di una classe può avere accesso garantito a tutti i membri grazie alla dichiarazione *friend*.

## Explicit

Finché non dichiarato explicit, un costruttore che accetta un singolo argomento definisce una conversione implicita dal tipo del suo argomento alla classe.

## Default generated operations - constructor, move, copy, destructor

Finché una classe non ha membri o basi che richiedono argomenti explicit o finché non ha altri costruttori (basta definirne anche solo uno e il default non sarà generato automaticamente!), un costruttore di default è generato automaticamente. Questo costruttore di default inizializza ogni base o ogni membro che ha un costruttore di default (lasciando i membri senza costruttori di default non inizializzati, come ad esempio *int*).

Si hanno anche distruttore, operazioni di copia e di move di default. Sono applicate ricorsivamente dalla classe base ai membri.

```
struct D : B1,B2 { M1 m1; M2 m2; };
```

Per inizializzazione, copia e move l'ordine di costruzione è da base a membri.

L'inizializzazione di default di *d* invoca in ordine: *B1::B1()*, *B2::B2()*, *M1::M1()*, *M2::M2()*

Se uno di questi non esiste o non può essere chiamato, la costruzione di *d* fallisce.

Analogo è ciò che avviene se viene chiamata una copia o una move.

Per il distruttore l'ordine è da membri a base.

Da distruzione *d* invoca in ordine: *M2::M2()*, *M1::M1()*, *B2::B2()*, *B1::B1()*

## Virtual functions

Quando una funzione virtuale è chiamata, la funzione invocata dalla chiamata sarà quella definita per la classe più derivata dell'oggetto a cui è riferita.

## Pure virtual function

Una funzione per cui l'*overriding* è necessario. Es. *virtual void draw() = 0;*

## Override

L'override di funzioni virtuali è possibile renderlo esplicito attraverso il suffisso *override*.

## Namespace

Riunisce dichiarazioni tra loro al fine di prevenire *clashes* di nomi. Tutti i nomi di un namespace possono essere resi accessibili grazie alla direttiva *using namespace xxx*.

## Aliases

Possiamo creare l'alias di un nome, ovvero un nome simbolico che significhi esattamente quello a cui si riferisce.

```
using Pint = int*; //Pint means pointer to int
```

```
namespace Lib = Long_library_name; //Lib means Long_library_name
```

Una reference è un alias, con la differenza che la reference funziona a tempo di esecuzione (*run-time*), mentre un namespace alias funziona a tempo di compilazione.

Codice vecchio usa la keyword *typedef* piuttosto che *using* per definire un alias di tipo:

```
typedef char* Pchar; //same as using Pchar = char*;
```

## Array

Array non è utilizzabile per la copia. `int z[100] = y;` è sbagliato

`char chars[] = { 'a', 'b', 'c' }; //nessuno 0 terminatore`

## Polimorfismo

Polimorfismo: abilità di associare comportamenti specifici diversi a un'unica notazione.

- Polimorfismo dinamico: implementato tramite ereditarietà e funzioni virtuali. A tempo di esecuzione (run-time)
- Polimorfismo statico: implementato tramite template. A tempo di compilazione (compile-time)

Il polimorfismo statico è collegato al concetto di programmazione generica.

Visione semplicistica: programmazione generica in C++ equivale a usare i template, ma così confondiamo un concetto con uno strumento del linguaggio. Più correttamente: programmazione generica significa scrivere codice che funziona con tipi diversi forniti come argomenti, posto che tali tipi soddisfino determinati requisiti sintattici e semantici.

## Reference

Una reference è il nome alternativo sia di un array che di una variabile

Assegnamento della reference: copia dell'oggetto a cui si riferisce (deep copy)

Non esiste una reference non valida

Non si possono creare array di riferimenti

I riferimenti non sono variabili che contengono l'indirizzo di altre variabili perché non sono variabili.

## Lambda expression

Una lambda expression è una tecnica che consente di scrivere una funzione inline senza nome.

Le lambda expressions possono catturare variabili locali per usarle al suo interno.

Se non definiamo cattura, ovvero teniamo [], nessuna variabile sarà catturata, e se proviamo ad usare una variabile all'interno della lambda expression avremo errore in



compilazione: *Variable 'a' cannot be implicitly captured in a lambda with no capture-default specified; error: 'a' is not captured.*

L'operator () di overloading delle lambda expressions è const di default (non può modificare le variabili locali catturate, a meno che queste non siano passate per riferimento).

- se passate per riferimento non ci sarà nessun errore, manda in stampa il valore aggiornato della variabile se richiesto;
- se passate per valore manda errore in compilazione (*Cannot assign to a variable captured by copy in a non-mutable lambda; error: assignment of read-only variable 'a'*)

Tuttavia, se definiamo la lambda expressions come mutable siamo in grado di rendere l'operatore di overloading non const. Potrà quindi modificare le variabili locali:

- se passate per riferimento si comporterà come se non avesse mutable (nessun errore, manda in stampa il valore aggiornato della variabile se richiesto)
- se passate per valore non manderà nessun errore e potrà modificarle ma solo al suo interno: essendo per valore la variabile interna alla lambda expression è una copia, non l'oggetto originale.

## Pre e post incremento

**Esistono pre e post incremento in c++.** Semplicemente per quanto riguarda i cicli *for* è indifferente l'uso dell'uno oppure dell'altro.

## Librerie

Header di sistema inclusi con `<>`, header definiti dall'utente inclusi con `""`

Progetti grandi prevedono molte dichiarazioni, molte definizioni, e per non avere file di eccessive dimensioni (in termini di righe di codice) siamo portati a creare header e avere più di un file sorgente. Questo dalla prospettiva del progettista. Per compilare un progetto composto da molti file avremo bisogno di preprocessore, compilatore e linker.

Il compilatore traduce il codice in codice macchina ("C++ source code -> ""Object code""), ma i diversi file di un progetto possono essere uniti (link) tra loro solo grazie al linker, che fa in modo di creare un file eseguibile. Il compilatore non corregge gli errori, ma li segnala.

*hello\_world.cpp -> compilatore -> hello\_world.obj -> linker -> hello\_world.exe*

*A library is simply some code - usually written by others - that we access using declarations found in a #included file. A declaration is a program statement specifying how a piece of code can be used;*

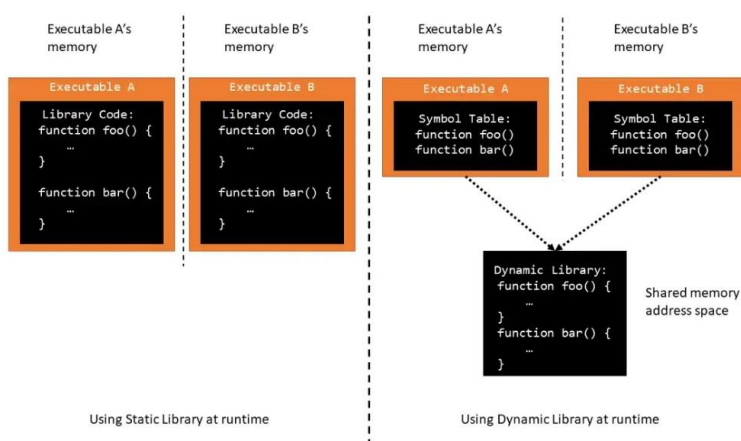
Se abbiamo incluso una libreria in un file, il linker dovrà fare in modo di creare un eseguibile che tenga conto delle proprietà della libreria inclusa nel file necessarie al suo funzionamento.

Una libreria statica contiene codice che è linkato al programma dell'utente a tempo di compilazione. L'eseguibile generato tiene copia del proprio codice della libreria. Genera quindi eseguibili che non possono essere spezzati in seguito (sono più adatte alla distribuzione di software monolitico).

Hanno una buona portabilità: il programma contiene tutto ciò di cui ha bisogno in un singolo eseguibile. Una libreria static gets merged dal linker nell'eseguibile finale durante il processo di linking.

Una libreria dinamica contiene dati binari (poiché file oggetto) come le statiche ma è destinata ad essere linkata all'eseguibile principale, *rather than be merged into it*. The linker creates a special connection between functions and variables used in the main executable and their actual implementations provided by the dynamic library. L'eseguibile finale è più piccolo, e non contiene l'effettivo codice della libreria. Grazie a questo molti programmi possono riferirsi alla stessa libreria senza il bisogno di avere ognuno una sua copia.

Una libreria dinamica riduce lo spazio occupato (una sola copia funziona per tutti gli eseguibili) e può essere ricompilata senza toccare gli eseguibili. Contiene codice che può essere condiviso da più programmi. Il contenuto della libreria è caricato nella memoria a tempo di esecuzione. Ogni eseguibile non mantiene la propria copia della libreria.



## Template

Il compilatore genera la classe con il tipo effettivo che verrà usato al posto del parametro template.

I template lo strumento che consente di realizzare il concetto di programmazione generica in C++. Programmazione generica significa scrivere codice che funzioni con una varietà di tipi presentati come argomenti, a patto che questi argomenti rispettino specifici requisiti sintattici e semantici.

Secondo la notazione Vandevorde/Josuttis il polimorfismo è l'abilità di associare comportamenti specifici diversi a un'unica notazione. Il polimorfismo statico è implementato tramite template a tempo di compilazione.

Il principale problema dei template è che la flessibilità e la performance avvengono a discapito della poca separazione tra l'interno del template (la sua definizione) e l'interfaccia (la dichiarazione).

Un argomento template deve rispettare una serie di requisiti propri del template al quale è associato, che possiamo chiamare element:

*template<typename T> requires Element<T>() class vector* in C++14

*template<Element T> class vector* in C++11

Una class template può essere specificata per tipi diversi. Una delle più comuni è la seguente:

*template<typename T, int N> struct array { T elem[N]; }*

Potremmo applicarla per esempio sugli array:

*array<int,256> gb = {1,5,6,7};*

La dimensione di un array è conosciuta a tempo di compilazione, così il compilatore può allocare memoria statica (per variabili globali) e memoria stack (per oggetti locali) piuttosto che usare il free store. In alcuni programmi, array ci dà tanti dei vantaggi di un vettore senza dover per forza usare il free store.

Per esempio, un array di dimensione non modificabile può esprimere un concetto, per esempio una tripletta di valori (R,G,B) esprimibile come *array<unsigned char, 3>*

Se per una classe template si specificano gli argomenti del template quando si crea un oggetto per una classe specifica, per una function template il compilatore di solito deduce l'argomento del template dagli argomenti passati alla funzione:

*template<class T, int N> void fill(array<T,N>& b, const T& val) {...}*

Possiamo tranquillamente chiamarla come *fill(gb,0)* al posto che *fill<int,256>(gb,0);*

La libreria standard ha una classe *allocator*, che gestisce la memoria inizializzata, quindi per risolvere il problema della distruzione di vettori metà pieni e metà vuoti possiamo fornire a *vector* un parametro *allocator*: `template<typename T, typename A = allocator<T>> class vector{ A alloc; }`. L'unico codice che risente di ciò sono le funzioni membro che interagiscono direttamente con la memoria, come `vector<T>::reserve()`. Al suo interno uso i distruttori di *allocator*: `alloc.destroy`, `alloc.deallocate`.

## Eccezioni e stack unwinding

Quando un'eccezione è lanciata, *the runtime support system in C++* cerca *up the call stack* per un catch adeguato all'eccezione lanciata: guarda all'indietro tutte le chiamate a funzioni che hanno originato il throw, finché non trova un match. Se non lo trova, il programma termina. In ogni funzione incontrata durante questa ricerca, sono chiamati i distruttori a liberare le risorse acquisite. Questo processo è detto *stack unwinding*.

## Gestione risorse e Smart pointers - unique e shared ptrs

N.B: allocare dinamicamente un vector è solitamente dannoso: sarà usato negli esempi perché il riempimento potrebbe causare a sua volta eccezioni.

Un problema che si presenta spesso con le eccezioni è che un'eccezione potrebbe essere lanciata prima di aver liberato le risorse acquisite da un oggetto (ovvero prima dell'istruzione di *delete*). La soluzione è acquisire una risorsa in un costruttore per un oggetto che la può gestire, e liberarla nel corrispettivo distruttore: questa tecnica è detta RAII. Meglio sempre usare un vettore piuttosto che espliciti *new* o *delete* quando c'è bisogno di "a nonconstant amount of storage within a scope".

Ma se invece questo vettore dovesse essere ritornato? Dal compilatore sarebbe distrutto subito dopo il *return*, ma se non ci fosse tempo di raggiungere il *return* che un'eccezione venisse lanciata?

```
vector<int>* make_vec()
{
    vector<int>* p = new vector<int>;
    try{ (...) return p}
    catch(...) { delete p; throw; }
}
```

*Basic guarantee*: uso di *try...catch* per fare in modo che la funzione abbia successo o lanci un'eccezione senza aver avuto *memory leak* (*leak any resources*).

*Strong guarantee*: se oltre alla *basic guarantee* la funzione fa in modo che tutti i valori non locali alla funzione siano gli stessi dopo l'eccezione di quello che erano alla chiamata della funzione. O la funzione ha successo o un'eccezione è chiamata senza che altro accada o si modifichi.

*No throw guarantee*: la funzione non lancia eccezioni.

Il blocco di *try...catch* è ancora brutto però: dobbiamo usare la RAI. Avremmo bisogno di un oggetto *to hold* quel `vector<int>` in modo che possa operare una *delete* quando viene lanciata l'eccezione. In `<memory>`, la libreria standard mette a disposizione *unique\_ptr*.

Uno *unique\_ptr* è un oggetto che detiene un puntatore. Possiamo immaginarlo come una sorta di puntatore: si può dereferenziare usando `->` o `*` esattamente come un puntatore *built-in*. Quando distrutto, distrugge anche l'oggetto a cui punta.

Attenzione: se dereferenzio lo *unique\_ptr* non ottengo l'accesso ai membro dell'oggetto, bensì l'accesso all'oggetto!

```
vector<int>* make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    return p.release();
}
```

Il comando *p.release()* estrae il puntatore da *p* in modo che possa essere ritornato, e fa anche in modo che *p* possieda ora un *nullptr* così che se distruggiamo *p* non si distrugge nessuna informazione utile.

L'unico problema che rimane ora è che si ritorna comunque un puntatore, che in qualche modo dev'essere distrutto. Possiamo quindi tornare uno *unique\_ptr*:

```
unique_ptr<vector<int>> make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    return p;
}
```

Lo *unique\_ptr* ha una restrizione: non può essere assegnato ad un altro *unique\_ptr* in modo da averne due che detengano lo stesso oggetto. Se si vuole avere uno "smart" pointer che garantisca i vantaggi di *unique\_ptr* e possa essere copiato, basta usare uno *shared\_ptr*. Due

*shared\_ptr* allo stesso oggetto condivideranno l'accesso a quell'oggetto (se modifichi uno, modifichi anche l'altro).

Per uno *shared\_ptr* la memoria è deallocata quando l'ultimo *shared pointer* che detiene la memoria è distrutto. Questo meccanismo consuma risorse.

Uno *unique\_ptr* viene definito come *smart pointer*: dealloca la memoria uscendo dallo scope, permette i move, non permette le copie, non ha sostanziali *overhead* rispetto a un puntatore.

Ma come siamo sicuri che nei *return* avvengano i debiti *delete*?

Quando abbiamo aggiunto l'operazione di *move* a *vector*, abbiamo risolto quel problema: il costruttore di *move* trasferisce efficacemente le risorse di un *return*.

```
vector<int> make_vec()
{
    vector<int> res;
    return res;
}
```

...forse la versione migliore. Semplice e efficace.

## Altro

set e map riordinano secondo criterio di interi, double o lessicografico gli elementi inseriti al loro interno (se oggetti deve essere specificato il criterio). Set riordina gli oggetti stessi secondo il criterio, map riordina le chiavi.

Nel caso di set di interi, l'iteratore ad un set vuoto ritorna 0.

Tramite la funzione copy è possibile copiare array, list e vettori

i function object consentono il passaggio di valori al predicato evitando l'uso di variabili globali, se implementati correttamente, i function object sono più efficienti di una chiamata a funzione, essendo degli oggetti, i function object possono essere utilizzati come parametri di un template

- allows *body* to modify the objects captured by copy, and to call their non-const member functions

Il valore a cui fa riferimento un iteratore è ottenuto con l'operatore `*`, non con `[]`.

Un iteratore è un componente di STL utilizzato per puntare ad un indirizzo di memoria di un container.

*iterator it = v.begin()* è un'istruzione sbagliata, è corretta invece *auto it = v.begin()*.

`std::vector` supporta *insert()* ed *erase()*, anche se sono inefficienti perché devono muovere molti elementi in memoria. Può fornire range check.

`std::string` è come i vettori, ma aggiunge le operazioni di manipolazione del testo (+, +=).

`char[]` al contrario è un array stile C, quindi non possiede iteratori, range check e i confronti (`==`, `!=` e `<<`) si riferiscono al puntatore al primo elemento, non al contenuto.

`std::list` non supporta l'operatore `[]`, possiamo usare *insert()* ed *erase()*

Per catturare ogni tipo di eccezioni in C++ è usata l'espressione *catch(...)*

La parola chiave utilizzata per controllare la presenza di un'eccezione in un blocco di testo è *try*.

*No-throw guarantee* e *Basic guarantee* possono andare a braccetto, così come *Basic guarantee* e *Strong guarantee*.

Un *dangling pointer* è un puntatore la cui memoria è stata deallocata.

Non si possono avere costruttori virtuali in C++

Conversioni da classe derivata a classe base non sono supportate implicitamente.


*Derived \*bd = new Base;* da errore perché *derived* potrebbe che so avere un raggio

Una funzione membro di una classe base può essere disabilitata tramite `=delete`.

Lo *slicing* avviene quando cerco di usare un contenitore di oggetti di una classe derivata come contenitore di oggetti della classe base: non sai cosa succede se metti un *Circle* con un raggio in una *Shape* che non ha raggio. Equivalente del troncamento degli interi.

The C++ standard library provides a framework for dealing with data as sequences of elements, called the STL.

Quando dobbiamo gestire dei loop di una struttura generica possiamo usare *auto*: usa il tipo dell'iteratore come il tipo della variabile.



Disaccoppiamento lato algoritmo: non è necessario conoscere i dettagli della struttura dati; è sufficiente usare i relativi iteratori (Es: un algoritmo di ordinamento può operare pur senza conoscere i dettagli della struttura dati)

Disaccoppiamento lato struttura dati: non è necessario esporre tutti i dettagli e gestire tutti i modi d'uso; è sufficiente implementare ed esporre gli iteratori (Es: un vector non ha necessità di conoscere l'algoritmo che opera su di esso per permettergli di funzionare; è sufficiente che esponga gli iteratori)

If you feel an urge to become a language lawyer, study the layout and triviality concepts in the standard (§iso.3.9, §iso.9) and try to think about their implications to programmers and compiler writers. Doing so might cure you of the urge before it has consumed too much of your time.

The first thing we do, let's kill all the language lawyers. – Henry VI, Part II