Corso 2021/2022

Appunti di Fondamenti di Programmazione

Francesco De Lucchini

1. Introduzione alla programmazione

L'informatica è lo studio sistematico degli algoritmi che descrivono e trasformano l'informazione, la loro teoria e analisi, il loro progetto e la loro efficienza, realizzazione e applicazione.

Algoritmo

Sequenza precisa e finita di operazioni che portano alla realizzazione di un compito, implementata da programmi scritti in un opportuno linguaggio e perciò eseguibili da un calcolatore.

Le operazioni utilizzate si distinguono in tre tipologie:

- operazioni **sequenziali** *realizzano una singola azione*
- operazioni **condizionali** *in base al valore della condizione selezionano l'operazione successiva da eseguire*
- operazioni **iterative** *ripetono un blocco di operazioni finché non è verificata una determinata condizione*

Le caratteristiche principali di un algoritmo sono:

- **Eseguibilità** *ogni istruzione deve essere eseguibile in un tempo finito*
- **Non-ambiguità** *ogni istruzione deve essere univocamente interpretabile*
- **Finitezza** *ha un numero totale di istruzioni da eseguire finito*
- **Correttezza** *svolge correttamente il compito per cui è stato realizzato*
- **Efficienza** *termina correttamente nel modo più veloce possibile*

Due algoritmi si dicono equivalenti quando hanno lo stesso **dominio di ingresso** *(ad esempio i numeri interi)*, lo stesso **dominio di uscita** e in corrispondenza degli stessi input producono gli stessi output. Ogni algoritmo è rappresentabile graficamente tramite il diagramma di flusso o *flow chart,* il quale tramite parallelepipedi simula ingresso e uscita, tramite rombi simula le condizioni e tramite rettangoli simula le istruzioni.

Linguaggio di programmazione

Il linguaggio naturale non si presta a descrivere processi computazionali automatizzabili e non evita le ambiguità, è dunque necessario utilizzare una notazione formale ossia un linguaggio di programmazione, ne esistono diversi e sono tutti caratterizzati da:

- **Sintassi** *insieme di regole formali per costruire frasi corrette nel linguaggio*
- **Semantica** *insieme dei significati da attribuire alle frasi costruite nel linguaggio*

Grammatica BNF (Backus-Naur Form)

Tip: questo argomento non viene mai chiesto agli esami, serve solo a capire alcuni concetti!

Dato un alfabeto V *(insieme di simboli)*, l'universo linguistico V* *(v-star)* è l'insieme di tutte le sequenze finite di elementi *(frasi)* di V possibili; di conseguenza un linguaggio LG su alfabeto V è un sottoinsieme di V* composto da frasi *(elementi di V*)* che soddisfano la sintassi espressa da una grammatica formale G <*V, VN, P, S*> dove:

- V *è l'insieme finito di simboli terminali*
- VN *è l'insieme finito di simboli non terminali*
- P *è l'insieme finito di regole di produzione*
- S *è un simbolo non terminale detto simbolo iniziale*

In altre parole si dice linguaggio LG l'insieme di frasi di V derivabili da S applicando le regole di produzione P, e le frasi di un linguaggio di programmazione vengono dette programmi di tale linguaggio.

La grammatica BNF è una grammatica le cui regole di produzione sono della forma:

X *(simbolo non terminale)*

$A_1, \dots, A_n$ *(simbolo terminale o non)*

Il **metalinguaggio** è un linguaggio secondario, in genere alcune parole del linguaggio naturale, usato per descrivere sintattica e semantica del principale, e le sue *"frasi"* sono dette costrutti.

one of è un costrutto del metalinguaggio, il quale indica che X assume solo uno tra i vari A_i

A_i può essere seguito dal costrutto del metalinguaggio |*opt* che ne indica l'opzionalità

A-seq è un costrutto del metalinguaggio che indica una sequenza di termini A_i

Esempio di grammatica per i numeri interi: *(derivazione left-most)*

V = {*-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9*} *simboli terminali*
VN = {*cifra, cifra-seq, intero*} *simboli non terminali*
S = *intero* *simbolo iniziale*
Regole di produzione P:

cifra
one of
0 1 2 3 4 5 6 7 8 9

cifra-seq *(modo ricorsivo di indicare una sequenza)*
cifra cifra-seq|*opt*

intero
-|*opt cifra-seq*

2. Rappresentazione dell'informazione

L'informazione è qualcosa di astratto e per poterla manipolare bisogna rappresentarla; in un calcolatore si rappresenta tramite i BIT *(Binary digIT, cifra binaria)*, che possono assumere solamente i valori 0 o 1; una sequenza di *n* bit può dunque contenere 2^n "*informazioni*".

Una qualsiasi sequenza di 8 bit è detta Byte ed è l'unità di misura dell'informazione, altre ricorrenti unità di misura della quantità di informazione sono i Kilobit *(*2^{10} *bit)*, Megabit *(*2^{20} *bit)* e Gigabit *(*2^{30} *bit) (e i corrispettivi in Byte)*.

Rappresentazione dei caratteri

La rappresentazione dei caratteri può avvenire tramite varie codifiche, tra cui:

- ASCII Standard *(su 7 bit)* converte 128 caratteri *(*2^7*)*
- ASCII Estesa *(su 8 bit)* converte 256 caratteri *(*2^8*)*

Nota: Il linguaggio C++ supporta solamente la codifica ASCII Estesa

Rappresentazione dei numeri naturali

I numeri naturali utilizzano la cosiddetta rappresentazione posizionale dove, data una sequenza di bit, quello più a sinistra è il più significativo e quello più a destra il meno.

Teorema fondamentale della rappresentazione dei numeri naturali:
Data una base B *(maggiore o uguale a due)* ogni numero naturale *n (compreso lo 0)* minore di B è associato ad un simbolo elementare detto cifra, la cui sequenza rappresenta in modo univoco i numeri naturali maggiori o uguali a B.

3. Basi del linguaggio di programmazione C++

Sintassi

Un programma in C++ è costituito da sequenze di parole dette *token*, costituiti da sequenze di:

- Digit *one of 0 1 2 3 4 ...*
- Letter *one of _ a b ... z A B ... Z*
- Special *one of ! % ^ ...*

Queste vengono delimitate da *whitespaces (spazio, tabulazione, ritorno carrello e nuova riga)*, e da *comments (sequenze di parole e whitespaces comprese tra /* e */ oppure tra // e la nuova riga)*.

Il C++ è case sensitive e i token vengono divisi in categorie sintattiche elementari:

- Identifier *i nomi assegnati a variabili, funzioni, classi… (sempre diversi dalle keyword)*
- Keyword *le parole chiave del linguaggio come: and while for…*
- Literal *denotano i valori assegnati alle variabili in fase di scrittura del codice*
- **Operator** *necessari per il calcolo delle espressioni, come: + - / …*
  Un operatore è caratterizzato da quattro proprietà:
  - **Posizione** *prefisso, postfisso o infisso (in mezzo: a + b)*
  - **Arietà** *numero di argomenti (unario, binario o ternario)*
  - **Priorità** *ordine di precedenza*
  - **Associatività** *ordine di esecuzione*
- Separator *terminano le istruzioni, separano e raggruppano, comprendono: ; () {} …*

Tipi

Si dividono in:

- Tipi f**ondamentali** *(o tipi aritmetici)*
  - Tipi predefiniti
    - Intero e naturale → *parte dei tipi numerici*
    - Reale → *parte dei tipi numerici*
    - Booleano
    - Carattere
  - Tipi di enumerazione
    Costituiti da insiemi di costanti intere servono a rappresentare informazioni non numeriche (ad esempio i colori di un semaforo).
- Tipi **derivati**
  Si ottengono a partire dai tipi predefiniti e consentono di creare strutture dati più complesse (reference, pointer, struct, union, array, class…)

Per eseguie certe operazioni è necessario convertire il tipo di uno o più argomenti, questo spesso avviene in modo implicito: il compilatore decide la conversione adatta in modo da **perdere la minor quantità di dati possibili**, la scelta si basa solo in base al tipo degli argomenti dell'operazione e non considera il tipo della variabile su cui andrà il risultato.

Osservazioni:

- *nella conversione tra intero e reale talvolta si può verificare una perdita di precisione perché gli interi sono rappresentati in forma esatta e i reali in forma approssimata*
- *in caso di argomenti dello stesso tipo ma di lunghezza diversa, quello con lunghezza minore viene convertito a quello di lunghezza maggiore (esempio: short int → int)*

Oggetti

Un oggetto è un gruppo di celle consecutive in memoria a cui viene attribuito un nome *(identifier)* e che vengono considerate dal programmatore come un'unica cella informativa.

Ogni oggetto ha:

- Attributi — ossia valore *(contenuto di tutte le celle)* e **indirizzo della prima cella**
- Tipo — cioè un insieme di tutti i possibili valori che può assumere, un insieme di operazioni consentite, e la dimensione che occupa in memoria *(sizeof)*.

Ogni oggetto viene:	Tipi derivati *(esempio struct)*	Tipi fondamentali *(esempio int)*
- Dichiarato	*struct name;*	*int idenftier;*
- Definito	*struct name {field1, fiel2 ...};*	*(inclusa nella dichiarazione)*
- Inizializzato	*name identifier = {1, 2, 3};*	*identifier = 5;*

Osservazioni:

- *Una stessa variabile può essere dichiarata più volte nello stesso scope, ma definita una sola volta all'interno di esso (**one-definition-rule**)*
- *Per le funzioni la dichiarazione corrisponde alla firma e al definizione al corpo, mentre l'inizializzazione non ha senso, la cosa più logicamente vicina all'inizializzaione è la chiamata a tale funzione, la quale ne crea un istanza inizializzata con i parametri scelti.*
- *Una variabile non inizializzata prende il valore che era in precedenza in memoria nelle celle che occupa, dunque è un valore "sporco" e imprevedibile.*

Stream

Detti anche *flussi*, necessari per la comunicazione tra l'esterno e il programma, gli stream sono una struttura logica costituita da una sequenza di celle *(byte)* **potenzialmente** illimitata la quale termina con una marca di fine stream. Uno stream è dotato di una variabile *streamstate*, un intero senza segno su 8 bit, la quale tramite il valore dei suoi primi tre bit esprime lo stato dello stream:

- 1° - **Bad bit** — *segnala un errore **non recuperabile** se 1*
- 2° - **Eof bit** — *segnale il raggiungimento della **marca di fine stream** se 1*
- 3° - **Fail bit** — *segnala un errore **recuperabile** se 1*

Lo stato di un flusso viene controllato da una serie di funzioni membro:

- **fail()** — *restituisce true se almeno uno tra fail e bad è a 1*
- bad() — *restituisce true se il bad bit è a 1*
- eof() — *restituisce treu se l'eof bit è a 1*
- **good()** — *restituisce true se tutti i bit sono a 0*

Uno stream può essere posto in una condizione, in tal caso viene chiamata la funzione fail e ne viene restituito il **complemento**, dunque ponendo uno stream in una condizione questa risulta vera se non ha errori e falsa in caso contrario. In questo caso lo stream non accetterà più operazioni fino a quando non verrà chiamata la funzione *clear(int=0)* la quale riceve in argomento un intero senza segno che rappresenta il nuovo valore dello streamstate *(viene fatto or logico e assegnamento)*. Da tastiera non esistono errori non recuperabili, mentre un esempio di errore recuperabile è mettere un char su un int.

Character INput *(cin)* è un'istanza predefinita della classe *istream* ed è l'unica collegata allo stream d'ingresso, che interagisce con la tastiera *(diventando effettivo solo dopo aver letto il carattere di ritorno al carrello)* o, se ridiretto in fase di compilazione, con il file specificato.

Se usato con l'operatore di traslazione a destra si ferma in caso di spazio bianco *(saltando però quelli in testa)* o in caso di errore di assegnamento, mentre se usato tramite la funzione membro *cin.get(identifier)* scrive sulla variabile carattere *identifier* anche spazi bianchi. Ci sono altre due funzioni membro utili della classe istream: *get(buf, dim, delim)* che legge caratteri dallo stream e li memorizza nel vettore *buf* fino ad arrivare al carattere *delim* o al numero di caratteri *dim*; e *read(s,n)* che legge *n* byte dal flusso e li memorizza a partire dall'indirizzo contenuto in *s*.

Character OUTput *(cout)* è un istanza predefinita della classe *ostream* ed è l'unica collegata allo stream di uscita, si usa con l'operatore di traslazione a sinistra e converte l'espressione argomento dell'operatore in una sequenza di caratteri che vengono trasferiti nelle varie celle dello stream a partire dalla prima; anche questo stream può essere ridiretto su un file. Alcune funzioni membro utili della classe ostream sono *put(c)* che trasferisce il carattere *c* sullo stream di uscita e la funzione *write(s,n)* che trasferisce la sequenza di *n* caratteri contenuti in *s* sullo stream di uscita.

La **manipolazione dei file** avviene tramite file-stream, gestiti dalla libreria **<fstream>**, associati ad un file di percorso *string (ed estensione facoltativa)* tramite la funzione membro *open*(*string, mod*), applicata ad un *identifier* di tipo *fstream*, dove *mod* è la modalità d'accesso e può essere:

- ios::in *per la lettura, segue le regole di cin, il file deve essere già presente.*
- ios::out *per la scrittura (da zero), segue le regole di cout, il file se non presente viene creato, se già presente ne sovrascrive i dati.*
- ios::out | ios::app *per la scrittura (in aggiunta), segue le regole di cout, il file se non presente viene creato, se già presente non ne sovrascrive i dati ma riparte dalla marca di fine stream precedente.*

In alternativa è possibile includere solamente la libreria per la lettura **<ifstream>** e dunque eseguire *ifstream identifier(string)* per aprire in lettura, oppure includere **<ofstream>** ed eseguire *ofstream identifier(string)* o *ofstream identifier(string, ios::app)*. La funzione membro che chiude lo stream aperto e libera il file per altri processi è *fstream_identifier.close()*, uno stream chiuso può essere riaperto in qualsiasi modalità ed in ogni caso tutti gli stream aperti vengono chiusi una volta usciti dallo scopo o dal programma.

Funzioni

Allo scattare di una chiamata *(creazione di un'istanza)* ad una funzione la prima cosa che viene allocata sullo *stack* è lo spazio necessario al tipo di ritorno della funzione, successivamente vengono allocati gli argomenti formali ed infine le variabili locali alla funzione; gli argomenti vengono inizializzati con i valori passati alla funzione e, al termine della funzione, tutto ciò che è stato allocato sullo *stack* viene eliminato.

Una funzione viene detta **ricorsiva** se nelle sue istruzioni contiene anche un'invocazione a se stessa *(con argomento diverso altrimenti entra in un ciclo infinito)*, ed almeno un caso base in cui fermarsi. Si può distinguere tra ricorsione in testa, quando la chiamata ricorsiva è l'ultima ad essere eseguita, o ricorsione in coda, quando la chiamata ricorsiva avviene per prima *(dopo i casi base)*, in questo modo il codice verrà eseguito a cascata una volta verificata la condizione base.

Gli **argomenti** di una funzione si dicono **di default** se vengono inizializzati nella dichiarazione della funzione *(omessi nella definizione)*, questi però devono essere gli ultimi argomenti e non possono assumere valori di variabili globali né altri argomenti; questi possono essere omessi durante la chiamata oppure sostituiti.

Una funzione provoca un **effetto collaterale** se *(non considerando la variabile di ritorno)* modifica, tramite riferimenti, puntatori o variabili globali, altre variabili esterne alla funzione; nota che anche cout è una variabile globale, dunque una funzione void che stampa produce un effetto collaterale.

Una funzione è identificata unicamente nel suo spazio di nomi da un **tipo**, che corrisponde al nome della funzione più il tipo dei vari argomenti passati alla funzione; dunque, se due funzioni differiscono solamente per tipo di ritorno queste ultime sono viste in modo uguale dal compilatore e quindi provocano errore *(doppia definizione)*.

Quando due o più funzioni hanno lo stesso nome ma tipo diverso si parla di **overloading**. Il compilatore sceglie attentamente quale delle alternative eseguire solamente in base agli argomenti passati e non in base al tipo di ritorno; se la chiamata dovesse risultare ambigua *(ad esempio funzione void, funzione con argomenti tutti di default e chiamata vuota, oppure funzione con due argomenti double, funzione con due argomenti int e chiamata con un int e un double)* provocherebbe un errore in fase di compilazione.

Una funzione può essere inoltre argomento di un'altra funzione, nella chiamata andrà specificato solamente il nome della funzione senza parentesi, mentre nella dichiarazione andrà specificato il tipo di ritorno e il tipo degli argomenti, avendo libertà nell'assegnazione del nome: data una funzione del tipo *int massimo(int n)…* e un'altra funzione *void media(int f_max(int), int n…)* allora posso effettuare la chiamata *media(massimo, …)* passando la funzione massimo come argomento.

```
# Puntatori a funzione

void(*pointer)();       function_type(* identifier)(function_arguments);
pointer = &fun2;        L'indirizzo puntato da pointer è la funzione, anche se
                        viene covertito implicitamente a uno se stampato a video

void fun(void (*pf)());         Passaggio puntatore a funzione come argomento
fun(pointer);
```

Librerie

Sono insiemi di funzioni precompilate, vengono utilizzate dal programmatore previa inclusione dello specifico file di intestazione, tra parentesi angolari per le librerie del pacchetto standard e tra virgolette con estensione *h* per le librerie non standard; alcune delle librerie standard più utilizzate sono:

- **<cstdlib>** *include la funzione exit(code), abs ed una per generare numeri random.*
- **<cctype>** *include controlli sul tipo, quali: isalpha, isdigit, islower, isspace…*
- **<cmath>** *include funzioni matematiche, quali: pow, sqrt, floor, ceil e abs*
- **<cstring>** *include funzioni per c-stringhe, quali: strlen, strcmp, strncpy…*
- <**iomanip**> *manipola l'output tramite i seguenti puntatori a funzione, che hanno tutti effetto fino alla nuova occorrenza del manipolatore, tranne setw che ha effetto solo sull'istruzione di scrittura immediatamente successiva:*
  - dec, oct *ed* hex *cambia la base in cui vengono visualizzati i numeri.*
  - boolalpha *visualizza come true o false i valori booleani.*
  - setprecision(*integer*) *imposta le cifre significative della parte decimale.*
  - setw(*integer*) *imposta l'ampiezza del campo di scrittura, compresi eventuali spazi per l'allineamento che verranno inseriti a sinistra.*

Riferimenti

Un riferimento è un identificatore che individua un oggetto *(dunque ha un tipo, e questo corrisponde allo stesso dell'oggetto)*, il riferimento di default è proprio il nome di un oggetto quando questo è un identificatore, oltre a questo se ne possono definire altri *(sinonimi o alias)*. Un riferimento non occupa spazio in memoria, **non è una variabile** ma solo un altro modo per chiamare la stessa variabile.

I riferimenti sono **sempre costanti**, ossia una volta assegnati ad un certo oggetto, per tutto il loro tempo di vita si riferiranno sempre a quell'oggetto, inoltre vanno **sempre inizializzati** in fase di dichiarazione, che avviene in questo modo: *type &alias = identifier;* riga dal quale l'alias si userà sempre senza prefisso & *(altrimenti starei usando l'operatore di indirizzo di memoria)*. Nota che è anche possibile definire riferimenti di riferimenti: *type &alias_2 = alias_1 (senza &).*

I riferimenti sono utili quando dichiarati come argomenti costanti di un tipo derivato, per evitare una chiamata al costruttore di copia ipoteticamente non ridefinito o comunque per non sprecare memoria copiando membro a membro anche valori inutili.

I riferimenti possono anche essere utili nel tipo di ritorno di una funzione, in tal caso ciò che ritorna è un riferimento alla variabile espressa nel return *(senza &)*. Attenzione però a non ritornare un riferimento di un oggetto sullo stack che poi andrà distrutto! *(ad esempio una variabile locale o un argomento passato per valore)*.

Un riferimento può anche essere dichiarato **a costante** *(deve obbligatoriamente se la variabile a cui si riferisce è costante)*, in questo modo non solo la variabile a cui si riferisce non può cambiare ma anche il suo valore in memoria, proprio come una variabile costante, resterà tale. Tuttavia, esiste un modo per rimuovere l'attributo costante *(flag const)* da un riferimento, ossia tramite l'operatore *const_cast<type>(identifier)*, nel modo seguente: *int& cast = const_cast<int&>(const);* Attenzione però che scrivendo su una zona di memoria da cui è stato tolto l'attributo const causa un comportamento non definito e dunque imprevedibile.

Non propriamente riferimenti ma concettualmente simili sono i **typedef**, particolari "sinonimi" che definiscono degli identificatori *(detti nomi typedef)* usati per riferisi a tipi nelle dichiarazioni, ad esempio posso usare *typedef struttura* Lista* oppure *typedef int Vett[5]* per poter usare *vett* ogni volta che voglio creare un vettore di cinque interi.

Puntatori

Un puntatore è un oggetto il cui valore rappresenta **l'indirizzo di memoria di un altro oggetto** o di una funzione, il tipo dell'oggetto puntato determina il tipo del puntatore, che però occupa sempre una **dimensione costante** *(a prescindere dal tipo puntato!)*, dipendente dall'architettura del calcolatore *(32 o 64 bit)*; inoltre, a differenza dei riferimenti, i puntatori sono **vere e proprie variabili** in memoria.

È possibile creare **puntatori a costanti**, che non devono per forza essere dichiarati e inizializzati insieme, e puntano ad un variabile che sarà sempre considerata come di sola lettura *(la variabile puntata può essere cambiata, ma non modificata nel suo valore)*; e creare **puntatori costanti**, coi quali la variabile puntata potrà essere modificata nel suo valore ma non nel suo indirizzo, che deve essere specificato obbligatoriamente in fase di dichiarazione.

```
# Dichiarazione di puntatori costanti ed a costanti

const int C = 0;
int I;

    Posizione l'asterisco prima dell'identificatore
const int *p1 = &C;            Puntatore a costante
int const *p2 = &I;            Puntatore a costante

    Posizione l'asterisco prima del const
int *const p3 = &I;            Puntatore costante
const int *const p4 = &C;      Puntatore costante a costante
```

Dopo la dichiarazione la variabile puntata sarà accessibile con **pointer*. Per quanto riguarda i puntatori l'inizializzazione può essere separata dalla dichiarazione, questo però è pericoloso in quanto un puntatore non inizializzzato punterà a variabili casuali, potenzialmente riservate, e dunque causare un **segmentation fault** *(rilevato solo a runtime)* in caso provassimo a scrivere o leggere su di esse. E' inoltre possibile cambiare l'indirizzo della variabile puntata, puntare al vuoto *(nullptr, ossia l'indirizzo inesistente 0)* e dichiarare puntatori di puntatori.

```
# Nota: priorità inferiore dell'operatore di deferenziazione

int a = 0, *pun = &a;
*pun++; → Errato! il puntatore punta alla cella di memoria successiva
(*pun)++;  → Corretto! la variabile puntata viene incrementata

# Indirizzo di memoria di un vettore qualsiasi

char strv[5] = {'T','e','s','t','\0'};
cout << "&Puntatore: " << &strv << endl; → Errato (non è ciò che cerco)
cout << "&Primo oggetto: " << (void *)&strv[0] << endl; → Corretto
cout << "&Vettore: "<< (void *)strv << endl; → Corretto e facile

Utilizzo il cast a puntatore void perché altrimenti stamperebbe il contenuto
della c-stringa, per un vettore qualsiasi invece va bene anche solamente
utilizzare il suo identificatore per stampare l'indirizzo del vettore.

Nota che in questi casi l'indirizzo del puntatore coindice con la prima cella
del vettore, ma nel caso di c-stringa dichiarata con letterale o di vettore
dinamico questo sarebbe errato.

# Indirizzo di memoria di una c-stringa

const char *str = "Test";
cout << "&str: " << &str << endl; → Errato (non è ciò che cerco)
cout << "(void *)str: "<< (void *)str << endl; → Corretto
```

C-Stringhe

Una c-stringa è un vettore di caratteri che contiene almeno una volta la marca di fine stringa '\0', ossia il punto in cui lo stream di uscita smette di scrivere caratteri quando mandati a video, nel caso questa non sia presente lo stream continuerà a scrivere potenzialmente all'infinito, fino a quando non si imbatterà in una marca di fine stringa in memoria.

Si dice **lunghezza logica** di una stringa l'indice in cui si trova la prima marca di fine stringa, mentre si dice **lunghezza fisica** di una stringa il massimo numero di caratteri *(più marca)* che può ospitare.

Una c-stringa può essere inizializzata come un normale vettore, ossia *char *str = {char1, … , '\0'} (con marca di fine stringa finale obbligatoria)* oppure direttamente con *const char* str =* "*value_here*"; in questo caso verrà automaticamente aggiunta la marca di fine stringa. Nota che in una c-stringa lunga n si possono inserire un massimo di n-1 caratteri in quanto viene riservato lo spazio per la marca di fine stringa.

Ordinamento di un vettore

Il **selection sort** o *ordinamento per selezione* funziona nel modo seguente *(ordine di complessità n^2)*:

- Scorre il vettore *dim-1* volte
- Ad ogni giro scorre il vettore partendo dall'indice più basso *(= i+1)* e andando verso l'indice più altro *(< dim)* e considera *i* come indice di minimo corrente
- Cerca l'elemento più piccolo e lo scambia con quello di indice i

Il **bubble sort** è un altro algoritmo di ordinamento, sempre dello stesso ordine di complessità, che però offre performance migliori su vettori che per ipotesi potrebbero già essere *"quasi"* ordinati:

- Scorre il vettore *dim-1* volte
- Ad ogni giro scorre il vettore partendo dall'indice più alto *(= dim-1)* e andando verso il più basso *(> i)*, scambiando due elementi contigui se non sono nell'ordine giusto
- Ottimizzazione: se dopo un giro completo l'algoritmo non esegue scambi è possibili terminare con la certezza che il vettore sia ordinato!

Ricerca in un vettore

La **ricerca lineare** non ha particolari perquisiti e con un ordine di complessità relativamente basso (*n*) è una soluzione semplice ed efficace in ogni situazione.

La **ricerca binaria** è molto più veloce *(ordine log(n))* ma ha come perquisito un vettore ordinato: si confronta l'elemento da cercare con quello centrale del vettore, se corrispondono l'algoritmo termina, se l'elemento da cercare è maggiore allora l'algoritmo ripete l'operazione per la metà superiore *(e così via per le successive iterazioni)* e operazioni analoghe avvengono nel caso opposto.

4. Strutture dati base

CODA

Una coda è un insieme ordinato di dati di tipo uguale su cui è possibile effettuare operazioni di inserimento e di estrazione secondo la politica **fifo** *(first in first out)*.

È caratterizzata da un **vettore circolare di dimensione fissa e due indici**: front e back; il primo rappresenta dove andare ad eseguire l'estrazione, il secondo dove andare ad effettuare l'inserimento. Entrambi gli indici partono in posizione uguale *(per convenzione la 0 del vettore)* e la coda sarà dunque vuota, successivamente ad ogni inserimento ed estrazione front e back avanzano, rientrando dall'inizio del vettore ogni qualvolta raggiungano la fine. *(index = (index + 1) % dim)*.

Una coda di dimensione *dim* può contenere solamente *dim-1* elementi, in quanto altrimenti non sarebbe possibile discriminare una coda vuota *(front = back)* da una coda piena *(back è dietro a front)*.

PILA

Una pila è un insieme ordinato di dati di tipo uguale su cui è possibile effettuare operazioni di inserimento e di estrazione secondo la politica **lifo** *(last in first out)*.

È caratterizzata da **un vettore di dimensione fissa e un indice che scorre il vettore**, partendo da *-1* e arrivando a *dim-1*, in questi casi si parla rispettivamente di pila vuota e pila piena; inserimento ed estrazione avvengono entrambe dove punta l'indice, che rimane sempre sull'ultimo elemento inserito.

UNIONE

Un'unione è un tipo derivato che si dichiara e usa con la stessa sintassi di una *struct* e rappresenta **un'area di memoria che in tempi diversi può contenere dati di tipo differente**, che corrispondono dunque a diverse interpretazioni di un'unica area di memoria *(grande quanto la dimensione che occupa il tipo di dato maggiore all'interno dell'unione)*.

Il valore significativo è l'ultimo che ha ricevuto un assegnamento e in fase di dichiarazione viene specificato solo il valore iniziale del primo membro con *union {field, ...} identifier = {rvalue}*; Nota che inizializzare un'unione come una struttura dando un valore a tutti i campi è errato.

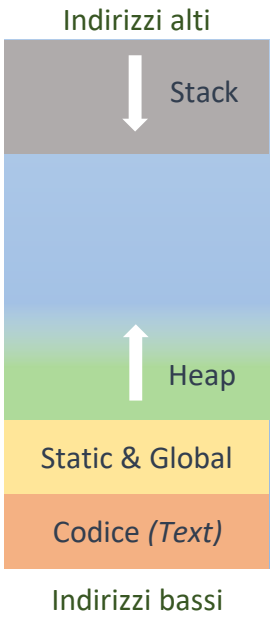
In un'unione come in una struttura non è possibile dichiarare un membro struttura dello stesso tipo, poiché avverrebbe una ricorsione infinita, è però consentito dichiarare un puntatore allo stesso tipo di struttura o unione *(vedi liste)*.

LISTA

Una lista è una struttura dati formata da **elementi dello stesso tipo collegati in catena, la cui lunghezza varia dinamicamente**, in cui ogni elemento è una struttura costituita da uno o più campi informativi e da un campo puntatore, contenente l'indirizzo dell'elemento successivo, fino all'ultimo elemento, avente come indirizzo *nullptr*; viene dunque memorizzato solamente l'indirizzo del primo elemento della lista, che sarà *nullptr* se la lista è vuota.

5. Gestione della memoria

RAM



Lo ***stack***, o memoria automatica è una **pila ad accesso sequenziale**, molto rapida ma **limitata** ad una dimensione decisa dal sistema operativo e dal compilatore e, quando raggiunge tale dimensione si parla di *stack overflow (un esempio è una ricorsione infinita).*

Lo ***heap***, al contrario **è un "mucchio" di memoria frammentata** e dunque più lento, ma **può espandersi** su richiesta del programma; le variabili vengono allocate in base allo spazio libero contiguo solo su richiesta del programmatore, che sarà poi responsabile anche del de allocamento di esse.

La memoria dinamica *o heap* è necessaria quando il programmatore non può stabilire il numero di variabili di un certo tipo che serviranno durante l'esecuzione del programma poiché **permette di allocare aree di memoria a runtime**, ed accedervi mediante puntatori, creando così oggetti dinamici che verranno deallocati al termine del programma, se non già deallocati dal programmatore tramite l'istruzione *delete* o *delete[]* in caso di vettori dinamici.

L'allocazione avviene tramite l'istruzione *type* identifier = new type;* l'operatore new cerca *sizeof(type)* memoria libera consecutiva nello heap, la etichetta come occupata **(senza nome)** e ne restituisce l'indirizzo della prima cella, che viene memorizzato in un puntatore del tipo opportuno *(quest'ultimo viene normalmente allocato sullo stack)*.

Memorizzare l'indirizzo restituito dall'operatore new è cruciale in quanto senza di esso non si avrebbe alcun collegamento con la memoria appena allocata e dunque non si avrebbe nemmeno la possibilità di de allocarla a runtime.

L'operazione di new fallisce in rare occasioni particolari, ad esempio se termina la memoria libera *(stack e heap si "toccano")*, per gestire questi rari casi è possibile usare la funzione *set_new_handler()* della libreria <new>, che ha come argomento una funzione void senza argomenti da chiamare in caso l'operazione fallisca.

L'operazione di delete invece può ha molti più punti critici: eliminare una variabile sullo stack causa errore a compilazione, eliminare la stessa variabile sullo heap due volte non ha un comportamento standardizzato ed è dunque da considerarsi un errore, inoltre dopo aver eliminato una variabile è ancora possibile utilizzarla nel codice poiché l'indirizzo di memoria rimane lo stesso, ma il valore in esso contenuto sarà profondamente diverso dato che la memoria nuovamente segnata libera potrà essere occupata nuovamente.

Nomenclatura dei vettori alla luce delle nuove conoscenze sulla memoria dinamica

Un vettore sullo stack prende il nome di **automatico** perché automatica è la sua de allocazione, in particolare si dirà:

- *Vettore automatico a dimensione statica* (*se la dimensione è una variabile globale o letterale)*
- *Vettore automatico a dimensione dinamica (se la dimensione è una variabile di blocco)*
  (Quest'ultimo viene anche definito come semi-dinamico)

Un vettore sullo heap prende il nome di **dinamico** perché dinamica è la sua dimensione, in particolare si dirà:

- *Vettore dinamico a dimensione statica (se la dimensione è una variabile globale o letterale)*
- *Vettore dinamico a dimensione dinamica (se la dimensione è una variabile di blocco)*

Un vettore dichiarato nello spazio globale prende il nome di **statico** *(e ha necessariamente dimensione statica ossia costante, nota a tempo di compilazione)*, e si dirà dunque:

- *Vettore statico a dimensione statica*

Endianness

Supponiamo di avere un'unione formata da due campi, uno intero e uno carattere, se assegno un valore al campo carattere, questo viene assegnato alla prima delle quattro celle di memoria *(indirizzo più basso)* o all'ultima *(indirizzo più alto)*? Questo non è standardizzato, se si verifica la prima ipotesi si parla memorizzazione **little-endian**, mentre se si verifica la seconda si parla di **big-endian**.

Supponiamo di avere una struttura formata come sopra, in certi casi può capitare che il *sizeof* della struttura ritorni 8 e non 5 come ci si aspetta (*un intero da 4 e un carattere da 1*), questo avviene perché alcuni compilatori su 32 bit sono ottimizzati per leggere su 4 byte ogni clock di processore, questo è noto come "**problema dell'allineamento in memoria**".

6. Definizioni aggiuntive e particolarità del linguaggio C++

CONDIZIONE

Una condizione è una qualunque **espressione della quale si possa stabilire un valore di verità** o, in altre parola, qualunque espressione che restituisca un risultato convertibile in maniera implicita ad un booleano.

UNITA' DI COMPILAZIONE

Un unità di compilazione **è un insieme formato da un file sorgente e dai file da esso inclusi tramite le direttive *#include***. Per fare un esempio in un progetto clion un'unità di compilazione è il main e le varie libreire incluse quali iostream ecc, se successivamente creassi un'altro file *cpp*, quella sarebbe una seconda unità di compilazione.

VISIBILTA'

La visibilità *o scope* di un determinato identificatore è **la parte di programma in cui tale identificatore può essere usato**; una variabile globale di dice che ha visibilità a livello di unità di compilazione. Se una variabile di **blocco *(sequenza di istruzioni racchiuse tra parentesi graffe)*** ne oscura una globale si può accedere a quest'ultima tramite l'operatore di risoluzione di visibilità globale *::*

SPAZIO DI NOMI

Uno spazio di nomi può essere dichiarato solo a livello di unità di compilazione o all'interno di un altro spazio di nomi e **corrisponde all'insieme di dichiarazioni e definizioni di un blocco**, ognuna delle quali introduce determinate entità dette membri. Per accedere ai membri occorre usare l'operatore di risoluzione di visibilità *namespace::member* oppure usare la direttiva *using namespace identifier;* la quale rende visibili fino alla fine del programma ogni membro di *identifier*, può però generare ambiguità e dunque errori noti a compilazione nel caso in cui usassi due spazi di nomi che contengono un membro con lo stesso nome.

Lo spazio di nomi è **aperto**, posso dunque estenderlo con membri a piacere, è inoltre presente uno spazio dei nomi di default ossia quello globale, uno spazio di nomi inizialmente vuoto contenente dichiarazioni e definizioni a livello di unità di compilazione.

Si può fare anche la using di un singolo membro *using spazio::membro* stando attenti però che non sia già stato dichiarato un oggetto con lo stesso nome nello spazio dei nomi globale o in uno in uso.

PREPROCESSORE

Parte di compilatore che elabora il codice del programma prima dell'analisi lessicale e sintattica, possono essere date particolari istruzioni a questa parte di compilatore, dette direttive per il preprocessore, e si distinguono dal prefisso *# (ad esempio #include ...)*.

MACRO

Simbolo che viene sostituito con una sequenza di elementi lessicali corrispondenti alla sua definizione

```
# L'intero codice viene compilato senza gli if inutili

#include <cstdlib>
#define LINUX oppure da riga di comando con la flag -DLINUX

int main() {
#ifdef LINUX system("clear");
#elifdef WINDOWS system("cls");
#endif
    return 0;
}

#ifndef PROJECT___CLASSNAME_H Senza queste istruzioni se includo il file più
#define PROJECT___CLASSNAME_H volte (o scelgo un nome già usato) non compila

class CLASSNAME {};

#endif
```

COLLEGAMENTO

In un programma un identificatore ha collegamento interno se si riferisce ad una certa entità visibile solo dall'unità di compilazione in cui è dichiarata, ha invece collegamento esterno qualora l'entità a cui si riferisce fosse accessibile anche da altre unità di compilazione, e tale entità deve essere ovviamente unica dove visibile. Di default gli identificatori di blocco o locali hanno collegamento **interno** mentre quelli globali (a livello di unità di compilazione) hanno collegamento **esterno** (a meno che non siano dichiarati const o static).

Oggetti e funzioni con collegamento esterno devono essere dichiarati per poter essere utilizzati in una seconda unità di compilazione del progetto, questo per quanto riguarda gli oggetti avviene tramite parola chiave **extern**, mentre la definizione avviene senza la parola chiave extern e con un valore iniziale; per quanto riguarda le funzioni invece la dichiarazione avviene specificando l'intestazione, e tale funzione viene anche definita se viene anche specificato il corpo di esssa.

MODULI

Un modulo è una parte di programma che svolge particolari funzionalità e che risiede su uno o più file; un **modulo servitore** offre risorse *(funzioni, variabili globali, tipi…)* ed è costitutito da un file di intestazione di formato *h* contenente solo le dichiarazioni e da un file di formato *cpp* contentente l'implementazione.

Compiere questa separazione ***(informatin hiding)*** serve a nascondere all'utilizzatore la struttura dati interna e l'implementazione di un determinato servizio, permettendone solo l'uso; questo consente di modificare la realizzazione di una parte di codice *(ad esempio cambiare algoritmo di ordinamento di un vettore)* senza influenzare il modo in cui il cliente utilizza quella data risorsa.

CLASSI DI MEMORIZZAZIONI

Un oggetto di blocco, dunque una qualsiasi variabile allocata sullo stack, viene detto di classe automatica perché automatica è la sua de allocazione alla fine del blocco.

Viene introdotta la keyword **static** per rendere un oggetto di classse statica, ossia con tempo di vita di una variabile globale e visibilità del blocco in cui è stata dichiarata; nota che anche una variabile globale può essere dichiarata static per avere un collegamento interno e dunque essere protetta, inoltre le variabili static vengono sempre inizializzate a zero in mancanza di inizializzaione.

Particolarità del linguaggio C++

```
# Tutti i prefissi o suffissi non sono case sensitive e servono solamente ad evitare una conversione implicita tra letterali

    Basi numeriche
int base_due = 0b10; // Prefisso 0b
int base_otto = 0765; // Prefisso 0
int base_sedici = 0xFDC; // Prefisso 0x

    Lunghezza interi
unsigned u = 1u;                32bit
short s = 1;                    <= 32bit (Di solito 16)
unsigned short us = 1;
long l = 1l;                    >= 32bit (Di solito 64)
long long ll = 1ll;             >= 64bit (Di solito 96)
unsigned long sl = 1ul;
unsigned long long sll = 1ull;
// Long vale solo per interi oppure double ma non float
// Unsigned vale solo per interi, caratteri e booleani

    Rappresentazione reali
float decimale = 1.5f;
float ometto_frazionaria = 1.f; // Ometto la parte frazionaria se vale zero
float ometto_intera = .5f; // Ometto la parte intera se vale zero
double reale = 1.5e+2; // 1.5 * 10^2

# Caratteri

Fra i caratteri si possono eseguire tutte le operazioni che si svolgono
normalmente tra interi, inoltre ci sono alcuni caratteri speciali quali:

    \n Nuova riga
    \t Tabulazione orizzontale
    \r Ritorno carrello (Sovrascrive il contenuto della riga corrente)
    \\ Barra invertita
    \' apice
    \" virgolette

Qui invece rappresento il massimo carattere possibile in basi diverse
char max_signed = 127;
unsigned char max_unsigned = 255;
char ottale = '\377';
char esadecimale = '\xFF';      // x solo minuscola
char reale = 127.9f;            // Troncato a intero
```

Traslazione oltre i bit consentiti

```
short bit = 1;
cout << "Valore prima della traslazione" << bit; // 1

bit <<= sizeof(bit) * 8;
cout << "Valore dopo della traslazione:" << bit; // 0
// Il bit che 'esce' viene scartato e subentra uno zero
```

Enumerati

```
enum {DIECI = 10, UNDICI, OTTO = 8}; // Dichiarazione senza nome
enum enumerato {ZERO, UNO, TRE = 3, ECC}; // Codificati su long int
enumerato enu = ZERO;
enum e; → Errato (no dichiarazione)

// Sono possibili tutte le operazioni sugli interi
int consentito = DIECI; // Consentito anche l'assegnamento

// Consentito assegnamento a tipo enumerato solo se dello stesso tipo enumerato
enu = 0; → Errato (assegnamento non di tipo enum)
enu = DIECI; → Errato (assegnamento di tipo enum ma diverso)
enu = ZERO;
enu = (enumerato)111; // Cast
enu = enumerato(222.5f); // Notazione funzionale
enu = static_cast<enumerato>(333);
```

Incrementi e decrementi

Operatore postfisso

```
i++++; (i++)++; ++i++; → Errato
Questo perchè l'operatore post fisso ritorna solo il valore (non incrementato)
```

Operatore prefisso

```
Mentre l'operatore prefisso ritorna un riferimento alla variabile già modificata
int *j = &++i; // (Prova che ritorna un riferimento)
----i; ++(++i); → Corretto

Supponiamo i intero di valore zero, allora:

cout << i << i++;      vale 0 perchè stampa 0 + 0 e poi incrementa
cout << i++ << i;      vale 1 perchè stampa 0 poi incrementa e stampa 1
// L'incremento va a finire subito dopo alla realizzazione
```

Switch

```
char expression = 0;
switch (expression){
    case 'A': break; Senza break eseguirebbe a catena tutte le istruzioni dal
    case 10: break;  primo match
    case .5f: break; → Errato (non converte implicitamente)
    case 9llu: break;
    case 5: case 25: break; Unire due casi
    case 10: break; → Errato (condizione ripetuta)
    case true:; E' l'ultima alternativa e non richiede dunque break
}
```

Vettori

Inizializzazione degli elementi specificati e zero ai restanti

```
int v4[3] = {3, 0, 2};
```

Fare accesso di un qualunque elemento fuori dall'array non dà errore ma stampa solo valori casuali, è comunque un errore logico ***(access out of bounds)*** *non segnalato nè a compilazione nè a runtime*

Operazioni consentite:

- ***Somma e sottrazione tra vettore e scalare*** *(aritmetica puntatori)*
- ***Sottrazione tra due vettori*** *(differenza indirizzi)*
- ***Confronto tra due vettori*** *(confronto indirizzi)*

Operatore di selezione con indice (solo tipo intero, no conv. Implicita)
*v[i] è come dire v + i * sizeof(*v)*
Per vettori mutidimensionali questo diventa:
*v[i][j] è come dire v[i*colonne + j] ossia v + (i*colonne + j) * sizeof(*v)*

Vettori multidimensionali

Dichiarazioni consentite

```
int m1[3][3];
// int m1[][c] → Errato (dimensione sconosciuta)
// int m1[r][] → Errato (dimensione sconosciuta)
int m2[][3] = {1,2, }; // E tutto il resto zero...
```

Specificare la prima dimensione o meno in questo caso è irrilevante

```
int c = 3;
int m3[c][c]; // Consentito
// int m4[c][c] = {1,2,3,4}; → Errato (dimensione non costante)

const int r = 3;
int m4[r][r] = {1,2,3,4}; // Consentito
```

Per poter utilizzare l'operatore di selezione con indice è necessario passare negli argomenti di una funzione l'array multidimensionale dotato di tutte le grandezze tranne la prima, il che può anche essere ottenuto tramite letterali o variabili globali ma è molto limitante in quanto adesso quella funzione accetterà solo array multidimensionali strettamente della stessa dimensione!

```
const int C = 5;
int dim_casuale = 10;

void matC_fun(int md[][C]){}; Accetta solo matrici n x C (5) (di tipo int[5]*)

int m10x5[dim_casuale][5];
matC_fun(m10x5); // Ok

int m5x5[dim_casuale][C];
matC_fun(5x5); // Ok
```

La stessa cosa avviene al contrario, non posso passare una matrice di dimensione n x m ad una funzione che accetta solo puntatori

Costanti

```
const int ci = 0;
int const ic = 0;
const int c; → Errato (no inizializzazione)
```

Strutture

```
struct struttura { int campo1; char campo2; };
struttura stack;        Camppi vuoti ma già allocati
stack = {1,2};          Inizializzazione stile vettore
stack = stack;          Copia membro a membro
// Non sono definite altre operazioni o confronti

struttura* puntatore = &stack;
puntatore->campo1 = 0; Operatore di deferenziazione e selezione
```

Nomi e identificatori

```
struct name {int i;};  Il nome permette di riferirsi ad un entità
name identifier = {1}; L'identificatore è il nome che viene assegnano ad una
identifier.i = 3;      precisa variabile

name name = {1};        I nomi possono essere identificatori
name.i = 3;
```

Collegamenti e classe di memorizzazione

Considerato il seguente file file.cpp

```
const int INTERN = 111;         interno
static int STATIC = 222;        interno
int EXTERN = 333;               esterno
struct punto {                  interno
    int x; int y;
};
punto point;                    esterno, Attenzione: andrà ridichiarata la struct
static void fun1();             interno
void fun2();                    esterno, Attenzione: la funzione non ha corpo
void fun3(){};                  esterno
```

Considerato il seguente file main.cpp dello stesso progetto

```
void fun1();    Statica, dichiarazione consentita ma esecuzione no!
void fun2(){}; Corpo definito, adesso può essere eseguita
void fun3();    Ok

int main() {
    extern int EXTERN;          Avviene la dichiarazione senza definizione
    EXTERN = 444;               La modifica è effettiva ovunque

    extern int STATIC;          Statica, dichiarazione si ma uso no!
    // cout << STATIC; → Errato (STATIC ha collegamento interno)

    extern int INTERN;          Costante, dichiarazione si ma uso no!
    extern int INTERN;          Dichiarazione più volte consentita
    // cout << INTERN; → Errato (STATIC ha collegamento interno)

    // extern punto point; → Errato (punto non dichiarato)
    struct punto {int x; int y;};      Interno
    extern punto point;                Consentito

Viene verificata l'uguaglianza solo tra gli identificatori (nome) di tipo
la struttura interna può essere diversa e dunque causare problemi logici solo a
runtime.

    // fun1(); → Errato (fun1 ha collegamento interno)
}
```

7. Classi

Il tipo classe consente a tutti gli effetti di **creare nuovi tipi di dato astratti**, oltre a ciò permette l'information hiding e di mantenere sempre uno stato consistente dei dati, la dimensione di una classe corrisponde alla somma del *sizeof* di tuti i mebri allocati sullo stack, escluse le funzioni. Una può essere costituita da:

- Tipi *enumerazioni o strutture*
- Campi dati *attributi, oggetti non inizializzati*
- Funzioni *anche dette funzioni membro*
- Classi *strettamente **diverse** da quella in cui sono membri*

Le classi membro di altre classi non hanno diritto di accesso ai membri dato dell'altra, dunque è necessario utilizzare la keyword ***friend***, che considera amico della classe l'identificatore specificato, quest'ultimo può essere una funzione globale o una classe differente *(in questo caso andrà dichiarata prima di essere posta friend altrimenti non verrà riconosciuta).*

Tutti i membri sono di default privati e dunque non visibili all'esterno della classe ***(la classe crea un namespace con il suo stesso nome** all'interno di cui inserisce tutti i mebri)*, ma possono essere resi pubblici a piacimento a partire dall'istruzione *public:,* tuttavia **l'operatore di risoluzione di visibilità può essere utilizzato solamente per funzioni membro, tipi e membri dato statici** *(se uso l'operatore di risoluzione di visibilità per un membro dato pubblico non statico causa errore a compilazione)*. **Ogni membro dati statico deve essere dichiarato** nel file che intende usarlo, nello spazio di nomi globale, con: *type Class_identifier::static_identifier;* inoltre se una funzione è dichiarata statica può accedere solamente a campi statici.

Una funzione può anche essere dichiarata costante tramite **suffisso *const*** al termine della dichiarazione, in questo caso potrà accedere a tutti i membri ma senza modificarne il valore; ad oggetti classe costanti possono essere applicate solo funzioni membro costatnti *(è buona norma applicare il suffisso const ovunque sia possibile farlo);* **const entra a fare parte del tipo della funzione**, dunque due funzioni con stesso nome e stessi argomenti possono differire per prorietà const.

Le funzioni membro definite *(implementate)* nella dichiarazione di una classe sono funzioni **inline**, ossia il compilatore ricopia il corpo della funzione al posto delle varie chiamate, questo procedimento può salvare del tempo a discapito però della dimensione del file, ed è sconsigliato per la leggibilità in caso di funzioni abbastanza grandi, è dunque consigliato definire le funzioni al di fuori della classe tramite l'operatore di risoluzione di visibilità.

Le operazioni predefinite di qualsiasi classe sono tre *(più due)* **(e non ne esistono altre)**:

- Distruttore *de alloca le variabili (solo sullo stack)*
- Costruttore di copia *copia membro a membro*
- Operatore di assegnamento *copia membro a membro*
- *Operatore di indirizzo &*
- *Operatore di sequenza ,*

Queste tre andranno **necessariamente** mascherate *(solamente dichiarate come funzioni membro private)* o ridefinite nel caso la classe utilizzi memoria dinamica.

Class(const Class&) *per mascherare il costruttore di copia*
Class& operator=(const Class&) *per mascherare l'operatore di assegnamento*

In ogni funzione membro il compilatore aggiunge un argomento in più ossia un riferimento all'oggetto di tipo classe con cui viene chiamata la funzione, questo viene assegnato ad un puntatore costante chiamato ***this*** e ci permette di ritornare un riferimento ad oggetto di tipo classe e consentire dunque la concatenazione delle chiamate di funzioni.

I costruttori sono funzioni membro il cui nome corrisponde al nome della classe, vengono invocati automaticamente alla creazione di un istanza di tale classe *(a seconda dei parametri specificati, tramite il meccanismo dell'overloading o degli operatori di default – lista di iniziallizzazione)* e non possono essere più chiamati successivamente; se un costruttore non ha parametri o ne ha ma con valori di default viene chiamato **costruttore di default** e se non presente viene aggiunto dal compilatore *(corpo e argomenti vuoti).*

Esistono i **costruttori di conversione** ossia quei costruttori che possono essere chiamati con un solo argomento *(ossia quando in dichiarazione inizializzo gli argomenti con un valore di default),* e che effettuano la conversione dal tipo degli argomenti al tipo classe.

Possono essere ridefiniti gli operatori predefiniti del linguaggio, senza però cambiarne le proprietà o crearne di nuovi, e devono **necessariamente avere per argomenti almeno un tipo di dato definito dall'ultente** *(se implementata come funzione membro questo è il punatore this)*, non posso per esempio ridefinire la somma tra due interi. Attenzione all'operatore di incremento o decremento postfisso che richiede un argomento intero fittizio *Class operator++(int);* !

Tutti gli operatori (**tranne: risolutore di visibilità ::, selettore di membro . e selettore di mebro attraverso puntatore a membro .*)** possono essere ridefiniti sia come mebri che globalmente *(tramite funzione friend)* **tranne l'operatore di assegnamento, indicizzazione, chiamata di funzione e selettore di mebro attraverso puntatore**, i quali devono essere obbligatoriamente ridefiniti come membri. Possono anche essere ridefiniti gli operatori di conversione esplicita, senza specificare il tipo durante la dichiarazione *(è implicito, esempio: operator int();).*

Gli operatori ridefiniti come funzioni membro **non sono simmetrici** *(gli operatori non sono intercambiabili, ad esempio int + classe non compila ma classe + int si)*, per evitare questo si devono utilizzare due funzioni friend globali in cui gli argomenti appaiono invertiti. Nota che gli operatori ridefiniti, se non specificato diversamente, **devono comportarsi esattamente allo stesso modo** di quelli standard, ad esempio l'operatore di incremento prefisso deve ritornare un riferimento all'oggetto modificato, mentre l'operatore di incremento postfisso deve ritornare una copia dell'oggetto antecedente alla modifica.

In una classe si possono dichiarare **riferimenti e membri costanti non inizializzati**, tuttavia devono necessariamente inizializzati tramite la lista di inizializzazione del costruttore, non possono infatti comparire nel corpo del costruttore perché sarebbero dei left-value.

Se all'interno di una classe vengono utilizzati tipi di altre classi il loro costruttore verrà chiamato nell'ordine in cui questi compaiono, qualora necessitino argomenti questi dovranno essere **specificati tramite lista di inizializzazione**, successivamente alla quale verrà eseguito il corpo del costruttore principale; segue che se tutte le classi secondarie hanno costruttore di default allora anche il costruttore principale potrà essere di default.

Nella dichiarazione di un array di oggetti classe è sempre necessario specificare la chiamata al costruttore, **se non definisco abbastanza elementi del vettore viene chiamato il costruttore di default,** se questo non è presente però il programma non compila; se l'array è allocato in memoria dinamica, deve necessariamente essere presente un costruttore di default poiché non sono possibili inizializzazioni esplicite *(anche perché se sapessi già cosa mettere nel vettore lo metterei sullo stack)*.