Appunti di Laboratorio di Programmazione

Giacomo Simonetto

Primo semestre 2023-24

Sommario

Appunti del corso di Laboratorio di Programmazione della facoltà di Ingegneria Informatica dell'Università di Padova.

"Il codice è professionale se altri lo possono riutilizzare"

Indice

1 Variabili

Definizione

- Un oggetto è un contenitore di dati, situato in memoria in grado di memorizzare valori che possono essere letti e modificati durante l'esecuzione di un programma.
- Una variabile è un oggetto identificato con un nome (*named object*).

In `C++` le variabili sono tipizzate, ovvero sono caratterizzate da un tipo che rimane costante per tutta l'esistenza della variabile.

1.1 Tipo di una variabile

Il tipo di una variabile definisce la natura di tale variabile, ovvero un range di valori che può assumere e un insieme di operazioni che possono essere eseguite. Specifica anche la dimensione in byte dell'area riservata in memoria. Il tipo può essere:

- **built-in**: tipi primitivi del linguaggio e inclusi nello standard
- **user-defined-type o UDT**: tipi definiti dall'utente in base alle esigenze del programma

Esistono delle funzioni di conversione tra tipi (casting) approfondite nel paragrafo 2.3.

Tipi built-in in |C++—

keyword	size	min	max
bool	1 bit	0	1
char	8 bit	-128	127
unisgned char	8 bit	0	255
short	16 bit	-32768	32767
unsigned short	16 bit	0	65535
int	32 bit	-2147483648	2147483647
unsigned int	32 bit	0	4294967295
long int	64 bit	-9223372036854775808	9223372036854775807
unsigned long int	64 bit	0	18446744073709551615
long long int	64 bit	-9223372036854775808	9223372036854775807
unsigned long long int	64 bit	0	18446744073709551615

keyword	size	min	nearest to zero	max	epsilon
float	32 bit	-3.40282e+38	1.17549e-38	3.40282e+38	1.19209e-07
double	64 bit	-1.79769e+308	2.22507e-308	1.79769e+308	2.22045e-16
long double	64 bit	-1.79769e+308	2.22507e-308	1.79769e+308	2.22045e-16

1.2 Nome variabile

Il nome di una variabile permette di richiamarla durante il programma in maniera univoca. Non ci possono essere più variabili con lo stesso nome. Alcune buone pratiche per nominare una variabile:

- evitare nomi troppo corti corti (eccetto per situazioni particolari es. contatori cicli)
- evitare acronimi difficili da interpretare
- usare nomi brevi che esplicitano il ruolo della variabile nel codice

1.3 Costanti

Le costanti sono particolari variabili il cui valore non può essere modificato durante l'esecuzione di un programma. Sono molto utili per identificare con un nome i valori costanti usati nel programma ed evitare di usare "magic numbers" di cui si potrebbe non comprenderne il significato. In base a come viene calcolato il valore, possono essere di due tipi:

- `const` se il valore assegnato è noto solo in fase di esecuzione
- `constexpr` se il valore assegnato è noto a tempo di compilazione

Esiste un terzo "caso" che utilizza la direttiva al preprocessore `#define`, solo che non è una vera variabile. Prima della fase di compilazione il preprocessore sostituisce tutte le occorrenze con il valore (o espressione) associato e, non avendo un vero tipo, non c'è type-safety.

```
1    const int i = a + b;           // valore noto a tempo di esecuzione
2    constexpr double pi = 3.14159; // valore noto a tempo di compilazione
3    #define w 12                   // direttiva al preprocessore
```

1.4 Literal

I literal sono particolari variabili costanti senza nome, contenute all'interno delle istruzioni del codice. Sono valori immediati utilizzati solo una volta. Un esempio le stringhe di testo da mandare in output o i valor i con cui le variabili sono inizializzate. I literal hanno un tipo definito automaticamente dal compilatore.

```
1    constexpr double pi = 3.14159; // 3.14159 e' literal di tipo float
2    string str = "Hello World";    // "Hello World" e' literal di tipo const char*
```

1.5 Enumerazioni

Le enumerazioni o enumerators sono delle strutture per definire delle etichette per esprimere delle scelte in un elenco. Una variabile enumeratore può assumere tutti i valori definiti nella dichiarazione. I valori sono associati ad un indice ed è possibile accedere all'i-esima etichetta tramite un costruttore.

```
1    enum Month {
2        January, February, March, April, May, ...
3    };
4
5    Month m = January;  // dichiarazione e inizializzazione di m
6    Month m = Month(5); // 5a etichetta (May)
```

È possibile definire uno scope per gli enumeratori aggiungendo la parola `class` nella dichiarazione:

```
1    enum class Month {
2        January, February, March, April, May, ...
3    };
4
5    Month m = Month::January; // necessario specificare lo scope
```

2 Istruzioni o statement

Le istruzioni o statement sono segmenti di codice che specificano un'azione che l'elaboratore dovrà eseguire al momento dell'esecuzione del programma. In genere terminano con `;` o con `{...}` (nel caso di istruzioni composte). Le direttive al preprocessore non sono istruzioni in quanto vengono risolte in fase di compilazione e non compaiono sull'eseguibile finale. Le istruzioni principali in `C++` sono:

- expression statements: espressioni da valutare
- compound statements: istruzioni composte, in genere seguite da un blocco `{...}`
- selection statements: selezioni (`if-else`, `switch-case`, ...)
- iteration statements: iterazioni (`for`, `while`, `do-while`, ...)
- declaration statements: dichiarazioni, inizializzazioni, definizioni
- try blocks: blocchi `try-catch`

2.1 Dichiarazioni, inizializzazioni, definizioni

Dichiarazioni

La dichiarazione è un'istruzione che introduce un nome in uno scope ne specifica il tipo. Le dichiarazioni possono essere di variabili o di funzioni. Non tutte le dichiarazioni hanno effetto sulla memoria (es. le dichiarazioni di funzioni non alterano la memoria).

```
1    extern int a;            // dichiarazione di una variabile
2    int somma(int a, int b); // dichiarazione di una funzione
```

Definizioni

La definizione è un'istruzione che specifica completamente l'entità introdotta. Viene specificato il nome, il tipo, eventuali valori di inizializzazione e dettagli implementativi.

```
1    int a = 1;                 // definizione di una variabile
2    int somma(int a, int b) { // definizione di una funzione
3        return a + b;
4    }
```

Inizializzazioni

L'inizializzazione di una variabile è una parte dell'istruzione di dichiarazione o definizione che assegna un valore iniziale alla variabile appena introdotta. È sempre opportuno inizializzare le variabili al momento della loro dichiarazione, altrimenti si potrebbe andare incontro a "bug oscuri".

```
1    int a;     // dichiarazione di una variabile
2    int a = 1; // dichiarazione di una variabile con inizializzazione
```

Differenza tra dichiarazione e definizione

Nella dichiarazione si introduce soltanto il nome di un'entità all'interno di uno scope, ma questo risulta inutilizzabile senza una successiva definizione. Le variabili soltanto dichiarate non hanno uno spazio riservato in memoria e non possono, quindi, essere utilizzate. Le funzioni soltanto dichiarate hanno solo l'header (o intestazione) e necessitano di una definizione per essere utilizzate.

Nella definizione si specificano tutte le informazioni necessarie per l'entità introdotta, rendendola utilizzabile a pieno nel programma. Nel caso delle variabili viene inclusa anche l'assegnazione di uno spazio in memoria. Per le funzioni viene indicata l'implementazione all'interno di un blocco `{...}`.

È possibile eseguire più dichiarazioni dello stesso nome, purché siano coerenti tra loro, ma al massimo si potrà avere una singola definizione.

2.2 Espressioni

Espressioni

L'espressione è il più piccolo segmento di codice usato per esprimere la computazione. Può essere semplice (literal, variabile, LValue, RValue), oppure composto (operazione con operandi).

```
    "Hello World";  // espressione semplice (literal)
    nome_variabile; // espressione semplice (nome variabile)
    a = b + c;      // espressione composta da operatore e operandi
```

Operatori

Gli operatori possono essere unari, binari o ternari in base al numero di operandi che coinvolgono e si dividono in base alla loro funzione:

- **operatori di assegnamento**

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

- **operatori di incremento e decremento**

```
    ++a; // preincremento  -> a = a + 1
    --a; // predecremento  -> a = a - 1
    a++; // postincremento -> a = a + 1
    a--; // postdecremento -> a = a - 1
```

- **operatori aritmetici**

```
    +a      // somma unaria
    -a      // sottrazione unaria (opposto)
    a + b   // somma
    a - b   // differenza
    a * b   // prodotto
    a / b   // divisione
    a % b   // modulo
    ~a      // NOR bitwise
    a & b   // AND bitwise
    a | b   // OR bitwise
    a ^ b   // XOR bitwise
    a << b // shift verso sinistra
    a >> b // shift verso destra
```

- **operatori logici**

```
    !a      // NOT logico
    a && b // AND logico
    a || b // OR logico
```

- **operatori di confronto**

```
    a == b  // uguaglianza
    a != b  // diverso
    a < b   // minore
    a > b   // maggiore
    a <= b  // minore o uguale
    a >= b  // maggiore o uguale
    a <=> b // segno della differenza a-b
```

- **operatori di accesso**

```
a[b]  // accesso in memoria
*a    // referenziamento (contenuto all'indirizzo puntato)
&a    // dereferenziamento (indirizzo della variabile)
a.b   // membro di oggetto
a->b  // membro di puntatore
a.*b  // puntatore al membro di un oggetto
a->*b // puntatore al membro di un puntatore
```

- **altri operatori**

```
a(...)      // chiamata a funzione
a, b        // separazione tra parametri (puo' essere ridefinito)
a ? b : c   // confronto condizionale: return b if a == true, c if a == false
static_cast      // casting
dynamic_cast     // casting
const_cast       // casting
reinterpret_cast // casting
C_style_cast     // casting
new         // allocazione dinamica
delete      // deallocazione
sizeof      // dimensione del tipo di una variabile o oggetto
typeid      // ...
noexcept    // ...
alignof     // ...
```

Ordine di valutazione di un'espressione

In un'operazione non è possibile determinare con esattezza l'ordine di lettura degli operandi. Utilizzando la stessa variabile più volte in un'espressione si potrebbe andare incontro a risultati inattesi.

```
a = b + c       // non e' possibile stabilire se sara' letto prima b o c
a = v[i] + i++ // possibile errore -> viene prima letto v[i] o eseguito i++?
```

Side effect degli operatori

Il side effect di un operatore indica la modifica che l'operatore esegue agli operandi, indipendentemente dal risultato finale dell'operazione.

```
a++     // risultato: a+1, side effect: incremento di a
a -= b  // risultato: a-b, side effect: a = a-b
```

Overloading e significato degli operatori

Grazie all'overloading degli operatori (classi) è possibile che uno stesso operatore abbia diversi significati in base agli operatori su cui è invocato.

```
cout << "Hello"   // operator<< inserisce una stringa nell'output buffer
cout << 4.5       // operator<< inserisce un double nell'output buffer
a + b             // somma tra interi
"Hello" + "World" // giustapposizione di stringhe
```

Rvalue e Lvalue

- Un lvalue è l'elemento a sinistra di un operatore di assegnamento. Viene modificato durante l'esecuzione dell'operazione, per cui deve essere modificabile.
- Un rvalue è l'elemento a destra di un operatore di assegnamento. Non viene modificato (a meno di side effects) per cui può essere anche costante.
- Le variabili possono essere sia lvalue, sia rvalue, i literals, le costanti e i valori restituiti dalle funzioni (a meno di lvalue reference o puntatori) sono solo rvaule.

2.3 Casting

Casting esplicito e implicito

La conversione o casting può avvenire in due modi:

- **casting implicito**: quando non viene specificata l'operazione di casting
- **casting esplicito**: quando si esplicita l'operazione di casting

```
1    double x = 12 // casting implicito da int a double
2    double x = static_cast<double>(12) // casting esplicito da int a double
```

Type safety

Il cast implicito può essere di due tipi:

- **type-safe conversion**: quando si esegue la conversione senza perdita di dati, in generale da un tipo con minore capacità a uno con maggiore capacità
- **non-type-safe conversion**: quando si esegue la conversione con perdita di dati, in generale da un tipo con maggiore capacità a uno con minore capacità

Le conversioni non-type-safe (con perdita di dati) ed eventuali overflow non sono segnalati dal compilatore.

Narrowing conversion

Da `C++11` c'è la possibilità di eseguire le conversioni con controllo di narrowing utilizzando le `{}` per indicare al compilatore di segnalare errore nel caso di conversioni insicure:

```
1    int x (2);    // int x = 2           -> type-safe, nessun errore
2    int x (2.3);  // int x = 2.3         -> non-type-safe, nessun errore
3    int x {2};    // int x = 2   + check -> type-safe, nessun errore
4    int x {2.3};  // int x = 2.3 + check -> non-type-safe, errore
```

Funzioni di casting esplicito

In `C++` sono presenti funzioni per eseguire il cast in modo esplicito tra diversi tipi:

- `static_cast` per conversione tra tipi built-in
- `dynamic_cast` conversione tra classi derivate
- `const_cast` conversione da `const` a `non-const` e viceversa
- `reinterpret_cast` conversione tra due tipi non relazionati (es. lettura di dati binari da file, buffer, o sensore e conversione in dati built-in)

Nel cast esplicito, il compilatore non esegue nessun controllo su overflow o perdita di dati.

3 Scope, namespace e visibilità

Scope

Lo scope è una regione di codice di un programma con la caratteristica che le entità (variabili, funzioni, ...) dichiarate al suo interno hanno validità (esistono) solo all'interno di quello scope. L'esistenza dello scope permette di rendere i nomi locali e riduce i problemi di clash sui nomi. Il nome di una variabile deve essere tanto descrittiva quanto più è esteso lo scope su cui è dichiarata.

Tipi di scope

- **globale**: al di fuori di ogni altro scope
- **scope di classe**: codice all'interno della classe
- **scope locale**: codice all'interno di un blocco `{}`, esempio nel corpo di una funzione
- **scope di statement**: codice all'interno di un blocco `{}` di uno statement (`if-else`, `for`, ...)
- **namespace**: scope con un nome definito globalmente o in un altro namespace

Scope globale

Le entità che di solito sono dichiarate globalmente sono le funzioni e i namespace. Questi sono disponibili all'interno dell'intero file in cui sono dichiarati e in tutti i file che lo includono. È possibile dichiarare variabili globali, solo che sorgono numerosi problemi:

- le variabili globali sono accessibili a chiunque -¿ non c'è incapsulamento
- tutte le funzioni possono modificarne il valore -¿ non c'è incapsulamento
- il debug risulta più difficile perché l'errore potrebbe essere ovunque
- il codice risulta più astruso perché il passaggio di dati tra le funzioni non avviene esplicitamente attraverso parametri

Scope annidati

È possibile dichiarare più scope annidati ad esempio: blocchi di statement annidati (`if-else` annidati), funzioni all'interno di classi (funzioni membro), classi annidate (poco usato), classi in funzioni (poco usato). Non è possibile dichiarare funzioni in funzioni.

3.1 Namespace

Il namespace è uno scope con un nome. Non definisce nessuna entità (funzione o variabile). Risulta molto utile se si utilizzano più librerie con funzioni o classi con nomi uguali: è possibile confinare le dichiarazioni di classi o funzioni con nomi uguali in diversi namespace in modo da non avere collisione tra nomi e richiamarle con il fully-qualified-name `namespace::membro`.

```
namespace Graph_lib {
    struct Color {};
    ...
}
namespace Text_lib {
    struct Color {};
    ...
}
Graph_lib::Color // si riferisce alla struct all'interno del namespace Graph_lib
Text_lib::Color  // si riferisce alla struct all'interno del namespace Text_lib
```

Per evitare di usare sempre il fully-qualified-name si ricorre alle `using declaration` o `using directive`. Queste hanno validità globale, per cui è opportuno non inserirle negli header files, altrimenti verranno importate insieme agli header all'insaputa del programmatore.

```
using std::cout;      // using declaration -> valido solo per std::cout
using namespace std; // using directive -> valido per tutti i membri di std
```

4 Librerie

Le librerie sono degli strumenti (o strutture) usati per importare codice scritto da altri o scritto dall'utente in diversi progetti o programmi. Sono composte da un insieme di files divisi in file header e file sorgente.

File header

I file header raggruppano gli header (o dichiarazioni) di diverse funzioni implementate dall'utente, che si riferiscono ad uno stesso problema. Terminano con l'estensione `.h` e il nome ne riassume il contenuto.

File sorgente

I file sorgente contengono le definizioni delle funzioni contenute nei corrispettivi header. Terminano con l'estensione `.cpp` e hanno lo stesso nome del corrispettivo header.

Linking

Per utilizzare una libreria all'interno di un programma è necessario includerla all'inizio del file attraverso la direttiva `#include` seguita dal nome della libreria. Se la libreria fa parte delle librerie standard del compilatore si indica il nome della libreria tra `<>` e non è necessario indicarne l'estensione. Se la libreria è Implementata dall'utente si indica tra `""` ed è necessario indicare l'estensione `.h`.

```
1    #include <iostream> // libreria dell'implementazione standard del cpp
2    #include "mylib.h"  // libreria implementata dall'utente
```

Si importano solo i file header di una libreria. I relativi file sorgente non si importano nei files, ma si passano al compilatore (linker) in fase di compilazione.

Include guards

Per evitare che una stessa libreria sia importata pià volte inutilmente, si utilizzano delle direttive al preprocessore chiamate include guards. Nel caso in cui la libreria sia già stata importata, il contenuto del file header viene ignorato.

```
1    #ifndef MYLIB_h
2    #define MYLIB_h
3    ...  // dichiarazioni
4    #endif // MYLIB_H
```

Librerie statiche e linking statico

Il linking statico avviene quando tutte le librerie necessarie al programma sono contenute all'interno dell'eseguibile finale. Il processo avviene in tre fasi:

1. il preprocessore aggiunge gli header ai sorgenti e si crea un blocco per il main e uno per ogni libreria
2. il compilatore crea i due file object compilati
3. il linker crea il file eseguibile unendo i due file object

- `main.cpp + mylib.h` $\rightarrow$ `main.o` $\longrightarrow$ `main.o + mylib.o` $\rightarrow$ `main.exe`
- `mylib.cpp + mylib.h` $\rightarrow$ `mylib.o` $\nearrow$

Spesso le librerie sono distribuite sotto forma di file precompilati. In tal caso verrà saltata la fase di compilazione delle librerie e il linker le inserirà nell'eseguibile alla fine. Il vantaggio di questo processo è avere un unico file con il programma e tutte le librerie e le dipendenze necessarie al funzionamento.

Librerie dinamiche e linking dinamico

Nel caso in cui più programmi condividano le stesse librerie si può optare per il linking dinamico. Ciò consiste nella creazione di una copia della libreria condivisa tra i diversi eseguibili. Le librerie condivise sono chiamate `SO` (shared object) su Linux o `DLL` (Dynamic Linking Library) su Windows. I vantaggi sono:

- evitare di avere copie superflue della stessa libreria
- aggiornare e ricompilare la libreria senza dover ricompilare i sorgenti

5 Funzioni

Le funzioni sono segmenti di codice identificate con un nome, che ricevono dei parametri o argomenti in input (detti parametri formali) e restituiscono un valore in output. La dichiarazione delle funzioni deve contenere tutte le informazioni sopra definite, mentre la definizione deve anche contenere il blocco con le istruzioni da eseguire. La funzione viene chiamata con il suo nome e le vengono passati i parametri richiesti detti parametri

```
/**
 * Dichiarazione (header) della funzione
 * - nome funzione:      somma
 * - parametri formali:  double a, double b
 * - tipo del valore di ritorno: double
 */
double somma(double a, double b);

// Definizione (implementazione) della funzione
double somma(double a, double b) {  // dichiarazione (header) della funzione
    return a + b;                   // corpo della funzione
}

c = somma(f,e); // Chiamata della funzione
```

5.1 Passaggio dei parametri

Passaggio per copia

Nel passaggio per copia, i parametri passati vengono copiati all'interno dei parametri formali. In questo modo la funzione riceve una copia dei valori, per cui questi possono essere modificati all'interno della funzione senza alterare i relativi valori nella funzione chiamante. Lo svantaggio è che quando si passano dati di grandi dimensioni, la copia risulta svantaggiosa.

Passaggio per reference

Il passaggio per riferimento o per reference permette di non effettuare la copia dei parametri, andando ad agire direttamente sui parametri del chiamante. Lo svantaggio è che la funzione potrebbe modificare i parametri del chiamante in maniera inaspettata. Le reference accettano solo lvalue (variabili modificabili). Quando avviene il passaggio per reference si suppone che i parametri coinvolti verranno modificati.

Passaggio per const reference

Il passaggio per `const reference` consiste nel passaggio per riferimento in cui i parametri all'interno della funzione sono dichiarati costanti. In questo modo non possono essere modificati all'interno della funzione e di conseguenza non verranno modificati nel chiamante. Le const reference accettano sia lvalue (variabili), che rvalue (literas o costanti).

```
int somma(std::vector<int> v); // passaggio per copia
int somma(std::vector<int> &v); // passaggio per reference
int somma(const std::vector<int> &v); // passaggio per const reference
sommaValori = somma(vec); // chiamata della funzione (indipendente dai casi sopra)
```

Considerazioni

Si cerca di evitare di usare il passaggio per reference in quanto è poco esplicito (non si sa se i valori verranno modificati o meno).

- per passaggio di piccoli valori → copia
- per passaggio di grandi dati non modificabili → const reference
- per passaggio di grandi dati da modificare → reference (o puntatori)
- per valori da restituire → valore di ritorno

6 Gestione della memoria

6.1 Memory Layout di un programma

In un programma la memoria è suddivisa in settori, partendo dalla parte alta (high address) verso la parte bassa (low address) si ha:

- **stack**: struttura lifo che memorizza i record di attivazione di ogni funzione, si espande dalla parte alta verso la parte bassa della memoria
- **heap**: contiene i dati allocati dinamicamente (memoria dinamica), si espande verso la parte alta
- **bss** (o uninitialised data segment): contiene le variabili globali (e statiche) non inizializzate
- **data** (o initialised data segment): contiene le variabili globali (e statiche) inizializzate
- **text**: contiene le istruzioni da eseguire

6.2 RDA o record di attivazione

Il record di attivazione di una funzione è l'insieme dei dati memorizzati dello stack quando una funzione è in esecuzione. Comprende i parametri passati, le variabili locali, il valore restituito ed eventuali altri dati necessari alla corretta esecuzione del programma.

Ad ogni chiamata di una funzione si crea un nuovo RDA che verrà aggiunto allo stack, sopra quello del chiamante. Alla terminazione di una funzione, lo stack di tale funzione viene estratto e si riapre quello del chiamante.

6.3 Durata in memoria delle variabili

La durata in memoria di una variabile è il tempo per cui tale variabile si trova in memoria ed è indipendente dallo scope, ovvero dall'area di codice in cui è possibile accedere a tale variabile. In base alla durata in memoria, le variabili si dividono in:

- **variabili statiche**: memorizzate nel `data` o `bss`, esistono per tutta la durata del programma, da prima dell'esecuzione del main, fino a dopo la sua terminazione. Mantengono il valore anche fuori dal loro scope e si dividono in:
  - globali: se definite nello scope globale, per cui accessibili ovunque
  - locali: se definite all'interno di uno scope, con accesso limitato allo scope, tramite la parola chiave `static`
- **variabili globali**: memorizzate nel `data` o `bss`, si comportano come le variabili statiche, essendone un caso particolare
- **variabili locali automatiche**: definite all'interno di uno scope (non globale), sono memorizzate nello stack all'interno dell'RDA di tale scope e vengono gestite automaticamente dal compilatore. Quando viene rimosso l'RDA, tutte le variabili locali automatiche verranno distrutte. Nel caso in cui tali variabili si trovano all'interno di un blocco iterativo queste vengono dichiarate e distrutte ad ogni iterazione e, nonostante ci possa essere ottimizzazione, non si potrà conoscerne il valore all'iterazione successiva.

Nel caso in cui si hanno più file (o translation unit) e si utilizzano inizializzazioni che usano variabili `extern`, si potrebbe andare incontro ad un errore in quando non è possibile stabilire l'ordine di creazione in memoria di diversi files.

```
1    // File 1 //
2    int x1 = 2;
3    int y1 = x1 + 2;
4
5    // File 2 //
6    extern int y1;   // richiama y1 del file 1
7    int y2 = y1 + 2; // non e' detto che y1 sia gia' stata creata in memoria
8    // potrebbero essere prima gestite qelle del file 2 e poi quelle del file 1
```

7 User Defined Type - UDT

Un UDT è un particolare tipo non presente tra i tipi built-in del `C++` e definito dal programmatore secondo le proprie esigenze. Nella libreria standard sono presenti alcuni UDT tra cui `vector`, `string`, `list`, `ostream`, ... Un tipo deve avere la possibilità di avere:

- descrivere lo stato in cui si trova (attraverso delle variabili)
- avere delle operazioni per cambiare stato (attraverso member function o helper function)

7.1 Struct

Le `struct` sono strutture ereditate dal `C`, costituite univocamente da variabili membro. Le variabili non hanno alcun controllo di accesso e l'oggetto può facilmente andare in uno stato non valido. Sono utilizzate principalmente come contenitori di dati.

```
struct Date {
    int day;   // variabile membro
    int month; // variabile membro
    int year;  // variabile membro
};
```

7.2 Classi

Introduzione e struttura

Le classi sono strutture tipiche del `C++` in grado di rappresentare concetti astratti in un programma. Sono composte da:

- variabili membro
- costruttori e distruttori
- member function (e helper function)
- overloading operatori
- meccanismo di protezione dei dati (incapsulamento)
- gestione della copia dei dati

```
class Date {
    private:
        int day;   // variabile membro privata
        int month; // variabile membro privata
        int year;  // variabile membro privata

    public:
        Date(int d, int m, int y);           // costruttore
        int get_day() const;                 // member function (getter)
        int set_day();                       // member function (setter)
        Date operator+(const Date &d) const; // overloading operatori
        ...
};

Date add(const Date &d1, const Date &d2); // Helper function
```

Stato di un oggetto e validità

Un oggetto ha uno stato valido se rispetta un certo invariante (regola definita dal programmatore) al momento della creazione e durante ogni azione eseguita su tale oggetto. Per rispettare tale regola si fa uso dell'incapsulamento e dei costruttori.

Incapsulamento

L'incapsulamento consiste nel "nascondere" variabili membro o member function rendendole inaccessibili all'esterno della classe. Il programmatore fornisce delle funzioni (chiamate `getters`) per accedere solo ai dati non sensibili e altre funzioni (chiamate `setters`) per modificarli rispettando una serie di controlli implementati dal programmatore. In questo modo è possibile modificare l'oggetto rispettando l'invariante. Tale principio si implementa attraverso gli attributi:

- **private**: rende l'entità accessibile solo all'interno della classe
- **protected**: rende l'entità accessibile solo all'interno della classe ed eventuali classi derivate
- **public**: rende l'entità accessibile a chiunque

Costruttori

I costruttori sono funzioni particolari con le seguenti proprietà:

- hanno stesso nome della classe a cui appartengono e non hanno valore di ritorno
- sono invocati automaticamente al momento della dichiarazione dell'oggetto garantiscono la validità dell'invariante alla creazione dell'oggetto.
- se non viene specificato alcun costruttore, il compilatore ne fornisce uno di default, nel caso in cui ne viene fornito uno, questo sovrascrive quello di default che non sarà più utilizzabile
- si cerca di implementare i costruttori esternamente alla classe, altrimenti vengono interpretati come `inline` dal compilatore; per implementarli esternamente bisogna specificare il nome della classe a cui appartengono `NomeClasse::NomeCostruttore(...)`
- oltre ai parametri specificati nella dichiarazione, hanno un parametro implicito immutabile `this` che punta all'oggetto appena creato, utile per accedere ai membri privati di tale oggetto

I costruttori si possono dividere in:

- **costruttore di default**: costruttore senza parametri

```
    Date() { ... }

    Date d;   // chiama il costruttore di default
    Date d{}; // chiama il costruttore di default
    Date d(); // errore -> viene vista come dichiarazione di una funzione
```

- **costruttore generico**: costruttore con parametri

```
    Date(int d, int m, int y) { ... }

    Date d(1,2,2024);        // costruttore con parametri vecchio stile
    Date d{1,2,2024};        // costruttore con parametri e type safety
    Date d = {1,2,2024};     // verboso, equivalente a quella sopra
    Date d = Date{1,2,2024}; // verboso, equivalente a quella sopra
```

- **costruttore con member initializer list**: costruttore in cui vengono specificate le inizializzazioni nella dichiarazione, fuori dal corpo del costruttore. Questa pratica a volte risulta inutile perché bypassa eventuali controlli sulla validità dei parametri

```
    Date(int _d, int _m, int _y) : d{_d}, m{_m}, y{_y} { ... }
```

- **delegating constructor**: costruttore che chiama un altro costruttore per evitare di ripetere operazioni base di inizializzazione già implementate e successivamente aggiunge eventuali correzioni

```
    Date(int d, int m) : Date(d, m, 1970) { ... }
    Date d{1,1}; // equivale a Date d{1,1,1970};
```

- **costruttore con parametri di default**: costruttore in cui sono specificati i valori che assumono i parametri, se non vengono passati al momento dell'invocazione

```
    Date(int d = 1, m = 1, y = 1970) { ... }
    Date d{2};         // equivale a Date d{2,1,1970};
    Date d{2,2};       // equivale a Date d{2,2,1970};
    Date d{2,2,1980}; // equivale a Date d{2,2,1980};
```

- **costruttore explicit**: costruttore che disabilita il casting implicito e la copy-initialization

```
    explicit Date(int d, int m, int y) { ... }
    Date d{1,1,1970};    // ok
    Date d = {1,1,1970}; // errore -> copy-initialization

    explicit A(int m) { ... } // costruttore che blocca il casting int -> A()
    void fx(A a) { ... }      // funzione che accetta un oggetto A() come parametro
    fx(A(2)); // ok     -> nessun casting
    fx(3);    // errore -> casting implicito disabilitato
```

- **costruttore con initializer list**: usa una `initializer_list` da cui prendere i parametri, usata molto spesso quando serve inizializzare un oggetto vector-like.

```
    Date(initializer_list<int> lst) {
        d = lst.size() > 0 ? lst.begin()[0] : 1;    // se lst ha almeno 1 elemento
        m = lst.size() > 1 ? lst.begin()[1] : 1;    // se lst ha almeno 2 elementi
        y = lst.size() > 2 ? lst.begin()[2] : 1970; // se lst ha almeno 3 elementi
    }

    Date d{2,2,1980};    // ok  -> {2,2,1980} e' una initializer list
    Date d(2,2,1980);    // err -> (2,2,1980) non e' una initializer list
    Date d = {2,2,1980}; // ok  -> {2,2,1980} e' una initializer list
    Date d(2);           // err -> (2) non e' una initializer list
    Date d{2};           // ok  -> {2} e' una initializer list
    Date d{2,2};         // ok  -> {2,2} e' una initializer list
    Date d{2,2,2,2};     // ok  -> {2,2,2,2} e' una initializer list
```

Member function

Le member function sono funzioni interne alla classe, in grado di poter accedere e modificare le variabili membro della classe. A differenza delle normali funzioni:

- si dichiarano internamente alla classe e si implementano esternamente, se implementate internamente vengono interpretate come `inline` dal compilatore; per implementarle esternamente bisogna indicare il nome della classe a cui appartengono `tipoRestituito NomeClasse::NomeFunzione(...)`
- per invocarle bisogna usare la notazione `nomeOggetto.nomeFunzione(...)`
- oltre ai parametri specificati nella dichiarazione, hanno un parametro implicito immutabile `this` che punta all'oggetto invocante, utile per accedere ai membri privati di tale oggetto

Le member function possono avere l'attributo `const` che specifica che non modificano nessun parametro ricevuto, nemmeno l'oggetto da cui sono invocate. Le member function `const` possono, quindi, essere invocate su oggetti costanti o su const reference. Il meccanismo dei parametri di default dei costruttori è esteso anche alle funzioni.

Helper function

Le helper function sono normali funzioni che vengono spesso collegate ad una classe in quanto implementano funzionalità specifiche della classe a cui sono legate.

Member vs helper functions

Bjarne suggerisce di implementare poche member function e delegare il più possibile dei compiti alle helper function. In questo modo se ci sono errori di invariante, il problema sarà sulle poche member function implementate.

Overloading degli operatori

È possibile ridefinire il comportamento degli operatori esistenti in C++ per una classe, attraverso opportune definizioni. Non è possibile effettuare overloading di operatori non esistenti e bisogna rispettare il numero di parametri previsti (unario, binario ...). È possibile implementarne l'overloading tramite member function o con helper function. È necessario che almeno uno dei due parametri dell'operatore sia UDT e non built-in (se fossero entrambi built-in) si ricadrebbe su un caso già gestito. Alcuni operatori hanno un pattern obbligato, ovvero l'ordine dei parametri non può essere scambiato

```
T operator+(const T &t1, const T &t2) { ... } // overloading con helper function
T T::operator+(/*this,*/ const T &t2) { ... } // overloading con member function
c = a + b;

T &operator++(T &t); // preincremento -> resituisco valore modificato
T operator++(T &t);  // postincremento -> restituisco valore non modificato

// pattern obbligato per overloading operator<< -> non puo' essere member function
ostream &operator<<(ostream &os, const T &t) {
    return os << t.member ... ;
}
```

Distruttori

I distruttori sono member function che vengono chiamate automaticamente quando un oggetto esce dal suo scope. Servono per liberare la memoria occupata dalle variabili membro e viene fornito automaticamente dal compilatore. È chiamato implicitamente, per cui la chiamata esplicita è un errore.

Nel caso in cui un oggetto abbia dei puntatori a memoria allocata dinamicamente, è necessario sovrascrivere il distruttore di default, inserendo la deallocazione della memoria allocata. Se ciò non viene fatto si va incontro a memory leak.

Se un oggetto contiene altri oggetti o altre variabili membro, il distruttore automaticamente provvede a chiamare il distruttore degli oggetti-membro e liberare le variabili. È sbagliato deallocare dinamicamente variabili non allocate dinamicamente.

```
class A {
    int *p;
    ...
    A(){ p = new int[100]; } // costruttore con allocazione dinamica
    ~A() { delete[] p; }     // distruttore con deallocazione dinamica
}
```

Copia - costruttore e assegnamento di copia

Quando viene effettuata un'assegnazione tra due oggetti dello stesso tipo, si effettua la copia membro a membro tra l'oggetto originale e l'oggetto di destinazione. Quando la copia viene utilizzata per costruire un nuovo oggetto o inizializzare una variabile viene chiamato il costruttore di copia, mentre quando la copia coinvolge oggetti già dichiarati, allora si invoca l'operatore di assegnazione `operator=`.

```
A obj1{1,2,3};
A obj2 = obj1; // copia membro a mebro da obj1 a obj2 (costruttore)
obj1 = obj2;   // copia membro a mebro da obj2 a obj1 (assegnazione)
```

Di default viene effettuata una copia membro a mebro, detta shallow copy, che è implementata di default dal costruttore di copia e dall'operatore di assegnamento di copia. Quando si hanno oggetti con puntatori, la shallow copy copia solo l'indirizzo, ma non duplica la memoria puntata. Si avranno due oggetti con puntatori alla stessa area di memoria. Per risolvere il problema si ricorre alla deep copy, per cui sarà necessario implementare il costruttore di copia e l'assegnamento di copia.

```
// header costruttore di copia
Obj(const Obj &a) { ... }

// operatore di assegnamento di copia
Obj &operator=(const Obj &a) { ... }
```

Nell'overloading dell'`operator=` è consigliato creare un puntatore temporaneo alla futura area di memoria con i dati, copiare i dati nell'area di memoria, eliminare i dati vecchi e infine sovrascrivere il puntatore con quello temporaneo. In questo modo si evita di eliminare i dati senza essere sicuri di avere quelli nuovi.

Move - costruttore e assegnamento di move

Quando ho un oggetto che contiene grandi quantità di dati allocati dinamicamente ed è in procinto di essere eliminato (es. valore di ritorno di una funzione), non ha senso effettuarne una copia. Basterebbe copiare l'indirizzo dell'area di memoria che contiene i dati. In questo modo si evitano operazioni unitili. Per fare ciò si definisce il costruttore di move e si esegue l'overloading dell'`operator=` per l'assegnamento di move.

```
1    // header costruttore di move
2    Obj(Obj &&a) { ... }
3
4    // operatore di assegnamento di move
5    Obj &operator=(Obj &&a) { ... }
```

Si nota che entrambi ricevono un rvalue, ovvero un oggetto che non potrà essere modificato, se non viene copiato. Siccome, nel momento di eliminazione dell'oggetto rvalue si invoca il distruttore, è bene impostare i membri dell'oggetto d'origine a valori di default. In questo modo si evita di deallocare i dati che memorizzava, siccome sono condivisi con il nuovo oggetto. Per rimuovere i dati dall'oggetto d'origine, i parametri non possono essere const reference.

Alcune volte il compilatore ottimizza gli istanziamenti degli oggetti di breve durante (valori di ritorno di funzioni), costruendoli direttamente sulla destinazione e bypassando eventuali copie e move. Questo processo è chiamato `copy-elision` e per evitarlo bisogna inserire in fase di compilazione `-fno-copy-elision` tra i parametri passati al compilatore.

Rule of 5 - istruzioni di copy e move

Si osserva che quando vengono utilizzati membri allocati dinamicamente all'interno di una classe, è bene:

1. implementare il distruttore per deallocare i membri dinamici

2,3. implementare il costruttore di copia e l'assegnamento di copia per corretta deep copy

4,5. implementare il costruttore di move e l'assegnamento di move per evitare copie superflue

Esiste una regola chiamata `Rule of 5`, per cui se è necessario implementare almeno una delle 5 member function sopra, allora sarà necessario implementare anche tutte le altre.

Accesso ai membri di una classe - `operator[]`

Per accedere ai dati di una classe che implementa una generica struttura dati position-based (es. vettore, stringa, ...) si utilizza l'`operator[]` in cui tra le quadre si indica l'indice della posizione a cui si vuole accedere. Per fare ciò si effettuano due overloading (const e non const) dell'operatore in questione.

```
1    // overloading non const -> restituisce un lvalue reference modificabile
2    Obj &operator[](int n);
3
4    // overloading const -> restituisce un rvalue reference di sola lettura
5    Obj operator[](int n) const;
```

8 Eccezioni

Le eccezioni sono un meccanismo di gestione degli errori in `C++`. Al lancio di una eccezione vengono chiusi ed estratti tutti gli RDA aperti fino a quando non si arriva ad una funzione in grado di catturare l'eccezione. Se l'eccezione non viene catturata nemmeno dal main, il programma viene terminato.

Definizione di una eccezione

L'eccezione è una classe particolare. Viene spesso definita all'interno di un'altra classe che la utilizza.

```
class A {
    private:
        ...
    public:
        ...
        class Invalid {}; // classe eccezione intenzionalmente lasciata vuota
}
```

Lancio di un'eccezione

Un'eccezione viene lanciata tramite la keyword `throw` seguita dal costruttore della classe eccezione.

```
if (invariante non verificato)
    throw Invalid();
```

Cattura di un'eccezione

L'eccezione viene catturata con un blocco `try-catch`.

```
try {
    ... // funzioni che possono lanciare l'eccezione
} catch (A::Invalid e /*eccezione da catturare*/) {
    ... // codice da eseguire quando l'eccezione viene catturata
}
```

Ereditarietà delle eccezioni

Per migliorare la gestione delle eccezioni è possibile definire una classe eccezione all'interno di una gerarchia di classi e sfruttare il polimorfismo per gestire più eccezioni contemporaneamente. Le eccezioni devono implementare la funzione `what()` che restituisce un messaggio di errore (`const char *`) che è possibile stampare in output.

```
// esempio di eccezione da lanciare quando la dimensione passata non e' valida
class InvalidLength : public std::logic_error {
    public:
        const char *what() const override { return "La dimensione non e' valida"; }
}

// esempio di cattura di un'eccezione basata su polimorfismo
try {
    ...
} catch (std::logic_error &e) {   // cattura le eccezioni derivate da logic_error
    std::cerr << e.what() << "\n";
} catch (std::runtime_error &e) { // cattura le eccezioni derivate da runtime_error
    std::cerr << e.what() << "\n";
} catch (std::exception &e) {     // cattura le altre eccezioni non ancora catturate
    std::cerr << e.what() << "\n";
}
```

9 Reference e puntatori

9.1 Reference

Le reference sono "variabili" collegate univocamente e immutabilmente ad altre variabili. Il collegamento viene stabilito nella loro definizione e non è possibile cambiarlo per tutta la durata della reference. (puntatore immutabile e dereferenziato automaticamente)

```
1    int a = 10;
2    int &b = a; // definizione reference della variabile a
3    a = 15  <->  b = 15 // a, b sono la stessa variabile
```

9.2 Puntatori

I puntatori sono variabili con un tipo (built-in o UDT) che contengono un indirizzo di memoria. Il tipo serve per specificare al compilatore come leggere o scrivere sull'area di memoria puntata e le operazioni che si possono fare con il dato. L'indirizzo può variare durante il programma.

```
1    int a = 10;
2    int *b = &a; // definizione puntatore alla variabile a
3    a = 15  <->  *b = 15 // a e il contenuto all'indirizzo in b sono equivalenti
4    int c = 20
5    *b = &c; // modifica dell'indirizzo puntato da b
6    c = 25  <->  *b = 25 // ora c e il contenuto all'indirizzo in b sono equivalenti
```

Operazioni con i puntatori

- `operator&` → referenziazione, restituisce l'indirizzo di una variabile o oggetto
- `operator*` → dereferenziazione, restituisce lvalue con il contenuto dell'indirizzo di memoria
- `nullptr` → valore di un puntatore con indirizzo non valido (null)

Tipi dei puntatori

Non esiste casting tra i tipi dei puntatori, per cui un puntatore a `int` non potrà mai essere assegnato ad un altro puntatore a `double`. Esiste inoltre un puntatore puro `void*` che non ha controlli sul type check.

9.3 Confronto tra puntatori e reference

Per i puntatori, l'assegnamento modifica l'indirizzo memorizzato nel puntatore, mentre per modificare il valore all'indirizzo di memoria, bisogna prima dereferenziare il puntatore ed eseguire l'assegnamento dereferenziato. Per le reference, invece, l'assegnamento modifica il valore delle variabili a cui sono linkate. Siccome per assegnare l'indirizzo ad un puntatore è necessario utilizzare l'operatore di referenziazione, il passaggio di parametri tramite puntatori avviene in maniera esplicita, a differenza delle reference.

10 Array

Introduzione

Gli array sono strutture dati ereditate dal `C` caratterizzate da una sequenza omogenea di elementi allocati in aree di memoria consecutive (senza buchi). La dimensione degli array deve essere conosciuta a tempo di compilazione. Esistono anche i `VLA` (Variable Length Array), ovvero array con dimensione non conosciuta a tempo di compilazione, ma non fanno parte dello standard `C`.

```
    int a[10]; // creazione array di 10 interi
```

Accesso e posizione in memoria

Gli elementi di un array sono indirizzati con un indice intero da 0 in poi e l'accesso è casuale (indipendente dalla posizione) e avviene con l'operatore `[]`. Vengono trattatati come variabili e possono essere globali (memorizzate nel `data` o `bss`) o locali (memorizzate nello `stack`).

```
    a[4] = 10; // scrittura nella 5a posizione
```

Algebra dei puntatori

Gli array possono essere trattati come dei puntatori const che puntano alla prima cella e che possono essere spostati di un numero finito di celle per accedere alle celle adiacenti. È possibile fare il viceversa: trattare un puntatore come un array.

```
    int a[10];
    int *p = &a[5]; // puntatore alla 6a cella
    p[-2] = 10; // scrittura sulla 4a cella
    p[2] = 15;  // scrittura sulla 8a cella
    p -= 2; // sposto il puntatore indietro di due celle
    p += 2; // sposto il puntatore avanti di due celle
```

Il salto dipende dal tipo del puntatore, se è di tipo `int`, salterà di 4 byte, se `char` salterà di 1 byte.

Stringhe in C

Le stringhe ereditate dal `C` sono array di char che terminano con `"\0"`. Hanno le stesse limitazioni e utilizzi degli array. I literal string sono stringhe di questo tipo.

Decadimento di un array

Una delle limitazioni degli array di questo tipo è che quando vengono passati ad una funzione (tramite puntatore al primo elemento) si perde la conoscenza della dimensione dell'array. Questo meccanismo di conversione da array a puntatore è una vecchia ottimizzazione del `C`, mantenuta nel `C++` per retrocompatibilità.

Inizializzazione

```
    int a[] = {1,2,3,4}; // array di dimensione 4
    int a[10] = {1,2,3,4}; // array di dim. 10 con {1,2,3,4,0,0,0,...,0}
    int a[10] = {}; // array di dim. 10 con tutte le celle a 0
    char s[] = {'a', 'b', 'c'}; // array di char, non stringa perche' manca '\0'
```

Errori da evitare

```
    int *p = nullptr;  *p = 7; // puntatore con indirizzo non valido

    int* f1() {       |   int& f2() {
        int x = 7;    |       int x = 7;
        return &x;    |       return x;
    }                 |   }
    int *p = f1(); // errore, la variabile puntata non piu' presente in memoria
    int &r = f2(); // errore, la variabile a cui e' linkata non esiste piu'
```

11 Allocazione dinamica della memoria

Allocazione dinamica con `new`

Per allocare dinamicamente un'area di memoria nell'heap, si utilizzano i puntatori e l'operatore `new`. I puntatori per accedere all'area di memoria possono essere utilizzati come array in stile `C`, con l'algebra dei puntatori.

```
int *p = new int;    // allocazione di un int
int *p = new int[4]; // allocazione di un array di 4 int
int *p = new int[n]; // allocazione di un array di n int (n variabile)
```

Inizializzazione

Bisogna distinguere l'inizializzazione del puntatore dall'inizializzazione dell'area di memoria puntata:

```
int *p;                      // puntatore non inizializzato
int *p = new int;            // puntatore inizializzato, intero non inizializzato
int *p = new int{2};         // puntatore e intero inizializzati
int *p = new int[4];         // puntatore inizializzato, array non inizializzato
int *p = new int[]{1,2,3};   // puntatore e array non inizializzati
int *p = new int[5]{1,2,3}; // puntatore e array non inizializzati
```

Quando si alloca un oggetto, viene chiamato il costruttore per ogni oggetto allocato. Se il costruttore ha bisogno di parametri, bisogna indicarli nell'allocazione.

```
X *p = new X;              // oggetto allocato con ctor senza parametri
X *p = new X[4];           // array di oggetti allocati con ctor senza parametri
Y *p = new Y{3};           // oggetto allocato con un parametro per costruttore
Y *p = new Y[4]{3,4,2,6}; // array di oggetti allocati con rispettivi parametri
```

Controllo validità

Per verificare che un puntatore contenga un indirizzo di memoria si utilizza la seguente condizione:

```
// metodo classico                  |   // abbreviazione
if (p != nulltptr) {                |   if (p) {
    // puntatore inizializzato      |       // puntatore inizializzato
}                                   |   }
```

Questo controllo non garantisce che l'area puntata sia accessibile e valida. Inoltre i puntatori non sono inizializzati di default a `nullptr` per cui nel caso di un puntatore non inizializzato, questo confronto non ha senso.

Deallocazione con `delete`

Per liberare l'area di memoria allocata dinamicamente si usa `delete`. Il sistema operativo conosce la dimensione dell'array da deallocare, per cui non serve indicarla nell'istruzione. Quando si deallocano oggetti, vengono chiamati i rispettivi distruttori.

```
int *p = new int;     ->  delete p;  // deallocazione di una variabile/oggetto
int *q = new int[10]; -> delete[] q; // deallocazione di un array
```

Dandling pointer

Dopo un `delete`, il puntatore mantiene l'indirizzo di memoria, anche se non è più valido. Per evitare errori è bene reinizializzarlo a `nullptr`.

Doppia cancellazione

```
int *p = int[100];
delete[] p; // deallocato array, ma puntatore mantiene l'indirizzo
delete[] p; // errore -> deallocazione di un'area gia' deallocata
```

Memory Leak

Quando si perde il riferimento ad un'area di memoria, si ha memory leak. L'area rimane occupata fino alla chiusura del programma. L'assenza di un garbage collector permette di ottenere una migliore efficienza computazionale e spaziale (posso eliminare subito dati di grandi dimensioni), però delega al programmatore l'attenzione di non provocare memory leak.

```
int *p = int;   int *q = int[10];
q = p; // l'array puntato precedentemente da q non e' piu' accessibile
```

Si verifica memory leak anche quando un puntatore viene eliminato per l'uscita dal suo scope. Per evitare questo errore, è necessario effettuare la deallocazione di ogni area di memoria dei puntatori che andranno persi all'uscita di ogni scope.

```
void fx(int a) {
    int *p = int[100];
    ...
    delete[] p; // deallocazione, altrimenti l'array non e' piu' accessibile
}
```

Vantaggi dell'allocazione dinamica

L'allocazione dinamica permette di:

- allocare dati di grandi dimensioni in maniera flessibile, solo per il tempo necessario
- allocare array con dimensione conosciuta solo a tempo di compilazione
- gestire efficacemente l'allocazione e la deallocazione

Gli svantaggi sono:

- il costo elevato dell'allocazione e deallocazione della memoria (bisogna chiamare il sistema operativo)
- la gestione delicata della memoria per evitare memory leak, dandling pointer o doppia cancellazione
- la gestione delicata dei puntatori

12 Template

Introduzione

I template sono strutture alla base della programmazione generica e del polimorfismo statico. Permettono di scrivere codice indipendente dal tipo dei dati, che poi verrà specializzato dal compilatore in base al tipo su cui verrà invocato. Questo processo è anche chiamato type generator, parametrized type o paramettized class, in quanto si generano un tipo diverso per ogni specializzazione e nella definizione si usano parametri al posto del tipo da specializzare.

Specializzazione

La specializzazione avviene in fase di compilazione, per cui si ha una maggiore velocità di esecuzione e se ci sono errori sui tipi verranno segnalati dal compilatore. Siccome si crea una specializzazione per ogni tipo utilizzato, si avranno possibili raddoppi di codice.

Struttura

```
template<typename T> // oppure template<class T>, crea parametro template
class vector {
    void push_back(const T &t); // T e' usato come parametro tipo
    ...
}

// parametro tipo per member function
template<typename T>
void vector<T>::push_back(const T &t) { ... }

// parametro tipo per funzione generica
// assumo esista operatore di confronto, se non eiste -> errore in compilazione
template<typename T>
T max(const T &x, const T &y) {
    return (x > y) ? x : y;
}
```

Interi come template

È possibile indicare anche interi come parametri template. Questo processo è utile quando si vuole implementare vettori o array per una generica dimensione che rimarrà costante per tutto il programma (es. triplette RGB). Questo permette maggiore efficienza e si evita l'utilizzo del free store.

```
template<typename T, int N>           // definizione tipi template
class array { ... }                   // definizione classe

array<unsigned char, 3> pixel{0,0,0}; // istanziamento oggetti
```

Template a deduzione automatica

Non serve sempre specificare i tipi template se si possono dedurre ad es. dai parametri della funzione.

```
template<typename T>
T max(const T &x, const T &y) { ... } // funzione template
k = max(i,j); // se i,j sono interi, non serve specificare max<int>(i,j)

template<typename T, int N>
void fill(array<T,N> &a; const T &val) { ... } // funzione template
fill(pixel, 100); // T,N calcolati automaticamente dal tipo del parametro pixel
```

Template su Header File

I compilatori richiedono necessariamente di specificare sia la dichiarazione, sia la definizione delle classi e funzioni template. Per questo motivo non è possibile separare interfaccia e implementazione in file header e file sorgente. Si sceglie di inserire le dichiarazioni in un file header `.h` e le implementazioni in un file header `.hpp` che poi andrà linkato alla fine del file `.h`. Nell'istruzione di compilazione non servirà aggiungere nulla perché ciò non costituisce una translation unit (sorgente da compilare).

13 Buona interfaccia

Quando di progetta una classe, per avere una buona interfaccia, bisogna considerare alcuni aspetti.

Considerazioni sui costruttori

- **costruttore di default**: ci sono valori di default comuni a tutti gli oggetti, ha senso tenerlo?
- **costruttore con parametri**: serve specificare dei valori particolari per l'oggetti in fase di istanziamento?
- **rule of 5**: è necessario implementare tali funzioni?
- **conversioni**: l'oggetto creato può essere convertito in altri tipi, oppure è necessario usare la keyword `expicit` nei costruttori?

Considerazioni su organizzazione dei progetti in più files

- i file header contengono le dichiarazioni delle funzioni e le definizioni delle classi
- i file sorgente contengono le definizioni delle funzioni e vanno inseriti nel comando di compilazione
- se è necessario fare `#include file.cpp` o inserire un file header nel comando di compilazione, allora c'è qualcosa che non è stato fatto correttamente

Per una maggiore organizzazione i file si suddividono in cartelle:

- **main directory**: contiene tutte le sottocartelle ed eventuali file come `README.md` e `CMakeLists.txt`
  - **build**: sono contenuti i file di compilazione e gli eseguibili
  - **include**: sono contenuti i file header `.h`
  - **src**: sono contenuti i file sorgente `.cpp` e il `main.cpp`

14 Compilazione, CMake e Git

14.1 Compilazione

Il processo di compilazione avviene in tre fasi:

1. **preprocessore**: gestisce le direttive al preprocessore, es. `#include, #define, ...`
2. **compilatore**: traduce in codice macchina i singoli file e produce file oggetto
3. **linker**: risolve i riferimenti e le relazioni tra i diversi file compilati e aggiunge le librerie necessarie

L'intero processo è gestito dal comando `g++ nome_file -o nome_eseguibile -parametri` che si occupa di tutte e tre le fasi.

14.2 CMake

CMake è uno strumento indipendente da compilatori e piattaforma che permette di indicare come sono organizzati i file in un progetto complesso per essere poi compilati agevolmente su ogni computer. Le indicazioni sono salvate all'interno di un file situato nella main directory chiamato `CMakeLists.txt`. Attraverso il comando `cmake ..` eseguito dalla cartella `build`, si generano i file che descrivono il progetto per sistema utilizzato (makefile, ninja, soluzione Visual Studio, ...) ed infine si può procedere alla compilazione attraverso i programmi di compilazione (makefile, ninja, ...)

14.3 Git e github

`Git` è un programma di controllo delle versioni e serve per tracciare, rimuovere e condividere modifiche in un progetto di codice. Esistono due modelli di software a controllo versione:

- sistemi centralizzati: tutte le modifiche e i file si salvano su un server ed ognuno può scaricare, modificare, ricaricare le proprie modifiche (non è possibile salvare modifiche localmente)
- sistemi a modello distribuito: ogni utente ha la propria stratificazione locale che può modificare a piacimento e infine caricarla in un server centralizzato

`Git` è appartiene al secondo tipo e il relativo server è `GitHub`.

15 Ereditarietà

Introduzione

L'ereditarietà tra classi è un meccanismo che permette di relazionare le classi secondo una relazione di dipendenza `classe base - classe derivata` che ne permette la condivisione di determinate funzionalità.

Relazione classe base - classe derivata

La classe base, in genere rappresenta un concetto astratto in codice che poi verrà specializzato in classi derivate. Le classi derivate ereditano dalla classe base tutti i membri (pubblici, privati e protected), però possono accedere solo ai membri public e protected. La relazione si stabilisce nella dichiarazione della classe con `class Derived : public Base { ... }`

Relazione "is a" - "has a"

- la relazione **"has a"** si implementa attraverso i membri di una classe; es. `Date` has a `day`
- la relazione **"is a"** si implementa con l'ereditarietà; es. `Derived` is a `Base`

Alcune notazioni

- **classe astratta**: una classe astratta è una classe non istanziabile, può essere ottenuta tramite la disabilitazione del costruttore (inserito protected), la disabilitazione dei costruttori di copia e move o l'inserimento di una funzione virtuale pura; è possibile definire puntatori ad una classe astratta.
- **funzione virtuale**: una funzione virtuale è una funzione di cui si vuole effettuare l'override nelle classi derivate. Bisogna sempre implementarla o definirla pura. Attraverso l'override delle funzioni virtuali si ottiene il polimorfismo dinamico o in runtime. Si aggiunge l'attributo `virtual` all'inizio della dichiarazione.
- **funzione virtuale pura**: una funzione virtuale pura è una funzione virtuale che non ha implementazione. Per specificare che la funzione non ha implementazione si aggiunge `= 0` alla fine della dichiarazione. Questo obbliga le classi derivate ad implementarla, altrimenti saranno instanziabili.
- **disabilitazione di funzioni**: per disabilitare una funzione non necesaria si aggiunge `= delete` alla fine della dichiarazione di una funzione (es. disabilitazione dei costruttori di copy e move).
- **override**: l'override la ridefinizione di funzioni virtuali della classi base nelle classi derivate. Per effettuare l'override bisgna avere stessi nome, parametri, constness, tipo restituito; inoltre è possibile inserire la keyword `override` in modo da ricevere errore di compilazione se ci sono probelmi. Per definire quale funzione verrà eseguita, si utilizzano le Virtual Table.

Problemi di slicing

Quando si copia una classe derivata all'interno di una classe base (grazie alla relazione "is a"), si effettua lo slicing dell'oggetto derivato, ovvero i membri e le funzioni extra della classe derivata che non sono presenti nella classe base vengono tagliati. Per evitare tale problema, si disabilitano i costruttori e gli assegnamenti di copy e move.

Virtual Table

Ogni oggetto con funzioni virtuali ha un virtual pointer creato dal compilatore che punta ad una tabella virtuale comune a tutte le istanze dell'oggetto. La tabella virtuale contiene i riferimenti delle funzioni virtuali da chiamare, altrimenti si avrebbe ambiguità siccome gli override hanno la stessa dichiarazione.

Dynamic Cast

È possibile definire un casting esplicito tra classi derivate e classi base attraverso l'operatore di casting esplicito `dynamic_cast<destination_type>(obj)`. L'upcasting (da classe derivata a classe base) è sempre concesso e non si hanno problemi. Il downcasting potrebbe essere problematico: se l'oggetto non soddisfa il tipo di destinazione, allora l'operatore restituisce `nullptr` nel caso di puntatori o lancia l'eccezione `bad_cast`.

Esempio di implementazione

```
class Base {
    private:
        int a; // accessibile solo all'interno di Base
    protected:
        int b; // accessibile da Base e dalle classi derivate
        Base(int _a, int _b) : a{_a}, b{_b} {} // costruttore classe base disabilitato
                                               // in questo modo e' classe astratta
    public:
        Base(const Base &) = delete;          // disabilitato costruttore copia
        Base(Base &&) = delete;               // disabilitato costruttore move
        int get_a() { return a; }             // getter per a
        int get_b() { return b; }             // getter per b
        virtual int sum() { return a + b; }   // virtuale somma tra a e b
        virtual int diff() = 0;               // virtuale pura (senza implementazione)
};

class Derived : public Base {
    private:
        int c; // accessibile solo all'interno di Derived
    public:
        Derived(int _a, int _b, int _c) : Base(_a,_b), c{_c} {} // ctor classe derivata
        int get_c() { return c; }             // getter per c
        int sum() { return Base::sum() + c; } // virtuale somma tra a, b, c
        int diff() { return c - b; }          // necessario implementarla
};

int main() {
    Base a(1,2);      // errore -> classe astratta
    Derived b(1,2,3); // ok -> classe istanziabile
    b.get_a();        // invoca la funzione della classe Base
    b.get_b();        // invoca la funzione della classe Base
    b.get_c();        // invoca la funzione della classe Derived
    b.sum();          // invoca la funzione di Derived
    b.diff();         // invoca la funzione di Derived
}

----------------------------------------

class B : public A {          |  class C : public B {    |  class D : public C {
    virtual void f() const;   |      void f() const;     |      void f();
    void g() const;           |      void g() const;     |      void g() const;
}                             |  }                       |  }

void call (const B &obj) {
    obj.f();
    obj.g();
}

call(b);  // chiama B::f e B::g -> obj ha tipo formale b e tipo reale b
call(c);  // chiama C::f e B::g -> obj ha tipo formale b e tipo reale c + override f
call(d);  // chiama B::f e B::g -> obj ha tipo formale b e tipo reale d + no override f

b.f();  b.g()  // chiama B::f e B::g
c.f();  c.g()  // chiama C::f e C::g
d.f();  d.g()  // chiama D::f e D::g
```

16 Polimorfismo

16.1 Polimorfismo statico

Il polimorfismo statico (o svincolato, risolto a tempo di esecuzione) è generato dall'uso dei template. La classe template, agli occhi del programmatore, può essere di diversi tipi contemporaneamente, che poi specializzeranno la classe in fase di compilazione in tante classi indipendenti. Si dice svincolato perché le classi template non hanno alcun collegamento tra di loro, hanno solo funzioni con lo stesso nome.

16.2 Polimorfismo dinamico

Il polimorfismo dinamico (o vincolato, risolto a tempo di esecuzione) è dato dall'utilizzo dell'ereditarietà, delle funzioni virtuali e dell'override di esse. Constiste nel vedere un oggetto di una classe base con il suo vero tipo (classe derivata) e gestire le chiamate delle funzioni invocate da tale oggetto attraverso le virtual tables. È detto vincolato in quanto le classi base e derivate hanno una relazione "is a" tra di loro ed è risolto a tempo di esecuzione attraverso le virtual table.

Confronto tra i due tipi di polimorfismo

1a. Nel polimorfismo dinamico sono sicuro che la funzione chiamata esista nella classe base o in una delle classi derivate

1b. Nel polomorfismo statico, se per un certo tipo manca la funzione da chiamare, viene segnalato un errore dal compilatore

2a. Il polimorfismo dinamico è più lento in esecuzione in quando le virtual table consumano risorse

2b. Il polimorfismo statico è più rapido in esecuzione, ma produce eseguibili più grandi

17 Standard Template Library

La Standard Template Library o STL è l'insieme di classi e funzioni comunemente utilizzate nel `C++` e opportunamente suddivise in librerie. La STL si divide in contenitori per memorizzare i dati `vector`, `string`, `list`, ... e algoritmi di manipolazione dei dati `sort`, `find`, ...

17.1 Container STL - accesso unificato

I container STL (o contenitori di dati / struttre dati) implementano un sistema di accesso unificato, comune a tutte le classi. Questo permette di trattare i diversi container in maniera generica ed è possibile disaccoppiare la struttura dall'algoritmo. Questo meccanismo è implementato attraverso sequenze e iteratori.

Iteratori

Gli iteratori dell'accesso unificato sono puntatori agli elementi di un container.

- Per ottenere un iteratore, si usano le funzioni `begin()` (iteratore al primo elemento) o `end()` (iteratore all'elemento successivo dell'ultimo elemento)
- Hanno un set di operazioni che permettono di muoversi all'interno della struttura (`++`, `--`), di eseguire confronti (`==`, `!=`) e di accederne agli elementi (`*`).
- gli iteratori sono implementati attraverso template (efficienza in esecuzione)
- spesso il tipo degli iteratori appesantisce il codice, per cui si utilizza la parola chiave `auto` per lasciare al compilatore la scelta del tipo da inserire

Quando si usano gli iteratori è necesassario fare attenzione ad alcune funzioni eseguite sul container. Ad esempio quando si rimuove un elemento da un vettore, viene eseguito lo shift dei valori, ma gli iteratori non cambiano posizione.

17.2 Algoritmi STL

Gli algoritmi STL implementano funzioni base sui container come ricerca, ordinamento, conteggio, copia e fusione. Tutti gli algoritmi vengono implementati attraverso template e utilizzano gli iteratori sugli STL-container. Nel caso in cui mancasse qualche funzionalità negli iteratori, il compilatore segnalerà errore.

Gli algoritmi hanno una variante che prevede l'utilizzo di predicati che si sostituiscono alle condizioni di confronto o di ordinamento e permetto di personalizzare le funzioni STL in base alle proprie esigenze. Ad esempio:

```
// Funzione di ricerca senza predicato
// - It e' iteratore
// - T e' tipo di dato memorizzato nel container
template<typename It, typename T>
It find(It first, It Last, const T &val) {
    while (first != last && *first != val)
        first++;
    return first;
}

// Funzione di ricerca con predicato
// - It e' iteratore
// - Pred e' predicato
template<typename It, typename Pred>
It find_if(It first, It Last, Pred pred) {
    while (first != last && !pred(*first))
        first++;
    return first;
}
```

17.3 Predicati

I predicati permettono di personalizzare il comportamento degli algoritmi STL. Sono particolari strutture che permettono di passare delle funzioni come parametri di una funzione. Sono implementati attraverso i function object e le lambda functions.

Function object

I function object sono classi che contengono l'overloading dell'`operator()`, ovvero possono essere utilizzati come funzioni e possono essere passati per copia ad una funzione. Possono contenere al più un membro (es. valore da cercare).

```
class Larger_than {
    private:
        int v; // valore discriminante scelto in fase di costruzione
    public:
        Larger_than(int _v) : v{_v} {}
        bool operator() (int x) { return x > v; } // overloading operator()
};
find_if(vec.begin(); vec.last(), Larger_than(3)); // trova un valore maggiore di 3
```

Lambda expression

Le lambda expression sono funzioni senza nome utilizzabili come parametri di una funzione. È possibile passarle delle variabili locali e il tipo restituito può essere omesso (dedotto dal compilatore). Il passaggio per argomento è fatto ad ogni invocazione e la cattura è simile all'utilizzo di dati membro di una classe.

```
[variabili_catturate](parametri) -> tipo_restituito { return ... }
```

Name requirement objects

I name requirement objects sono strutture definite da regole dello standard `C++` per svolgere compiti. Se non vengono rispettate si potrebbe andare incontro ad errori logici (o sintattici).

```
// esempio compare
... bool operator() (const T &x, const T &y) { return x < y; } ...
```

18 RAII e smart pointers

18.1 RAII - Resource Acquisition is Initialization

Il RAII è un paradigma di programmazione in `C++` che prevede la gestione delle risorse (lettura da files, memoria dinamica) attraverso degli oggetti. È previsto che la risorsa venga acquisita con l'istanziamento del relativo oggetto e l'esecuzione del costruttore e venga rilasciata quando l'oggetto viene distrutto, con la chiamata implicita al distruttore. In questo modo, quando si esce da uno scope con risorse aperte, queste vengono chiuse automaticamente con la distruzione degli oggetti a cui sono legate.

Gestione della memoria ed eccezioni

Il problema è che per la gestione della memoria dinamica non si utilizzano oggetti, per cui bisogna prevedere che all'uscita dello scope (in qualsiasi caso) bisogna liberare la memoria allocata. Per fare ciò si può ricorrere alla chiamata di `delete` come nel caso seguente, o all'utilizzo degli shared-pointers.

```
... {
    int *p = new int[100]; // allocazione dinamica
    try {
        ...
    } catch (std::exception &e) {
        delete[] p; // deallocazione all'uscita dello scope per lancio di eccezioni
        throw; // rilancia l'eccezione catturata -> non viene gestita qui
    }
    delete[] p; // deallocazione all'uscita dello scope per normale proseguimento
}
...
```

Exception guarantees

- **basic guarantee**: attributo di una funzione che libera correttamente le risorse al lancio di un'eccezione
- **strong guarantee**: come il precedente, in più se la funzione modifica dei valori, questi non vengono modificati se viene lanciata un'eccezione
- **no-throw guarantee**: attributo di una funzione che non lancia eccezioni

18.2 Smart Pointer

Gli smart pointers sono delle classi template definite nella libreria `memory` che permettono di gestire la memoria secondo il paradigma RAII. Gestiscono automaticamente la deallocazione ed evitano i memory leak e i dandling pointers. L'overhead è limitato ed è comunque più vantaggioso di un garbage collector. Esistono due tipi di smart pointers: gli `unique_ptr` e gli `shared_ptr`.

Unique pointers

- non permette di avere più copie, ovvero non ci possono essere più puntatori che puntano alla stessa area di memoria, ma permette di eseguire move
- l'inizializzaizione può avvenire con costruttore o tramite la funzione reset (assegnamento)
- l'accesso all'area puntata avviene tramite operatori `*` e `->`
- il rilascio del puntatore restituisce il puntatore senza le protezioni di uno smart pointer

```
int *p = new int; | unique_ptr<int> up {p}; // inizi da puntatore
unique_ptr<int> up { new int }              // iniz. con allocazione dinamica
unique_ptr<int> up = make_unique<int>();    // iniz. con make_unique piu' sicura
int *p = up.release();                      // rilascio del puntatore
```

Shared pointers

A differenza degli `unique_ptr`, questi accettano più copie dello stesso puntatore. Hanno un reference counter e quando l'ultimo `shared_ptr` esce dallo scope, viene deallocata la memoria.