Parte 1

Espressioni, lettura e assegnazione

Impariamo ad acquisire familiarità con alcuni elementi base della programmazione: dichiarazioni, assegnazioni, e input-ouput.

Cheatsheet

`//commento`	riga di commento
`/*commento */`	commento
`10`	costante intera (letterale)
`int a;`	dichiara una variabile di tipo `int`
`int a = 10;`	dichiara una variabile di tipo `int` con valore iniziale
`float x = 3.14159;`	dichiara una variabile di tipo `float` con valore iniziale
`char c;`	dichiara una variabile di tipo carattere
`a = 10;`	assegna un valore alla variabile `a`
`a = b;`	assegna alla variabile `a` il valore della variabile `b`
`b+1;`	espressione, ha un valore
`a = b+1;`	assegna valore dell'espressione `b+1` alla variabile `a`
`"ciao"`	stringa costante (letterale)
`'c'`	carattere costante (letterale)
	Errore comune: usare singoli apici per una stringa di più caratteri
`true  false`	valori logici costanti (letterali)
`cin >> a`	legge variabile `a` da standard input (tastiera)
`cin >> a >> b`	legge variabili `a` e `b` da standard input
`cout << a`	scrive variabile `a` su standard output (schermo)
`cout << a << " " << b`	scrive variabili `a` e `b` su standard output con spazio in mezzo
Operatori aritmetici	*Operandi: tipi numerici, risultato espressione: un tipo numerico*
`+ - * /`	operazioni aritmetiche elementari (potenza non esiste)
`%`	operatore *modulo*, calcola il resto della divisione intera
Operatori di confronto	*Operandi: tipi numerici o altri, risultato espressione: `bool`*
`==`	operatore di confronto di uguaglianza
`!=`	operatore di confronto di disuguaglianza
`< <= > >=`	operatori di confronto d'ordine
Operatori logici	*Operandi: `bool`, risultato espressione: `bool`*
`!`	negazione (unario, vuole un solo operando)
`&&`	connettivo logico "AND", `true` se entrambi operandi sono `true`
`||`	connettivo logico "OR", `true` se almeno un operando è `true`

1.1 Esercizi di riscaldamento

1. Scrivere un programma che legge due interi e ne stampa la somma.

   (a) Seguire l'algoritmo proposto (che fissa una serie di dettagli ulteriori). Il programma deve essere scritto in un file chiamato `sum.cpp`.

Parte 2

Scelte condizionali

Impariamo ad utilizzare i costrutti di scelta condizionale `if-else` e di scelta multipla `switch`.

Cheatsheet

code-block = istruzione; *oppure* { sequenza-di-istruzioni }

```
if ( espressione-booleana )
  code-block
```

```
if ( espressione-booleana )
  code-block-1
else
  code-block-2
```

N.B.: `else` non vuole una espressione booleana!

```
switch ( espressione ) {
   case valore-1:
    sequenza-di-istruzioni
    break;
   case valore-2:
    sequenza-di-istruzioni
    break;
   default:
    sequenza-di-istruzioni
}
```

Tipi primitivi:

CATEGORIA	NOME	LUNGHEZZA
Tipi interi	`int`	non specificato, tipic. 4 byte
	`long int` (o solo `long`)	>= `int`, tipic. 8 byte
	`short int` (`short`)	<= `int`, tipic. 2 byte
	`char`	1 byte
	Tutti possono essere anche `unsigned`	
Tipi floating point (reali)	`float`	32 bit (4 byte)
	`double`	64 bit (8 byte)
	`long double`	80 bit (10 byte)

Altri tipi comuni:

Tipo `std::string`	(stringhe “stile C++”) NON È UN TIPO PRIMITIVO: infatti occorre aggiungere in testa al file `#include <string>`
Costanti di tipo “stringa”	(stringhe “stile C”) SOLO PER COSTANTI LETTERALI, NON VARIABILI Esempio: `"Hello, world!"`

Parte 3

Cicli

Impariamo ad utilizzare i cicli `for`, `while`, e `do while`, per richiedere in modo efficiente l'esecuzione di un gruppo di istruzioni per più di una volta.

Cheatsheet

`++i` `--i` pre-incremento/pre-decremento (prima si incrementa/decrementa, poi si calcola il valore risultante)
`i++` `i--` post-incremento/post-decremento (prima si calcola il valore, poi si incrementa/decrementa)

Ciclo `for` "preconfezionato" per ripetere `N` volte un blocco di codice:

```
for ( i = 0; i < N; ++i )
   code-block
```

Ciclo `for` in generale:

```
for ( istruzione-1 ; espressione-booleana; istruzione-2 )
   code-block
```

Chi fa cosa:
`istruzione-1`: inizializzazione
`espressione-booleana`: test di terminazione
`istruzione-2`: avanzamento

Ordine esecuzione:
`istruzione-1` → `espressione-booleana` → `code-block` → `istruzione-2` ⟶ ↑

Ciclo while

```
while ( espressione-booleana )
   code-block
```

Ciclo do. . . while

```
do
   code-block
while ( espressione-booleana );
```

N.B.: entrambi terminano quando `espressione-booleana` vale `false`

Quale costrutto devo usare? Regola di prima approssimazione:

Conosco il numero di iterazioni da effettuare	⟶	`for`
Posso verificare una condizione di arresto prima di iniziare il ciclo	⟶	`while`
Posso verificare una condizione di arresto ma solo alla fine del ciclo	⟶	`do ... while`

Tecnica del look-ahead

```
leggi-input
while(input ok) {
  fai-qualcosa-su-input
  leggi-input // (successivo)
}
```

Parte 4

Array

Impariamo ad utilizzare gli array.

Cheatsheet	
`float x[N];`	Dichiara un array di `float` di lunghezza `N`
`int x[4]={1,2,3,4};`	Dichiara e inizializza un array di 4 interi
	N.B. Anche `x[]={1,2,3,4};` `x[8]={1,2,3,4};` (inizializza i primi 4 e azzera il resto)
	N.B. `x[2]={1,2,3,4};` ⟶ errore "too many initializers"
`x[i]`	Accede all'i-esimo elemento dell'array `x`
	Se `i<0` o `i>=N` dà segmentation fault!
	errore comunissimo, verificate sempre la validità dei valori degli indici!!
`x[i] = 1;`	Imposta un valore per l'elemento i-esimo ("scrittura")
`a = x[i];`	Usa valore dell'elemento i-esimo ("lettura")

Non si possono applicare operatori aggregati, ad es. no `array1 = array2`, no `cout << array`, no `array = 0`.

4.1 Esercizi di riscaldamento

1. **Stampa un array di `int`.** Scrivere un programma che, dati un array `a` di `int` e la sua lunghezza `N`, stampa tutto l'array. [File `stampaArrayInt.cpp`]

```
// Creare e popolare un array a di lunghezza 7.
int a[7]={2, 4, 34, 78, 4, 3, 876};
// Iterare sulla variabile intera i a partire da 0 e fino a 7 escluso:
// - Stampare il valore i-esimo dell'array in posizione i e a capo
```

   Output atteso:

```
Valore di a[0] = 2
Valore di a[1] = 4
Valore di a[2] = 34
Valore di a[3] = 78
Valore di a[4] = 4
Valore di a[5] = 3
Valore di a[6] = 876
```

2. **Stampa inverso di un array di `float`.** Scrivere un programma che lavora su un array di `float` e li stampa in ordine inverso [File `stampaArrayFloat.cpp`]

```
// Creare e popolare un array a di lunghezza 5.
float a[5]={2.4, 5.67, 34, 28.456, 846.42};
// Iterare sulla variabile intera i a partire da N-1 (4) e fino a 0 incluso:
```

Parte 5

String

In questa sezione ci concentriamo sull'utilizzo degli string tramite il tipo `std::string` (cf https://cplusplus.com/reference/string/string/).

Cheatsheet	
`#include <string>`	header per i `std::string`
`using namespace std;`	per evitare di ripetere sempre `std`
`string s`	Dichiara la variabile `s` di tipo `string`
`string s="Hello"`	Crea una `string s` e la inizializza con `"Hello"`
`s.length()`	Ritorna la lunghezza della string `s` (cioé il numero di caratteri che contiene)
`s=s+"Hello"`	Concatena (aggiunge) `Hello` alla fine della string `s`
`s=s+'A'`	Concatena (aggiunge il carattere) `A` alla fine della string in `s`
`string t="Hello"+"World"`	**NON FUNZIONA!**
`string t=string("Hello")+string("World")`	Funziona
`s[i]`	Accede all'elemento i-esimo della string (con`0<=i<s.length()`)
`getline (cin,s)`	Legge una string finita con A capo da input e la memorizza nella variabile `s`
`s.substr(i,k)`	Estera da `s` la string con al massimo `k` caratteri iniziando dalla posizione `i`
`s.erase(i,k)`	Cancella da `s` al massimo `k` caratteri iniziando dalla posizione `i`

5.1 Esercizi di riscaldamento

1. Scrivere un programma che chiede all'utente di inserire una string e poi stampa il numero di spazi nella string.

   [File `numberSpaces.cpp`]

```
// dichiarare una variabile string st
// leggere st da input con getline
// dichiarare una variabile int count e inizializzarla a 0
/* iterare su i a partire da 0 e fino a st.length()
   - se il carattere alla posizione corrente (i) e' uguale a ' '
   -- aumentare count di 1
*/
// scrivere su output: st " ha " count " spazi!"
```

2. Scrivere un programma che chiede all'utente di inserire una string e verifica se questa string rappresenta un numero positivo (deve contenere solo cifre e non iniziare con 0).

   [File `isNumber.cpp`]

```
// dichiarare una variabile string st
// leggere st da input con getline
// dichiarare una variabile bool isNumber e inizializzarla a true
// se st contiene almeno un carattere
//     se il primo carattere è diverso di 0
```

Parte 7

Struct

Impariamo ad utilizzare il tipo di dato struct, che ci permette di aggregare dati sia omogenei (stesso tipo) che non omogenei.

Cheatsheet

Creazione del *tipo* `tipo-struttura`:

```
struct tipo-struttura {
   dichiarazione-membro-1;
   dichiarazione-membro-2;
   dichiarazione-membro-3;
};
```

Le dichiarazioni sono normali dichiarazioni di variabili. Le variabili-membro si possono usare individualmente.
N.B.: Qui le dichiarazioni non possono contenere inizializzazioni!
N.B.: Un membro può a sua volta essere di un tipo struttura!

Dichiarazione di una *variabile* di tipo `tipo-struttura`:

```
struct tipo-struttura nome-variabile;
```

oppure

```
tipo-struttura nome-variabile;
```

N.B.: Qui `struct` è opzionale.

Uso dei membri:

```
data.giorno = 25;
data.mese = 12;
data.anno = 800;
if(data.giorno == 1) std::cout<<"Oggi inizia un nuovo mese\n";
```

Nota: No operazioni aggregate:
no `cout << data`, no `data++`

Però **OK assegnazione**: `data1 = data2`

Usare funzioni matematiche: in testa al file aggiungere `#include <cmath>`

`fabs(x)`	Valore assoluto (float abs)
`sqrt(x)`	Radice quadrata
`exp(x)`	Esponenziale in base e
`pow(x,y)`	Potenza (power), x^y
`log(x)` `log2(x)` `log10(x)`	Logaritmo in base e, 2, 10
`sin(x)` `cos(x)` `tan(x)`	Funzioni goniometriche
`ceil(x)` `floor(x)`	Arrotonda a intero per eccesso/per difetto

Operano tutte su dati di tipi `double` (anche `float` che viene convertito automaticamente in `double`); restituiscono `double`.

7.1 Esercizi di riscaldamento

1. Definire un tipo struct `Person` per rappresentare i dati relativi a una persona:

```
struct Person {
    std::string name;
    std::string surname;
    int birthYear;
};
```

   Dichiarare due variabili di tipo `Person`:

Parte 8

Funzioni

Impariamo a scrivere funzioni, l'elemento-base per costruire un programma complesso partendo da operazioni più semplici.

Per ciascuna funzione sarà necessario anche scrivere un `main` per poterla sottoporre a dei test ("testare"), ovvero chiamare con argomenti noti per verificare che il risultato restituito sia quello atteso.

NOTA. Ritroverete in questa parte molti esercizi che avete già svolto senza usare funzioni. D'ora in avanti, tenete presente che alcune funzioni sviluppate in un esercizio possano essere utili allo svolgimento di un esercizio successivo. Cercate di riutilizzare (ossia di richiamare) le funzioni già scritte, quando è appropriato.

Cheatsheet

Una funzione riceve in ingresso n **argomenti** o **parametri** (con n che può anche essere 0, nessun argomento) e **restituisce** in uscita un valore, oppure niente. Il tipo del valore restituito va dichiarato; nel caso "niente", il tipo è `void`.

Dichiarazione:
Esempio

```
int funz(float);
```

- Una funzione viene dichiarata scrivendo il suo **prototipo**
- Un prototipo può non contenere i nomi degli argomenti, ma deve contenerne il tipo. L'ordine conta.
- Un prototipo si può ripetere più volte nello stesso file (purché sempre identico, altrimenti rappresentano funzioni diverse)
- Solo un file in cui compare un prototipo può chiamare la corrispondente funzione

Definizione:
Esempio

```
int funz(float x) // intestazione
{
  // corpo = un code-block
}
```

- Una funzione viene definita scrivendo il suo codice
- Il codice di una funzione contiene una intestazione (simile a un prototipo) e un corpo
- Nell'intestazione, i nomi e i tipi degli argomenti **sono necessari**. L'ordine conta.
- Una funzione si deve definire una volta sola, pena errore
- All'interno di un sorgente, dopo che una funzione è stata definita si può usare anche senza prototipo. Se voglio usarla prima, devo aggiungere un prototipo prima.

Invocazione o chiamata:
Esempi:

```
a = funz(y);
a = funz2(x,y)+z;
std::cout << funz3() << std::endl;
```

MA ATTENZIONE: `funz4(x) = a; //ERRORE!`

- Un'invocazione di funzione è una espressione
- Una funzione viene chiamata con il nome seguito, tra parentesi, da tutti gli argomenti nell'ordine definito
- Più argomenti separati da virgole: es. `funz(x,y)`
- Se zero argomenti: parentesi vuote, ma comunque necessarie: es. `funz()`
- Una chiamata di funzione è ammessa dove è ammessa *in lettura* una variabile del tipo restituito dalla funzione
- Una chiamata di funzione **non** si può usare *in scrittura*, ovvero non le si può assegnare un valore. Non è un *lvalue*

Errori comuni:

- `return x;` in funzione che restituisce `void`
- `return;` in funzione che restituisce un tipo non-`void`
- omettere `return x;` in funzione che restituisce un tipo non-`void`
- dichiarare un argomento nella intestazione della funzione, e poi dichiarare una variabile con lo stesso tipo e nome *anche all'interno* del corpo della funzione
- leggere da `cin` il valore di un argomento di input
- modificare argomento dichiarato `const`
- passare argomenti in ordine sbagliato
- sperare che le modifiche fatte a un argomento (non passato come reference) siano conservate nel programma chiamante (approfondimento prossima parte eserciziario)

Parte 10

Puntatori (no allocazione dinamica di memoria) + Compilazione separata

In questa sezione vedremo l'uso dei puntatori per accedere ad aree di memoria *già allocate altrimenti*, ad esempio variabili e parametri di funzioni.

Cheatsheet

» Puntatori «

- Un **puntatore** è una variabile atta a contenere come valore l'indirizzo di un'altra variabile. Un puntatore è sempre associato ad un tipo `T`.
- Definizione di una variabile di tipo "puntatore a un oggetto di tipo `T` ":
  - senza inizializzazione: `T* p;` Esempio: `int* p;`

    Meglio inizializzare le variabili, perché rischia di usarne il valore senza avergliene assegnato uno prima. Nel caso dei puntatori questo vuol dire accedere ad un'area di memoria a caso.
  - con inizializzazione: `T* p = <espressione-di-tipo-puntatore>;`

    Esempi: `char* q = p;` oppure `char* p = nullptr;` (`nullptr` è il puntatore nullo)

  Attenzione: Usare dichiarazioni separate per variabili separate: `T* p; T* q;`
- **Operatore di referenziazione &** (detto anche operatore "indirizzo di"). Data una variabile restituisce un valore indirizzo, corrispondente all'indirizzo in memoria della variabile.

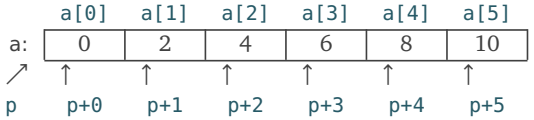
  Esempio: `&a` permette di ottenere l'indirizzo di `a`. `float* p = &a;` dove `a` è una variabile di tipo `float`.

  Se abbiamo definito `T* p;` per assegnarle un valore useremo `p = &a;`.

  L'operatore di referenziazione `&` produce sempre un valore destro (l'indirizzo della variabile che non posso cambiare).
- **Operatore di dereferenziazione** `*`. Dato un puntatore, restituisce la variabile puntata dal puntatore.

  Esempi: `*p = 2;` (scrive 2 nell'area puntata da `p`); `a = *p + 4` (legge il valore contenuto nella memoria puntata da `p`, le somma 4 e assegna il risultato ad `a`)

  L'operatore di dereferenziazione `*` può essere usato sia come valore destro (il contenuto della variabile puntata da ...) che come valore sinistro (la variabile stessa).
- Dimensione in memoria di qualcosa che ha tipo `T`: `sizeof T`. esempio `sizeof int` (dimensione in bytes di un `int`) oppure `sizeof *int` (dimensione in bytes di un puntatore a `int` cioè di un indirizzo di memoria)
- Dimensione in memoria di una variabile `a`. esempio `sizeof(a)`
- **Aritmetica dei puntatori**: se `p` è un puntatore a un oggetto di tipo `T` ed `n` è un intero, allora `p+n` è l'indirizzo che si ottiene sommando `n` volte `sizeof(T)` all'indirizzo contenuto in `p`. Corrisponde a spostarsi, rispetto all'indirizzo puntato da `p`, di `n` elementi di tipo `T`. Esempio: `int a[6] = {0, 2, 4, 6, 8, 10};` `int* p = a;` (NB: un array è considerato la stessa cosa di un puntatore al primo elemento nell'array) si avrà:



dove ogni cella occupa quanto un intero = `sizeof int`.

Quindi l'istruzione `*(p+3)=7;` scrive 7 nella cella di indice 3 di `a`, sovrascrivendo il 6.

Analogamente, `cout << *(p+4);` stampa il contenuto della cella di indice 4 di `a`, ovvero stampa 8.

Uso frequente: usare un puntatore per scorrere un array, ad esempio per stamparlo:

```
for(int i=0;i<6;i++) cout<< *p++ <<endl;
```

N.B. `*(p++)` equivale a `*p++` perché l'operatore `++` ha la precedenza sul `*`

- **Passaggio di Array a Funzioni**: Il C++ non consente di passare un intero array come argomento di una funzione. Tuttavia, è possibile passare un puntatore all'array di tipo T e la dimensione dello stesso ad esempio nei seguenti modi:

```
void function(T* array, int size) { //corpo funzione }
void function(T array[], int size) { //corpo funzione }
```

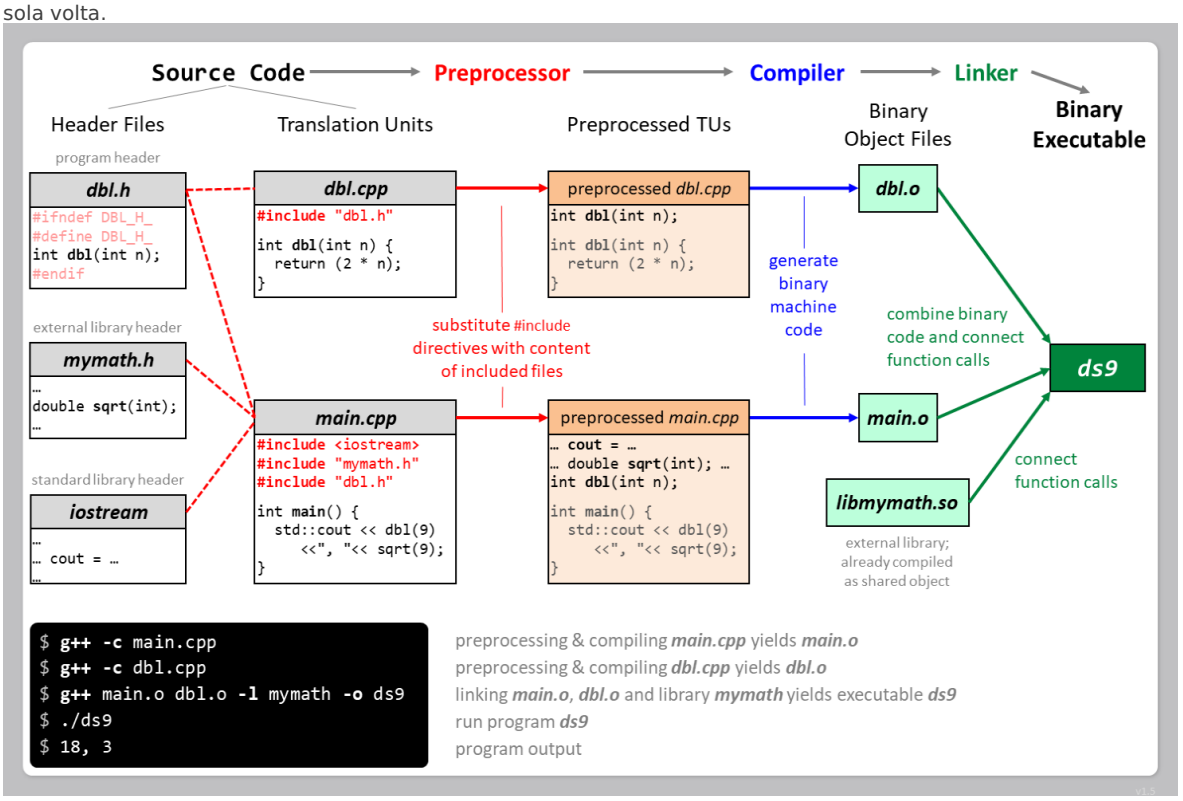
» Compilazione Separata «

Organizzare il codice di un programma in *file separati* rende lo stesso più facile da navigare e mantenere, e inoltre promuove la riutilizzabilità del codice. Le funzioni definite in diversi file sorgente (`.cpp`) possono essere *riutilizzate* in un altro programma includendo il relativo file di intestazione (`.h`). Questo favorisce la *creazione di componenti modulari e riutilizzabili*, che è fondamentale per ottenere un buon design del software.

Suddividere un programma in diversi file sorgente consente anche di effettuare la **compilazione separata**. La compilazione separata ottimizza il processo di compilazione. Infatti, quando si modifica un file sorgente e si ricompila, è sufficiente ricompilare solo quel file, e non l'intero programma. Questa strategia di compilazione è particolarmente vantaggiosa per progetti di grandi dimensioni, dove ricompilare tutto per una piccola modifica è molto dispendioso.

Creazione di File di Intestazione: In C++, i file di intestazione sono un componente chiave della compilazione separata. Contengono ad esempio i prototipi delle funzioni, le dichiarazioni delle struct etc, che informano il compilatore sulla struttura del codice senza rivelarne i dettagli implementativi. I file di intestazione solitamente hanno un'estensione `.h` e vengono inclusi nei file sorgente con la direttiva `#include` (vedi esempio sotto). Le Include Guard `#ifndef X #define X ... #endif` sono una parte fondamentale della creazione di file di intestazione. Impediscono che lo stesso file di intestazione venga incluso più di una volta all'interno di una singola unità di compilazione. Questo è cruciale perché includere un file più volte può causare errori di ridefinizione.

Per dettagli aggiuntivi analizzare la seguente figura dove è mostrato anche come compilare separatamente i vari files. Notare che la direttiva `#include "dbl.h"` è presente anche nel file `dbl.cpp` che contiene l'implementazione della funzione. In questo caso non sarebbe strettamente necessaria in quanto il `.h` contiene solo il prototipo della funzione (mentre è necessaria quando il `.h` contiene ad esempio la definizione di struct usate poi nelle funzioni). Aggiungere la direttiva `#include` del file `.h` nel file delle funzioni `.cpp` è considerata una buona pratica per evitare alcuni tipi di errori (ad esempio: disallineamento tra nomi/parametri delle funzioni tra il `.h` e il `.cpp`: con l'include vengono già rilevati durante la compilazione separata). Inoltre, se le funzioni nel `.cpp` si richiamano fra loro è necessario che i prototipi siano noti prima delle rispettive chiamate, quindi metterli tutti all'inizio includendo il `.h` lo garantisce. Un prototipo può comparire più volte (in quanto è una dichiarazione senza definizione) a differenza della definizione che deve comparire una sola volta.



Parte 11

Puntatori - allocazione dinamica di memoria + Valgrind

In questa sezione vedremo l'uso dei puntatori per *allocare dinamicamente memoria* e lavorarci.

Cheatsheet

Operazioni per allocare (ovvero riservare) memoria (su heap) e rilasciarla quando non serve più.

- **Per allocare la memoria necessaria per memorizzare un singolo elemento di tipo `T` si usa `new T`**

  Quindi ad esempio per allocare la memoria necessaria per memorizzare un `float` si usa `new float`, per allocare la memoria necessaria per memorizzare un `int` si usa `new int` e così via.

  Per evitare di allocare memoria a cui non si può accedere (che quindi sarebbe sprecata), quando si usa `new` bisogna sempre associare la memoria che si sta riservando ad un puntatore. Esempi tipici di uso:

  - *dichiarazione di puntatore con inizializzazione a una nuova area di memoria:* `T* p = new T;`

    Ad esempio nel caso `T` sia `float` avremo `float* p = new float;`

    nel caso `T` sia `int` avremo `int* p = new int;` e così via.
  - *assegnazione di una nuova area di memoria ad un puntatore già dichiarato:* `T* p; p = new T;`

    Ad esempio nel caso `T` sia `float` avremo `float* p; p = new float;`

    nel caso `T` sia `int` avremo `int* p; p = new int;` e così via.

- **Per allocare un blocco di memoria necessaria per memorizzare `N` elementi di tipo `T` si usa `new T[N]`**

  Ad esempio per allocare la memoria necessaria per memorizzare `7` `float` si usa `new float[7]`, per allocare la memoria necessaria per memorizzare 5 `int` si usa `new int[5]` e così via.

  Come nel caso in cui si riserva la memoria per un singolo elemento, anche quando si riserva un blocco contiguo per memorizzarvi `N` elementi bisogna sempre associare la memoria che si sta riservando ad un puntatore; il puntatore a cui viene assegnata l'area allocata (sia in fase di dichiarazione o con una assegnazione successiva) punta al primo elemento del blocco. Esempi tipici di uso:

  - *dichiarazione di puntatore con inizializzazione a una nuova area di memoria:* `T* p = new T[N];`

    Ad esempio nel caso `T` sia `float` e `N == 4` avremo `float* p = new float[4];`

    nel caso `T` sia `int` e `N == 11` avremo `int* p = new int[11];` e così via.
  - *assegnazione di una nuova area di memoria ad un puntatore già dichiarato:* `T* p; p = new T[N];`

    Ad esempio nel caso `T` sia `float` e `N == 9` avremo `float* p; p = new float[9];`

    nel caso `T` sia `int` e `N == 2` avremo `int* p; p = new int[2];` e così via.

  Bisogna fare molto attenzione a non perdere l'indirizzo iniziale di un blocco che si è allocato. Ad esempio se si vuole usare un puntatore per scorrerlo non si può usare quello impiegato per allocare la memoria (quanto meno non prima di averlo salvato altrove).

- **Per deallocare, ovvero rilasciare, la memoria precedentemente allocata con una `new` si usa `delete p` o `delete[] p`:**
  - `delete p`, dove `p` è il puntatore a cui è stata assegnata la memoria allocata, se è stato riservato spazio per un singolo elemento.
    Ad esempio `float* p = new float; delete p;` oppure `int* p; p = new int; delete p;`
  - `delete[] p`, dove `p` è il puntatore a cui è stata assegnata la memoria allocata, se è stato riservato spazio per più elementi. Ad esempio `float* p = new float[6]; delete[] p;`
    oppure `int* p; p = new int[3]; delete p[];`

Note

- Quando si crea *aliasing* fra puntatori, bisogna fare molta attenzione. Ad esempio, se facciamo
  `float* p = new float; q = p; /*...*/delete p;`
  - poiché è un errore fare due volte la `delete` della stessa area di memoria, l'istruzione `delete q;` causa errore.
  - il puntatore `q` punta ad un'area di memoria non più riservata. Se questa memoria viene riassegnata ad altri, usare `*q` può dare un errore, ma non è molto probabile. Nella maggior parte dei casi se la si usa per leggere avremo risultati inattesi (qualcuno ci ha scritto altri valori) e se la si usa per scrivere si modificano dati su cui stanno lavorando altri, causando verosimilmente errori in altre parti del programma.
- Dopo aver rilasciato la memoria puntata da un puntatore `p`, il puntatore si trova in uno stato pericoloso, perché punta ad un'area di memoria non più riservata per il nostro programma. È quindi buona norma seguire immediatamente una `delete` con un'assegnazione (eventualmente a `nullptr` se non si vuole riusare `p` con altro valore), a meno che `p` non sia una variabile locale che sta per essere eliminata all'uscita da uno scope (ad esempio la `delete` è l'ultima riga di una funzione e `p` è una variabile locale a quella funzione).

Valgrind è uno strumento per il debug di problemi di memoria e la ricerca dei memory leak. Per usarlo per verificare un programma C++ si possono seguire questi due passi:

1. Compilare il programma attivando le informazioni utili per il debug del programma (opzione `-g`): questo consente a Valgrind di fornire informazioni riguardo a quale LOC contiene la dichiarazione dell'oggetto che è stato coinvolto nel memory leak. Ad esempio:
   `g++ -Wall -std=c++14 -g leaks.cpp`
2. Eseguire poi Valgrind con il comando:
   `valgrind --leak-check=full --track-origins=yes -s ./a.out`
   Analizzare i risultati ottenuti.

Nota. Valgrind (http://www.valgrind.org/) ***rileva i memory leak solo sul codice effettivamente eseguito durante l'analisi***. Quindi se ad esempio un memory leak si trova in un ramo di un `if` che non è eseguito durante la verifica quel memory leak non sarà riportato. E' quindi importante ***cercare di testare in modo esaustivo i vari casi*** in modo da eseguire (coprire) il più possibile le varie porzioni del codice del vostro programma: in generale, pensate ad un insieme di test con input differenti che permettono di coprire i vari statement del vostro programma (criterio denominato ***Statement Coverage Testing***).

11.1 Esercizi di riscaldamento

1. Scrivere un programma che implementa il seguente algoritmo:

```
// dichiarare una costante N intera inizializzandola a un valore moderato (es. 5 o 10)
// dichiarare una variabile v di tipo puntatore a int
// allocare una quantità di memoria pari a N int, assegnandola a v
// scrivere nella memoria puntata da v la sequenza di valori  1, 3, 5, ... , 2*N-1 (i primi N dispari)
// stampare v usando l'aritmetica dei puntatori
// deallocare v
// allocare una quantità di memoria pari a 2*N int, assegnandola a v
// scrivere nella memoria puntata da v la sequenza di valori 1, 3, 5, ... , 4*N-3 (i primi 2*N dispari)
// stampare v usando l'aritmetica dei puntatori
// deallocare v
```

Nota. Fate attenzione a non perdere il riferimento a `v` nell'inizializzarne il contenuto o nella stampa. [File `alloc.cpp`]

Parte 12

Liste e ripasso/approfondimento su Librerie

In questa sezione vedremo vari esercizi sulle Liste come spiegate a lezione e organizzeremo i file in librerie.

Ripasso/approfondimento su Librerie

Negli esercizi di questa sezione sarà richiesto di creare *librerie*. Si tratta quindi di esercizi sulla modularità: le funzioni che forniscono gli strumenti per risolvere un problema specifico vengono raccolte in un unico *modulo* = un file sorgente.

Chi usa tali funzioni deve naturalmente avere accesso alle definizioni (tipi, prototipi...). Queste sono però specificate in un file header, l'unico file che l'utente ha bisogno di conoscere. Le funzioni sono così *incapsulate* in un componente che può anche essere compilato separatamente: in tal caso non serve nemmeno più avere il file sorgente.

La modularità (file separato) rende più facile riusare il codice. L'incapsulamento (non conoscere il sorgente del modulo) rende possibile modificare l'implementazione senza dover modificare i programmi che le usano. Questo è utile in caso di errori, aggiornamenti tecnologici, miglioramenti e arricchimenti... Inoltre incapsulare le informazioni serve a nascondere i dettagli implementativi che non interessano chi usa le funzionalità fornite (*information hiding*).

Cheatsheet

Definire una libreria – si separa la sua *interfaccia*, ovvero tipi e prototipi di funzioni, dall'*implementazione*.
Promemoria: chi usa la libreria deve vedere solo l'interfaccia, chi la programma vede anche l'implementazione.

- Interfaccia ⟶ file *header* con estensione `.h`

  Cosa si può inserire in un file header? Solo **dichiarazioni** che è ammesso ripetere in più file, ovvero:

  - prototipi di funzioni
  - dichiarazioni di tipo come struct o enum
  - typedef

  NON si possono inserire né **corpi di funzioni** né **dichiarazioni di variabili/costanti**.
- Implementazione ⟶ file `.cpp`

  Cosa si può inserire in un file sorgente (.cpp)? **Definizioni**, cioè:

  - corpi di funzioni (ovviamente incluso `int main()`)
  - variabili/costanti globali

  Possono esserci anche **dichiarazioni** locali al file, cioè che vengono usate nell'implementazione ma non servono all'utilizzatore.

Usare una libreria

- Il file header va incluso in tutti i file che hanno bisogno di usare la libreria:

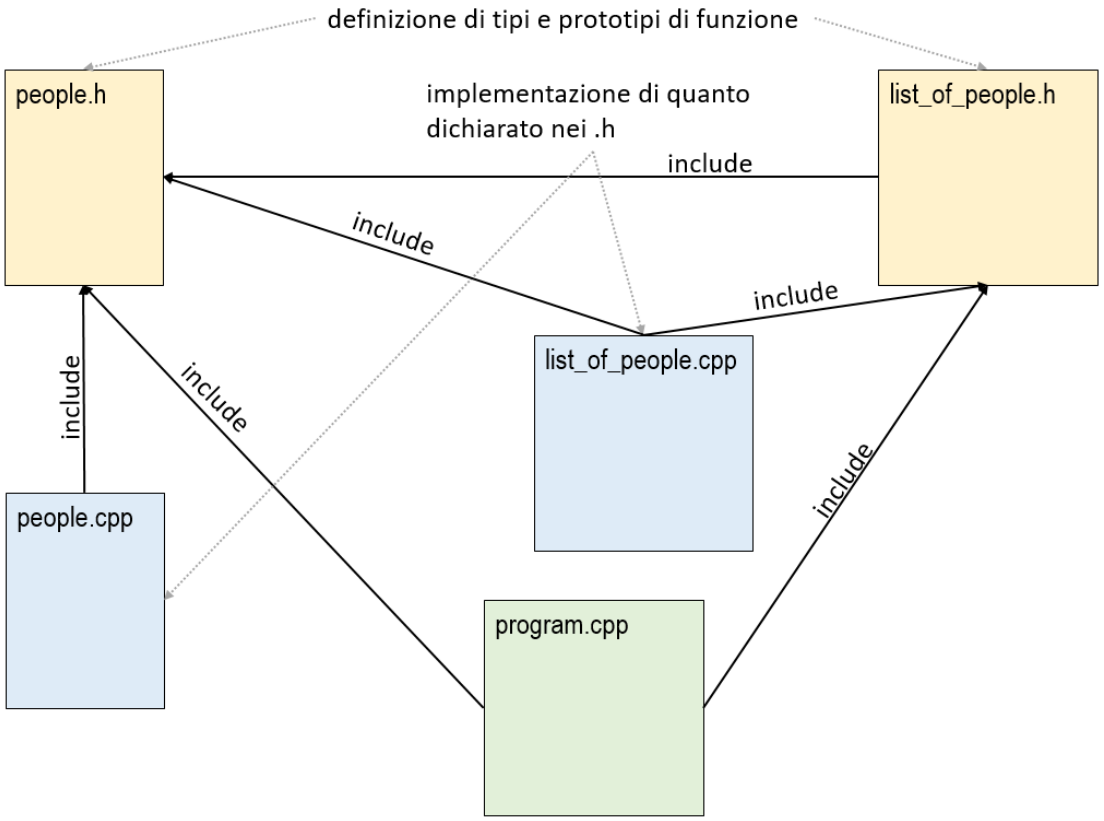
  - **sempre** nel file (`.cpp`) che contiene il programma che usa la libreria
  - spesso anche nel file che contiene l'implementazione (come buona pratica conviene inserirlo sempre)
  - eventualmente nei file di header di altre librerie basate su questa
    Esempio: la mia libreria usa il tipo `std::string` → il mio header include `<string>`.

  Promemoria: Dove uso una libreria, includo il relativo header; dove non ne uso neanche una, non devo includerlo. Non serve includere tutto ovunque.
- Nel comando di compilazione bisogna elencare tutti i file che fanno parte del vostro progetto: il file .cpp che contiene il main e tutti i file che contengono l'*implementazione* delle librerie che usate (.cpp se avete i sorgenti, .o se sono già compilati).

NON bisogna compilare i file header perché vengono automaticamente copiati all'interno dei .cpp.

Le librerie di sistema sono sottintese e linkate automaticamente; eventuali librerie non standard vanno invece specificate.

Ecco un esempio tipico di uso di librerie (un programma per gestire un'anagrafe, ad esempio, oppure il registro degli esami... la libreria *liste di persone* potrebbe essere riusata per molti scopi, corrispondenti a diversi `program.cpp`):



Notate che nell'esempio (come in molti casi concreti), i file `program.cpp` e `list_of_people.cpp` includono due volte il file `people.h` una volta direttamente e una indirettamente attraverso l'inclusione di `list_of_people.h`. Questo è un'inutile perdita di tempo, perché vengono aggiunte due volte le stesse cose. Inoltre, in casi complicati, è possibile che si creino *cicli* di inclusione che causano errore in compilazione. Per evitare questo problema, bisogna dare ai propri file .h questa struttura:

```
#ifndef NOME_VOSTRO_FILE
#define NOME_VOSTRO_FILE
/*tutto il codice del .h */
#endif
```

In questo modo la prima volta che viene processata l'inclusione del file la costante `NOME_VOSTRO_FILE` non è definita e il codice viene copiato tutto. Se si include una seconda volta lo stesso file, `NOME_VOSTRO_FILE` è già stata definita (la prima volta che lo si è incluso) e quindi si salta il blocco e non si copia nulla.

Inoltre, nello header:

- **NON** inserire #include, se non sono necessari a **tutti** i file che useranno l'include
- **NON** inserire using namespace (se necessario usare qualificatore std::, per esempio per string e vector)

NOTA. La direttiva `#include` va usata solo per includere file header `.h`, non per incorporare altri file sorgenti `.cpp`!! Tecnicamente funziona, ma è vietato dalle buone pratiche di programmazione.