Tipi di dati

- Tipi fondamentali
- Puntatori e reference, nullptr
- Array
- Stack e free store (heap)
  - Operatori new e delete
- Codice di esempio
  - https://github.com/egalli64/corso-cpp folder b2

Variabili

- Le variabili sono usate per memorizzare valori di un determinato tipo
  - Ogni variabile dichiarata dal programmatore in C++ ha
    - Un tipo di dato associato
    - Un nome
    - Un valore
- Azioni sulle variabili
  - Dichiarazione: si comunica al compilatore che useremo una variabile di dato tipo e nome
  - Inizializzazione: si mette un valore iniziale in una variabile precedentemente dichiarata
  - Definizione: combinazione in un unico statement di dichiarazione e Inizializzazione
  - Assegnamento: si mette un valore in una variabile già inizializzata
  - Accesso: lettura di un valore contenuto in una variabile
- Non abbiamo garanzie sul valore di una variabile non inizializzata
  - Il compilatore g++ con il flag -Wall segnala come warning la non inizializzazione

sizeof

- Non abbiamo garanzia sulla dimensione di un tipo di dato
  - Può essere diversa a seconda della piattaforma usata
  - Approccio dato dalla necessità di essere vicini all'hardware
- L'operatore sizeof()
  - Ritorna la dimensione in byte del tipo a cui viene applicato
    - *È possibile applicarlo anche su di una variabile*
- Abbiamo la garanzia sulla dimensione minima di un byte
  - Tranne rare eccezioni, un byte è *esattamente* otto bit

Tipi fondamentali

- Booleano, **bool**, può assumere solo i valori **false** (0) o **true** (1, o non-zero)
- Singolo carattere, **char**, il valore letterale va indicato tra apici singoli 'a', '7', '@', ...
- Numerici interi, possono essere **signed** (default) o **unsigned**
  - **short**
  - **int** ← *uso comune*
  - **long**
  - **long long**
- Numerici reali *(floating point)*, hanno sempre il segno
  - **float**
  - **double** ← *uso comune*
  - **long double**
- Stringhe, i letterali sono rappresentate tra doppi apici, es: "Hello"
  - In C non esiste un vero tipo stringa, viene reso con un array di caratteri terminato da '\0'
  - In C++ normalmente si usa la classe **std::string**

Operatori

- Inizializzazione di una variabile per mezzo dell'operatore di assegnamento **=**
  - Se necessario, converte implicitamente il valore a destra per adeguarlo al tipo della variabile a sinistra
  - Per evitare la conversione implicita, da C++11 si preferisce **{}**, inizializzatore uniforme
- Operatori aritmetici
  - Binari +, -, *, /, %
    - In caso di operandi di diverso tipo, il compilatore converte il più "piccolo" al più "grande"
    - La divisione può essere intera o reale, dipende dagli operandi
    - Si noti l'overload dell'operatore + per std::string, concatenazione tra stringhe
  - Unari +, -
- Operatori di confronto ==, !=, <, <=, >, >=
- Operatori logici !, &&, ||
- Operatori orientati al bit (su unsigned!) &, |, ^, ~
- Altri operatori di assegnamento +=, -=, *(… etc)*, ++, --

Cast e auto

- Cast esplicito via **static_cast**<*type*>()
- Altri cast
  - **const_cast** – rimuove o aggiunge il const a una variabile
  - **dynamic_cast** – usato all'interno di una gerarchia
  - **reinterpret_cast** – assegna un nuovo senso ai byte correlati
    - Corrisponde al cast C-style (type)value
    - Da usarsi solo in assenza di alternative
- Per passare da tipi fondamentali a stringhe e viceversa
  - Funzioni specifiche: stoi(), stod(), to_string(), …
- La deduzione del tipo di una variabile può essere demandata al compilatore
  - Usando la keyword **auto** al posto del tipo della variabile nella sua definizione

Costanti

- constexpr: la valutazione del valore avviene durante la compilazione
- const: il valore non deve essere modificato
- Enumerazioni
  - enum: ereditati dal C, un nome con assegnato un valore int
    - enum { <ENUM_VALUE> [= N] [, ...] }
    - per default i valori assegnati agli enum vanno da zero in su
  - enum class: C++11, hanno uno scope e un tipo proprio
    - enum class <EnumName> [: integral_type] { <ENUM_VALUE> [= N] [, ...] }
    - per default l'integral_type di riferimento per una enum class è int

```
enum { SAT = 0, SUN };
cout << ALPHA;
```

```
enum class WeekendDay : bool { SAT = 0, SUN };
cout << static_cast<int>(GreekLetter::ALPHA);
```

Puntatori e nullptr

- Una variabile puntatore è definita in base al tipo della variabile puntata
  - Il suo tipo è il tipo della variabile indirizzata decorato con un asterisco
    - Ad esempio una variabile puntatore a carattere ha tipo char*
- Una variabile puntatore contiene l'indirizzo di un'altra variabile
  - Es: una variabile char* può avere come valore l’indirizzo di una variabile char
    - Se *“non punta"*, va segnalato usando il valore riservato **nullptr**
- L'indirizzo di una variabile è ritornato dall'operatore &
  - Se 'c' è una variabile di tipo char
  - Allora '&c' rappresenta il suo indirizzo

Reference

- Una variabile reference è un alias ad una variabile
  - Deve obbligatoriamente riferirsi a una variable
  - *(non c'è niente di simile a nullptr per un reference)*
  - Non può cambiare variabile di riferimento
- Una variabile reference è definita in base al tipo della variabile referenziata
  - Il suo tipo è il tipo della variabile indirizzata decorato con una &
    - Ad esempio, un reference a carattere è di tipo char&
- Il reference di una variabile è generato automaticamente in fase di definizione
  - Es: se 'c' è una variabile di tipo char
  - char& ref = c, definisce ref come reference a c

Array

- Un array è un blocco contiguo di celle di tipo determinato
- La dimensione di una variabile di tipo array:
  - Deve essere data al momento della sua definizione
  - Non può essere cambiata in seguito
- La dimensione di un array allocato su stack deve essere una costante
  - type array_name[size];
- Per un array locale non inizializzato esplicitamente:
  - Il compilatore ci mette a disposizione lo spazio relativo
  - *Non* fornisce alcuna garanzia sui valori assegnati alle singole celle
- I valori negli array non locali sono invece implicitamente inizializzati a zero
  - *(false, nullptr, o come specificato dal default ctor, a seconda del tipo)*

Uso di array

- Inizializzazione contestuale alla dichiarazione
  - type array_name[size]{ v0, v1, v2, v3, v4, v5 };
    - L'elemento in prima posizione ha indice 0, l'ultimo size - 1
- Non è necessario esplicitare la dimensione dell’array
  - È implicitamente pari al numero di elementi passati all’inizializzatore
  - Se indicata, deve essere maggiore o uguale del numero degli elementi passati in inizializzazione
    - Se maggiore, gli altri elementi sono inizializzati a zero *(o false, …)*
- Accesso a un elemento
  - Es: assegnamento a una variabile del valore in posizione index nell'array
  - a_variable = array_name[index] // index può assumere un valore tra 0 e size – 1
- Assegnamento di un valore a un elemento
  - array_name[index] = new_value;

Stack

- Di default, le variabili sono allocate sullo stack
  - Accesso più veloce
  - Meno overhead nella gestione
  - Automaticamente rimosse dalla memoria all'uscita dello "scope"
- La gestione della memoria è demandata al compilatore
  - Gli oggetti devono essere di dimensione statica
- Preferito l’uso dello stack, a meno che
  - Non vogliamo una gestione automatica
  - la dimensione della variabile non sia nota a compile-time

Heap

- Gestione più complessa
  - Maggior flessibilità
  - Rischio di memory leak
- Operatori specifici per allocare / rimuovere oggetti
  - new / delete
    - Per singoli oggetti
  - new [] / delete []
    - Per array
- È possibile usare anche il meccanismo proprio del C
  - Via funzioni malloc() / calloc() / realloc e free()
  - Vedremo perché non è un approccio consigliato parlando di classi e oggetti