TIPO ENUMERATIVO

- Un tipo enumerativo può memorizzare un insieme di valori definiti dall'utente
  - Una volta definita, un'enumerazione è usata come un tipo intero
  - Un'enumerazione definisce costanti intere, dette enumeratori, che possono essere nominate

enum keyword {*ASM*, *AUTO*, *BREAK*};

keyword key;

per default, ASM = 0, AUTO = 1, BREAK = 2

TIPO ENUMERATIVO

- Dichiarare una variabile keyboard piuttosto che int aumenta la leggibilità del programma e facilita il lavoro del programmatore

```
switch(key) {
    case ASM:
        // do something
        break;
    case AUTO:
        // do something
        break;
    case BREAK:
        // do something
        break;
}
```

TIPO ENUMERATIVO

- L'intervallo di un'enumerazione può essere modificato
  - Comprende tutti i valori minori della prossima potenza binaria

```
enum e1 {dark, light};                    // range 0:1


enum flag {x = 1, y = 2, z = 4, e = 8};     // range 0:15

flag f1 = 5;              // type error: 5 is not of type flag
flag f2 = flag(5);    // ok: flag(5) is of type flag and within the range of flag
```

DICHIARAZIONI E DEFINIZIONI

- Un identificatore, per essere usato in un programma C++, deve essere **dichiarato**, ossia deve essere specificato il suo tipo
- Una **dichiarazione** diventa una **definizione** se ad un identificatore gli viene associata un'entità

```
char ch;
string s;
int count = 1;
const double pi = 3.14159265358979323385;
extern int error_number;
char* name = "Njal";
char* season[] = {"spring", "summer", "fall", "winter"};
struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0?a:a; }
typedef complex<short> Point;
struct User;
enum Beer {Carlsberg, Tuborg, Thor};
namespace NS { int a; }
```

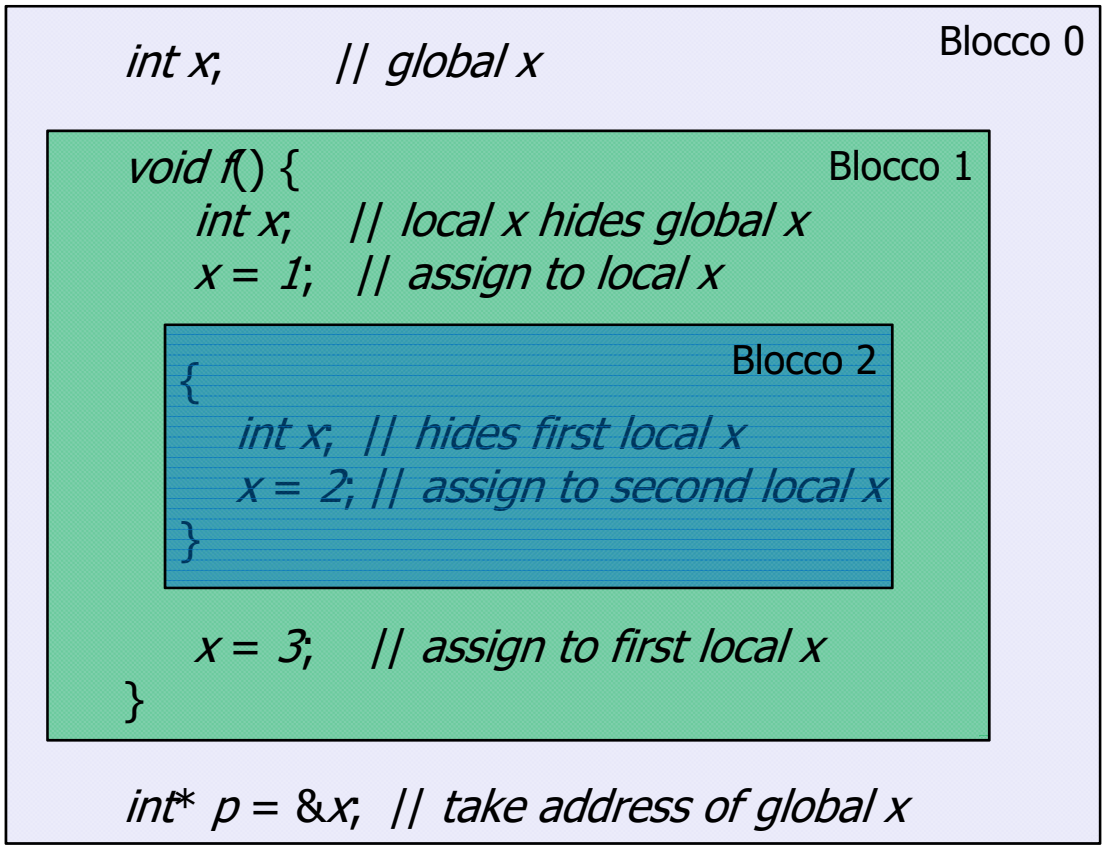
Le istruzioni in rosso sono solo dichiarazioni: le entità a cui tali identificatori fanno riferimento devono essere definite altrove

SCOPE

- Un identificatore introduce un nome in uno scope (o visibilità), ossia tale nome può essere usato solo in una specifica parte del programma
- Se un nome è dichiarato in una funzione (nome locale), lo scope si estende dal punto della dichiarazione alla fine del blocco
  - Un blocco è una sezione di codice delimitata dalla coppia { }
- Un nome è detto globale se è definito al di fuori di una funzione, classe o namespace
  - Lo scope di un nome globale si estende dal punto della dichiarazione alla fine del file
  - La dichiarazione di un nome in un blocco può nascondere una dichiarazione in un blocco di livello inferiore o un nome globale (cioè, un nome può essere ridefinito)

SCOPE

- I blocchi possono essere innestati
- Dopo un blocco, un nome riprende il suo precedente significato



SCOPE

- Un nome globale nascosto può essere riferito usando l'operatore di *risoluzione di scope ::*

```
int x;

void f2() {
    int x = 1;    // hide global x
    ::x = 2;      // assign to global x
    x = 2;        // assign to local x
    // ...
}

void f5(int x) {
    int x;    // error
}
```

```
int x = 11;

void f4() { // perverse:
    int y = x;    // use global x: y = 11
    int x = 22;
    y = x;        // use local x: y = 22
}
```

TYPEDEF

- Una dichiarazione premessa dalla parola chiave *typedef* dichiara un nuovo nome per un tipo piuttosto che una nuova variabile di un dato tipo
  - Tale nome può essere un diminutivo di un nome esteso

```
typedef char* Pchar;
Pchar p1, p2;    // p1 and p2 are char*s
char* p3 = p1;

typedef unsigned char uchar;
```

- Un altro uso di ***typedef*** è quello di limitare il diretto riferimento ad un tipo

```
typedef int int32;
typedef short int16;
```

Se necessario, *int32* può essere facilmente ridefinito:

```
typedef long int32;
```

ALCUNI CONSIGLI…

- Creare ***scope*** piccoli
- Non usare lo stesso nome in ***scope*** innestati
- Scegliere nomi univoci per ogni dichiarazione
- Usare nomi brevi per nomi comuni e locali e nomi lunghi per nomi non comuni e non locali
- Evitare nomi simili
- Scegliere nomi che riflettano il reale significato
- Preferire ***int*** a ***short int*** o a ***long int***
- Preferire ***double*** a ***float*** o a ***long double***
- Preferire ***char*** a ***signed char*** e ***unsigned char***
- Evitare operazioni aritmetiche sui tipi unsigned
- Sospettare delle conversioni signed/unsigned, unsigned/signed, floating-point/intero e delle conversioni da tipi più grandi a tipi più piccoli come int/char

Operatori

OPERATORE DI ASSEGNAMENTO

- L' operatore di **Assegnamento (=)** è utilizzato per assegnare un valore ad una variabile
  - **a = 5;**
  - Significa: "assegna il valore intero **5** alla variabile **a**".
- Da non confondere con il classico simbolo di confronto
- La parte alla sinistra dell'operatore è nota come *lvalue* (left value) e la parte destra *rvalue* (right value).
  - *lvalue* deve essere sempre una variabile mentre la parte destra può essere una costante, una variabile, il risultato di una operazione

```
int main() {
    int a=3;
    int b=a+3;
    b=b-3;
}
```

Significa: assegna alla variabile *b* il risultato di *a*+3

Quale sarà il valore di b?

OPERATORI ARITMETICI

- Gli **Operatori Aritmetici** sono:
  - \+ Addizione
  - \- Sottrazione
  - / Divisione
  - * Moltiplicazione
  - % Modulo
- L'operatore di modulo fornisce il resto della divisione di due numeri interi.
- Ad esempio, se scriviamo:
  - **a = 11 % 3;**
- viene assegnato alla variabile **a** il valore **2** in quanto **2** è proprio il resto che si ottiene dividendo 11 per 3

OPERATORI DI INCREMENTO/DECREMENTO

- Gli **Operatori di Incremento (++) e Decremento (--)** aumentano o diminuiscono di 1 il valore di una variabile e sono equivalenti a:
  - a++; ⟷ a=a+1; ⟷ a+=1;
  - a--; ⟷ a=a-1; ⟷ a-=1;
- Questi operatori si possono usare sia come *prefissi* (++a) che come *postfissi* (a++)
  - Come *prefisso*: Il valore della variabile viene incrementato prima della valutazione dell'espressione e quindi l'espressione viene valutata usando il valore incrementato.
  - Come *postfisso*: il valore della variabile viene incrementato dopo la valutazione dell'espressione e quindi l'espressione viene valutata usando il valore non incrementato.

OPERATORI RELAZIONALI

- Per confrontare i valori di due espressioni si usano gli **Operatori relazionali**
- Il risultato di un operatore relazionale è di tipo **bool** e può assumere soltanto uno dei due valori booleani **true** o **false**, a seconda del risultato del confronto.
  - **== uguale**
  - **!= non uguale**
  - **> maggiore**
  - **< minore**
  - **>= maggiore uguale**
  - **<= minore uguale**
  - Ad esempio:
    - 7==5 risultato **false**
    - 3!=2 risultato **true**
    - 5>4 risultato **true**
    - 6>=6 risultato **true**
    - 5<5 risultato **false**
- **ATTENZIONE** → Non confondere l'operatore di assegnamento **(=)** con l'operatore di uguaglianza **(==)**

OPERATORE LOGICO NOT

- L'operatore logico **!** è l'operatore logico di negazione NOT.
- Esso ha un **unico operando** (di tipo **bool**) posto alla sua destra e il suo risultato è l'opposto del valore dell'operando
- Se l'operando è **true** esso ritorna **false** , se l'operando è **false** esso ritorna **true** . Ad esempio:
  - !(5 == 5) ritorna **false** in quanto l'espressione alla sua destra (5== 5) è **true**
  - !**true** ritorna **false**

OPERATORI LOGICI – 2/2

- Gli operatori **&&** e **||** sono gli operatori logici di congiunzione (AND) e disgiunzione (OR). Essi hanno **due operandi** (di tipo **bool**), uno alla sinistra ed uno alla destra
- Il loro risultato è riportato dalla corrispondente **tabella di verità**

Primo operando **a**	Secondo operando **b**	Risultato di **a && b**	Risultato di **a || b**
true	true	**true**	**true**
true	false	**false**	**true**
false	true	**false**	**true**
false	false	**false**	**false**

- Ad esempio:
  - **((5==5) && (3>6)) -** ritorna false (true && false)
  - **((5==5) || (3>6)) -** ritorna true (true || false)

OPERATORI DI CONVERSIONE (CASTING)

- Gli operatori di conversione servono per convertire valori appartenenti ad un tipo in valori di un altro tipo.
- In C++ sono possibili due modi:
  - Far precedere le espressioni che devono essere convertite dal nome del nuovo tipo racchiuso tra parentesi. Ad es:

```
int e;
float f=3.14;
e=(int)f;
```

Converte il numero float 3.14 nel valore intero 3.

  - Usare la funzione *costruttore*. Ovvero, far precedere l'espressione da convertire, racchiusa tra parentesi, dal nome del nuovo tipo racchiuso da parentesi ()

```
int e;
float f=3.14;
e=int(f);
```

ESERCIZIO - ECHO2

- Scrivere un programma che, dopo aver chiesto all'utente di inserire un carattere, lo stampa sul terminale insieme alla sua codifica ASCII

- Scrivere un programma che, dopo aver chiesto all'utente di inserire un numero intero, lo stampa sul terminale insieme al carattere ASCII corrispondente

OPERATORE SIZEOF

- L'operatore **sizeof** ritorna la memoria, in byte, necessaria a memorizzare un valore di tale tipo o il valore dell'espressione
- Questo operatore ha un parametro che può essere sia il nome di un tipo che una espressione.
- Ad esempio:

```
int s;
s=sizeof(char);
```

- ritorna **1** in **s** perché un valore di tipo **char** occupa un byte.

PRIORITÀ TRA OPERATORI

Priorità	Operatore	Descrizione	Associatività
1	**::**	scopo	Sinistra
2	**() [ ] -> . sizeof**		Sinistra
	++ --	incremento/decremento	
	~	Complemento a uno (bit a bit)	
	!	NOT	
3	**& ***	Referenziazione e Dereferenziazione (puntatori)	Destra
	(*tipo*)	Conversione di tipo	
	+ -	Operatori di segno unario	
4	*** / %**	Operatori aritmetici moltiplicativi	Sinistra
5	**+ -**	Operatori aritmetici additivi	Sinistra
6	**<< >>**	Spostamento dei bit	Sinistra
7	**< <= > >=**	Operatori relazionali	Sinistra
8	**== !=**	Operatori relazionali di uguaglianza	Sinistra
9	**& ^ |**	Operatori bit a bit	Sinistra
10	**&& ||**	Operatori logici	Sinistra
11	**?:**	Operatore condizionale	Destra
12	**= += -= *= /= %= >>= <<= &= ^= |=**	Assegnamento	Destra
13	**,**	Operatore virgola, separatore	Sinistra

- Ad es., l'espressione:
  - *p++

E' analizzata come *(p++), e non come (*p)++

- L'espressione:
  - !b && c

E' analizzata come (!b)&&c e non come !(b&&c)

ESEMPIO


```
#include<iostream>
#include<string>

using namespace std;

/* main */
int main() {
   int ore = 4;
   int minuti = 21;
   int secondi = 0;

   bool timeslTrue = ore && minuti && secondi;

   cout<<"Risultato: "<<timeslTrue<<endl;
   return 0;
}
```

```
osboxes@osboxes:~/Documents/examples$ ./tipi_ex
Risultato: 0
osboxes@osboxes:~/Documents/examples$
```

Poiché l'operatore && richiede operandi di tipo boolean, per le variabili secondi, minuti e ore è implicito un cast a boolean.
Coerentemente con l'interpretazione dei boolean nel C, il cast a boolean di un intero segue la regola

Intero == 0 ➔ cast a false
Intero != 0 ➔ cast a true

- L'espressione *ore && minuti && secondi* si traduce in:
  - true && true && false
- Il risultato è **false**. Il cout stampa a video il valore false convertito in intero, ovvero 0.

INPUT/OUTPUT DA CONSOLE IN C++

- Abbiamo visto in precedenza come stampare a video con **cout**
- Oltre a stampare a video, è possibile anche chiedere un **input** all'utente tramite inserimento dal terminale:

```
#include <iostream>

using namespace std;
int main() {
   int elemento;
   cout << "Inserire un intero:"<<endl;
   cin>>elemento;
   cout<<"Hai inserito: "<<elemento;
}
```

Output

```
> Inserire un intero:
> 5
> Hai inserito: 5
```

- Il valore inserito dall'utente deve essere memorizzato. Per questo definiamo una variabile "elemento" che sarà utilizzata dal **cin**
- endl significa "**endl**ine" e serve ad andare a capo

ESERCIZI DI AUTOVALUTAZIONE


```
int a=4;
int b;
float c;

a++; // quanto vale a?
b =--a+2; // e b?
b =a-- +2; // ora?
b=a++; // ???
c=b/a; // quanto vale c?
c=(float)b/a; // quanto vale c?
```

```
bool d = a; // quanto vale d?
bool e = !(!d || a); // ??
bool f  = ...  // vero se d falso e
               // e vero
 f= ... // se d ed e sono
        //entrambi falsi
f= ... // vero se solo uno
       // tra d ed e è vero
```

ESERCIZIO

- Considerando il seguente output:

```
Esercizi$ ./esercizio
Inserire la prima lettera del proprio nome (MAIUSOLO): L
Il tuo nome inizia per ... L
Il codice ascii di questa lettera è: 76
La lettera L è la 11a dell'alfabeto
```

- Si completi il seguente programma:

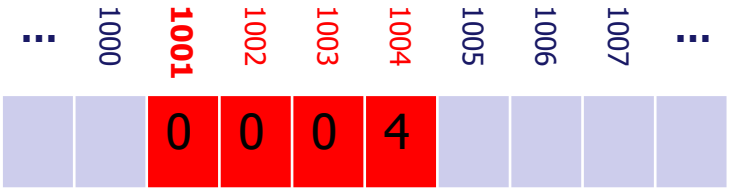
```
#include <iostream>
using namespace std;
int main() {
    char a;
    cout << "Inserire …";
    … >> a;
    cout << "Il tuo nome inizia per ... " << …;
    cout << "Il codice ascii di questa lettera è: " << …
    << endl;
    cout << "La lettera " << a
    << " è la " << … << "a dell'alfabeto" << endl;
}
```

ESERCIZI

- Esercizio 1 – Scrivere un programma che:
  - Richieda in input all'utente due interi
  - Sommi i due addendi
  - Stampi a video il risultato
- Esercizio 2 – Scrivere un programma che:
  - Richieda in input all'utente due interi
  - Scriva in una variabile il risultato del confronto di uguaglianza tra i due numeri
  - Stampi a video il risultato
- Esercizio 3 – Scrivere un programma che:
  - Riceva in input un float
  - Converta in intero
  - Stampi a video il risultato ed anche la grandezza della variabile

RIEPILOGO


```
int a = 4;
```



&a = 0x1001

- Tipi fondamentali:
  - int (short / long)
  - float
  - double
  - char
  - bool
- Qualificatori: signed/unsigned
  - const
- Esercizio: riepilogo.cpp
- I/O

  C++

  cin >> a;

  cout << a;

TIPI FONDAMENTALI

- Tipo **booleano** (*bool*)
- Tipi **carattere** (come *char*)
- Tipi **interi** (come *int*)
- Tipi **floating-point** (come *double*)
- Tipi **enumerativi** (*enum*)
- Tipo che specifica l'assenza di informazioni (*void*)

Tipi integrali

Tipi aritmetici

- Tipi **puntatori** (come *int**)
- Tipi **array** (come *char[]*)
- Tipi **riferimento** (come *double&*)
- **Strutture dati** e **classi**