Il linguaggio C

Mauro Fiorentini

Bibliografia (1)

- Testi di riferimento per il linguaggio
  - M. Fiorentini, C. Tibaldi,
    Programmare in C, **Seconda edizione**,
    Zanichelli, 1992.
  - Kernighan, D. Ritchie,
    The C Programming Language,
    Second edition, Prentice-Hall, 1988. R. W.
  - Kernighan, D. Ritchie,

    Linguaggio C - Seconda edizione,
    Gruppo Editoriale Jackson, 1989.
  - Darnell, P.E. Margolis,
    C - Manuale di programmazione,
    McGraw-Hill, 1991.

Bibliografia (2)

- Esercizi svolti
  - L. Tondo, S. E. Gimpel,

    The C Answer book, **Second edition**, Prentice-Hall, 1989.
- Lo standard ISO
  - ISO/IEC 9899,

    Programming languages – C,

    Prima edizione: 15/12/1989,

    Seconda edizione: 12/1/1999,

    Terza edizione: 8/12/2011.

Bibliografia (3)

- Per approfondire
  - Koenig,
    C Traps and Pitfalls,
    Addison-Wesley, 1989.
- Per dubbi sullo standard
  - Plauger, J. Brodie,
    ANSI and ISO Standard C Programmer's Reference,
    Microsoft Press, 1992.
- Per le motivazioni dello standard
  - *Rationale for the ANSI C Programming Language*,
    Silicon Press, 1990.

Bibliografia (4)

- Testi generali sui concetti dei linguaggi
  - R. W. Sebesta,

  Concepts of Programming languages,

  V ediz., Addison Wesley, 2002.
  - C. Ghezzi, M. Jazayeri,

  Programming Language Concepts,

  Wiley, 1982.

Sommario del corso (1)

- Introduzione
- Caratteristiche lessicali
- Tipi base
- Variabili
- Costanti
- Operatori ed espressioni
- Istruzioni
- Funzioni

Sommario del corso (2)

- Struttura di un programma e classi di memoria
- Vettori e matrici
- Puntatori
- Vettori e puntatori
- Vettori, puntatori e funzioni
- Stringhe
- Memoria dinamica
- Strutture

Sommario del corso (3)

- Unioni
- Valori composti
- Approfondimenti sui tipi
- Il preprocessore
- Inclusione di file
- Macro
- Compilazioni condizionali
- Errori e direttive per il compilatore

Introduzione

The power of a language is not only determined by what it allows to express, but equally much by what it prohibits to express.

Breve storia (1)

- Nei primi anni ’60 all’università di Cambridge fu sviluppato il CPL (Cambridge Programming Language, poi ridefinito come Combined Programming Language).
  - Voleva essere un superamento dell’ALGOL-60.
  - Molto grande, il primo compilatore vide la luce intorno al 1970.
  - Non fu mai molto popolare e scomparve nei ptimi anni ‘70 senza lasciare tracce.

Breve storia (2)

- Nel 1967 Martin Richards implementò il BCPL (Bootstrap CPL, poi ridefinito come Basic CPL).
  - Pensato come versione molto ridotta del BCPL.
  - Nato per scrivere compilatori, doveva essere molto piccolo e facilmente portabile.
    - Aveva un unico tipo di dato: word.
    - Un compilatore richiedeva solo 16k di codice.
  - Molto usato su piccole macchine
  - Divenne popolarissimo in UK.
    - Fu “il” linguaggio della didattica per 20 anni.
  - Tuttora in uso, principalmente in UK e USA.

Breve storia (3)

- Nel 1969 Ken Thompson e Dennis Ritchie definirono ai Bell Labs una versione ridotta del BCPL, chiamata B.
  - Nato per scrivere la prima versione di UNIX.
  - Linguaggio ora estinto.
- Nel 1970 i due definirono un successore del B, per scrivere la seconda versione di UNIX, che chiamarono C.
  - Mancava di molte caratteristiche del C attuale, come vettori ed enumerativi.

Breve storia (4)

- Il C si diffuse rapidamente negli anni ’80, grazie alla "esplosione" di UNIX …

  …ma ormai svincolato da UNIX (MS-DOS, Windows, ...).
- Largamente adottato nel mondo industriale (programmazione di sistema, tool, ...).
- Ci furono proposte per un successore, da chiamarsi D o P, che non vide mai la luce.
  - L’attuale D (Walter Bright, 2001) è un linguaggio a oggetti, nato come miglioramento del C++.

Gli standard

- ANSI e ISO emisero uno standard per il linguaggio nel 1989-1990.
  - Fu grande lavoro per correggere alcuni grravi difetti.
  - Rese **obsoleto** il vecchio stile “K&R”, che fu però duro a morire.
- La seconda versione dello standard, fu emessa nel 1999.
  - Corresse ulteriori difetti e aggiunse estensioni.
- La terza versione dello standard, con ulteriori estensioni, è stata emessa nel 2011.

Lo “spirito” del C

- **Fidati** del programmatore.
- **Non impedire** al programmatore di fare ciò che deve essere fatto.
- Mantieni il linguaggio piccolo e semplice.
- Genera codice **efficiente**, anche a scapito della portabilità.

(Dalle motivazioni del comitato di standardizzazione del C)

Vantaggi del linguaggio (1)

- Estrema portabilità.
  - Sono disponibili implementazioni per praticamente tutte le piattaforme, da microprocessori a 8 bit a supercalcolatori.
  - Esistono subset (non standard) per macchine piccolissime.
- Implementazioni molto efficienti.
- Non necessita di un supporto run-time.
  - Un programma C può girare su macchine con memoria ridottissima.
  - Facilmente interfacciabile con il C++.
- Grande versatilità.
  - Può essere utilizzato per qualsiasi scopo.
  - Si può gestire la memoria nei minimi dettagli.
  - Si possono gestire le caratteristiche di basso livello della macchina.

Vantaggi del linguaggio (2)

- Ha una grande libreria standard per facilitare il lavoro.
  - Comprende varie centinaia di funzioni.
- Si trovano in commercio o gratuitamente grandi quantità di codice già pronto.
- Largamente usato nell'industria dove contano le prestazioni (giochi, sistemi operativi strumenti di sviluppo) o dove le risorse h/w sono modeste.
  - Domina nel mercato embedded, dove ha sostituito i vari assembler.

Svantaggi del linguaggio (1)

- Ha una sintassi indecente.
  - Largamente copiata da altri linguaggi (C++, Java, Go).
- Ha una struttura dei tipi **primitiva** (non molto meglio del FORTRAN77).
  - Non è possibile ridefinire tipi in modo da renderli incompatibili col tipo originario.
  - Eclatante la mancanza di subrange.
  - La gestione delle stringhe è primitiva, anche se facilitata dalle funzioni di libreria.
  - Caratteri ed enumerativi considerati come interi.
- Non ci sono controlli o meccanismi automatici per scopire errori.
  - Il programmatore **non deve sbagliare**!
  - Per chi vuole sbagliare ci sono Java e Go.

Svantaggi del linguaggio (2)

- Manca ancora di caratteristiche relativamente elementari.
  - Per esempio, un fattore di ripetizione nelle inizializzazioni.
- Richiede una grande attenzione.
  - La portabilità richiede **profonda conoscenza** dello standard, che lascia numerose caratteristiche dipendenti dall'implementazione.
- Permette un po' di tutto.
  - La coesistenza di strutture di medio livello, operatori di basso livello e accessibilità dell'hardware richiede grande competenza per essere gestita correttamente.
    - Si può programmare in modo **assolutamente incomprensibile**.

Caratteristiche del linguaggio

- Linguaggio imperativo:
  - general purpose,
  - procedurale,
  - a blocchi.
- Nato per la programmazione di sistema:
  - grande attenzione all'efficienza;
  - grande flessibilità e libertà;
  - compatto;
  - relativamente semplice;
  - portabile, ma con attenzione!

Caratteristiche generali (1)

Tipi base		Keyword

aritmetici	interi	`short int`
		`int`
		`long int`
		`long long int`
		`char`
		`_Bool`
		} `signed` o `unsigned`

	reali	`float`
		`double`
		`long double`

	complessi e immaginari	`float _Complex`
		`double _Complex`
		`long double _Complex`
		`float _Imaginary`
		`double _Imaginary`
		`long double _Imaginary`

vuoto		`void`
enumerati		`enum`

Caratteristiche generali (2)

Tipi derivati	**Keyword**
record	`struct`
unioni	`union`
puntatori	
vettori	

Caratteristiche generali (3)

Strutture di controllo	**Keyword**
sequenza	
selezione	`if` `switch`

iterazione	`for`
	`while`
	`do-while`

salto	`goto`
	`break`
	`continue`
	`return`

Caratteristiche generali (4)

Scomposizione del programma
unità di compilazioni separate
funzioni
blocchi

Caratteristiche generali (5)

- Alcune caratteristiche sono implementate tramite librerie standard:
  - I/O;
  - gestione stringhe;
  - gestione della memoria dinamica;
  - funzioni matematiche.

Caratteristiche generali (6)

- Il C prevede un preprocessore, che processa il codice prima del compilatore.
- Utilizzato per:
  - macroespansioni, con e senza parametri;
  - compilazioni condizionali;
  - inclusione di file.

Implementazioni

- Lo standard prevede due tipi di implementazioni:
  - hosted, vale a dire su una macchina con shell e sistema operativo;
  - freestanding, ossia su macchine prive di sistema operativo e shell.
- Le implementazioni hosted devono essere complete.
- Le implementazioni freestanding possono mancare di alcune caratteristiche (ben definite).
  - Sono pensate per processori piccoli, con poca memoria, sui quali sarebbe difficile o impossibile implementare tutto il linguaggio.
- Inoltre per molti microprocessori sono disponibili implementazioni parziali, che non possono dirsi standard.
- Molte implementazioni hanno poi estensioni varie.

Il primo programma C

File prog.c

```
# include   <stdio.h>
# include   <stdlib.h>
int   main(void)
  {
  printf("Ciao\n");
  return EXIT_SUCCESS;
  }
```

Per compilare:

```
cc prog.c
```

L’eseguibile per default si chiama a.out (Linux) o prog.exe (Windows).

Un secondo esempio (1)

Programma che legge i caratteri in input e ne scrive il codice esadecimale, raggruppandoli a 10 per linea.

```
# include <stdio.h>
# include <stdlib.h>
int main(void)
  {
  int           c;
  unsigned int n;

  n = 1;
```

Un secondo esempio (2)

```
while ((c = getchar()) != EOF)
  {
  printf("%x",c);
  if (n++ == 10)
              {
              n = 1;
              printf("\n");
              }
  else
              printf(" ");
  }
```

Un secondo esempio (3)

```
  // chiude la linea incompleta
if (n != 1)
  printf("\n");
return EXIT_SUCCESS;
}
```

Caratteristiche lessicali

Struttura del codice (1)

- Un programma C è formato da una o più unità di compilazione separate (file).
  - Ciascuna contiene, in ordine arbitrario, dichiarazioni e definizioni di variabili e/o funzioni, sia globali, sia locali al file.
  - Per evitare duplicazioni si possono includere file interi in differenti unità di compilazione.
- In ciascuna unità, la dichiarazione di ogni entità (tipo, variabile, funzione) deve **fisicamente precedere** il suo utilizzo.

Struttura del codice (2)

- Per motivi di chiarezza, di solito le dichiarazioni di un'unità di compilazione vanno messe nel seguente ordine:
  - macrodefinizioni;
  - tipi;
  - dichiarazioni di variabili;
  - dichiarazioni di funzioni;
  - definizione di funzioni.

Codifica dei caratteri

- Un programma in C si scrive col set di caratteri disponibile sulla macchina.
  - Di solito ASCII o EBCDIC.
- Per rappresentare caratteri non disponibili si possono usare i caratteri universali:
  - `\u` seguito da 4 cifre esadecimali;
  - `\U` seguito da 8 cifre esadecimali.
- Sebbene permesso dallo standard, i caratteri universali non vanno usati per scrivere codice e commenti, ma solo entro stringhe e costanti carattere!

Terminazione delle linee

- Una linea del sorgente termina normalmente col carattere a capo (\n).
- Se il carattere a capo è immediatamente preceduto da \ (backslash), i due caratteri vengono eliminati e la linea **prosegue** col contenuto della seguente.
  - Questa operazione viene eseguita prima di qualsiasi altro trattamento della linea.

Raggruppamento di token

- I token lessicali sono formati aggregando la più lunga sequenza di caratteri possibile (**maximal munch strategy**), anche se il risultato è un errore sintattico.
  - Non vengono effettuati altri tentativi di raggruppamento.
- Per esempio:

  `a--b` è letto come `a -- b`, non come `a - (- b)`, anche se la prima interpretazione è un errore sintattico e la seconda sarebbe valida.

  `5---b` è letto come `5-- - b`, non come `5 - --b`.

Formato del codice (1)

- Il C è un linguaggio a formato libero.
  - Si può andare a capo quando si vuole.
  - Si possono mettere più istruzioni o dichiarazioni sulla stessa riga.
  - Si può spezzare un'istruzione o dichiarazione su più righe.
- Newline, line feed, tabulazioni orizzontali e verticali, commenti e spazi possono essere inseriti a piacere tra i token.

Formato del codice (2)

- Un buon utilizzo dell'indentazione migliora la leggibilità.
  - Bisogna **evitare** di scrivere più dichiarazioni o istruzioni su una riga.
  - E’ **normale** che una dichiarazione o istruzione si estenda su più righe.

I commenti

- Vi sono due modi di scrivere commenti:
  - tra `/*` e `*/` ;
  - iniziando con `//`, senza chiuderli: terminano a fine riga.
- I commenti non possono contenere commenti.
  - Aperto un commento, ogni carattere viene ignorato sino al terminatore corretto.

Gli identificatori

- Usati come nomi delle varie entità (tipi, variabili, funzioni).
- Sequenze di lettere e cifre inizianti con una lettera, di lunghezza arbitraria.
  - L'underscore conta come lettera.
  - Almeno 63 caratteri sono significativi.
    - Solo 31 garantiti per gli identificatori esterni.
  - Maiuscole e minuscole sono distinte.
    - Distinzione non garantita per gli identificatori esterni.
- Devono essere diversi dalle parole chiave.

Identificatori riservati

- Lo standard riserva gli identificatori che iniziano con doppio underscore o underscore e lettera maiuscola per eventuali estensioni future.
  - Usare ora identificatori del genere potrebbe provocare problemi in futuro.
- Lo standard riserva gli identificatori che iniziano con underscore e lettera minuscola per estensioni locali.
  - Usarli potrebbe creare problemi di portabilità su alcune piattaforme.
  - P. es., `_pascal` è parola riservata in Visual C.

Scope degli identificatori

- Ogni identificatore deve essere **unico** nel suo scope, che può essere:
  - l'intero programma (identificatori esterni);
  - una unità di compilazione;
  - una funzione;
  - un blocco di istruzioni in una funzione;
  - una struttura dati.

Parole chiave

- Sono 37:

```
auto      enum      register  typedef
break     extern    restrict  union
case      float     return    unsigned
char      for       short          void
const     goto      signed    volatile
continue  if        sizeof    while
default   inline    static    _Bool
do        int       struct    _Complex
double    long      switch    _Imaginary
else
```

Tipi base

I tipi

- Il C è un linguaggio tipizzato.
  - La compatibilità dei tipi è verificata staticamente in tutti i casi, tranne nel caso di conversione di puntatori, nel qual caso non è verificata.
- I tipi si dividono in:
  - primitivi, predefiniti, identificati da una parola chiave;
  - enumerati, definibili dal programmatore;
  - record, unioni e vettori, definibili dal programmatore.

Tipi predefiniti

- Si dividono in:
  - numerici:
    - interi, inclusi caratteri e booleani;
    - reali;
    - immaginari e complessi;
  - `void`.

Tipi interi (1)

- Sono sei:
  - `_Bool`, almeno 1 bit;
  - `char`, almeno 8 bit;
  - `short` [`int`], almeno 16 bit;
  - `int`, almeno 16 bit;
  - `long` [`int`], almeno 32 bit;
  - `long long` [`int`], almeno 64 bit;
- Due o più possono coincidere.
  - Spesso `int` coincide con `short` o `long`.
  - Dipende comunque dall'implementazione.

Tipi interi (2)

- La dimensione non è specificata, ma sia come numero di bit, sia come **intervallo di valori rappresentabili**, deve essere:

  $|\texttt{long long}| \geq |\texttt{long}| \geq |\texttt{int}| \geq |\texttt{short}| \geq |\texttt{char}|$.

- La rappresentazione è in base 2, ma **non necessariamente** in complemento a 2.

Caratteristiche dei tipi interi

- Il tipo `int` è il tipo intero di dimensioni più "naturali" per la macchina.
  - Spesso di dimensioni pari a un registro.
  - Garantisce la **massima efficienza**.
- Il tipo `short int` è un tipo di dimensioni di solito inferiori.
  - Permette un **risparmio di memoria**, a scapito delle prestazioni.
- I tipi `long int` e `long long int` hanno dimensioni maggiori.
  - Permettono di rappresentare interi **maggiori**, a scapito di occupazione e prestazioni.

Modificatori di tipo

- Sono due: `signed` e `unsigned`.; possono essere scritti prima del nome del tipo.
  - Il modificatore `signed` è il default e normalmente si omette.
- Gli interi `signed` permettono di rappresentare valori positivi e negativi in un intervallo simmetrico (o quasi) intorno allo zero.
- Gli interi `unsigned` hanno la stessa dimensione dei corrispondenti tipi `signed` e permettono di rappresentare valori non negativi, sino a un massimo circa doppio.
  - I valori rappresentabili contengono i valori non negativi rappresentabili dai corrispondenti tipi `signed`.

Uso corretto dei tipi interi (1)

- Dovendo rappresentare valori interi:
  - si utilizza **di norma** il tipo `int`;
  - si utilizzano i tipi `long int` e `long long int` se è necessario rappresentare valori "grandi";
  - si utilizza il tipo `short int` **solo per risparmiare memoria**.
- Servono **motivi precisi** per non utilizzare il tipo `int`.
- L'utilizzo di tipi diversi da `int` può peggiorare le prestazioni.

Uso corretto dei tipi interi (2)

- Se una variabile (funzione) non può **concettualmente** contenere (produrre) valori negativi, va dichiarata `unsigned`.
  - **Si guadagna in chiarezza**.
  - Si può avere un marginale guadagno di prestazioni.

Tipi carattere

- Ci sono 3 tipi carattere, considerati tipi aritmetici interi: `char`, `signed char` e `unsigned char`, rappresentati con almeno 8 bit.
  - Hanno tutti dimensione 1.
  - A seconda dell'implementazione il tipo `char` può coincidere con `signed char` o `unsigned char`, oppure essere un tipo distinto.
  - Il tipo `char` deve poter rappresentare qualsiasi carattere del set base della macchina.
  - Il set utilizzato dipende dall'implementazione
    - Di solito ASCII o EBCDIC.

Caratteri e ordinamento

- È garantito **l'ordinamento** dei caratteri alfabetici maiuscoli e minuscoli e numerici tra di loro; quindi:

  `'A'` < `'B'`

  `'0'` < `'9'`

  `'a'` < `'z'`

- **Non è garantito** l'ordinamento **relativo** dei suddetti insiemi, né dei restanti caratteri; quindi:

  `'0'` < `'A'` oppure `'0'` > `'A'`

  `'a'` < `'A'` oppure `'a'` > `'A'`

  `'.'` < `'A'` oppure `'.'` > `'A'`

Caratteri e consecutività

- È garantita la **consecutività dei caratteri numerici**, quindi:

  `'9' – '0'` è uguale a 9.

- **Non è garantita la consecutività** dei caratteri **alfabetici**, quindi:

  `'Z' – 'A'` non è necessariamente uguale a 25.

Tipi carattere e interi (1)

- I tipi carattere sono considerati interi e sulle variabili carattere sono ammesse operazioni aritmetiche.
  - Per esempio:

```
signed char    a, b, c;

a = b + c;
a = b * 2;
a = 'a' * 2; // orribile, ma lecito
```

Tipi carattere e interi (2)

- Variabili di un tipo carattere carattere possono essere utilizzate per rappresentare valori interi piccoli.
  - Da –127 a 127 se `signed`, da 0 a 255 se `unsigned`.
  - Si risparmia memoria e si peggiorano le prestazioni.
  - Ha senso solo se componenti di strutture allocati in gran numero o grandi vettori.
  - Usarli come interi molto piccoli **peggiora la leggibilità**.
- Se necessario risparmiare memoria, meglio usare nomi diversi.
  - P. es.: `typedef unsigned char byte;`

Il tipo wchar_t

- Non è un tipo predefinito: è definito in stddef.h e coincide con uno dei tipi interi.
  - Rappresentato con almeno 8 bit, è il tipo usato per rappresentare il massimo insieme di caratteri supportato dall'implementazione.
  - Generalmente `unsigned short`, per supportare Unicode.
  - Può essere `signed` o `unsigned`.

Tipo booleano

- Introdotto dalla seconda versione dello standard col nome `_Bool`.
- È di fatto un tipo intero, che contiene i valori 0 e 1.
- Il file stdbool.h contiene le definizioni delle seguenti macro, che **non sono parole chiave**:

  `true`, valore 1;

  `false`, valore 0;

  `bool`, valore `_Bool`.

Tipi reali

- Sono 3:
  - `float`, di solito rappresentato con 32 bit;
  - `double`, di solito rappresentato con 64 bit;
  - `long double`, di solito rappresentato con più di 64 bit.

Caratteristiche dei tipi reali

- Due o più possono coincidere.
  - Spesso `double` e `long double` coincidono.
  - Dipende comunque dall'implementazione.
- In ogni caso, sia come dimensione, che come intervallo di valori rappresentabili, che come precisione:

  `long double` ≥ `double` ≥ `float`

Esempio di uso di tipi reali (1)

Conversione da Fahrenheit a Celsius

```
# include <stdio.h>
# include <stdlib.h>

  // print Fahrenheit-Celsius table
  // for fahr = 0, 20,…,300;
int  main(void)
  {
  float   fahr, celsius;
  int     lower, upper, step;
```

Esempio di uso di tipi reali (2)

```
  lower = 0;   // lower limit of table
  upper = 300; // upper limit
  step = 20;   // step size

  printf("Fahr Celsius\n");
  for (fahr = lower; fahr <= upper;
     fahr = fahr + step)
     {
     celsius = (5.0 / 9.0) * (fahr  - 32.0);
     printf("%3.0f %6.1f\n", fahr, celsius);
     }
return EXIT_SUCCESS;
}
```

Tipi complessi

- Sono 3, introdotti dalla seconda versione dello standard, corrispondenti ai tipi reali, con nome:
  `float _Complex`,
  `double _Complex`,
  `long double _Complex`.
- Non sono obbligatori nelle implementazioni freestanding.
- Ciascuno corrisponde a un vettore di 2 componenti del corrispondente tipo reale.
  - Il primo elemento contiene la parte reale, il secondo quella immaginaria.

Tipi immaginari

- Sono 3, introdotti dalla seconda versione dello Standard, corrispondenti ai tipi reali, con nome:
  `float _Imaginary`,
  `double _Imaginary`,
  `long double _Imaginary`.
- Nelle espressioni sono trattati come numeri complessi.
  - Corrispondono a numeri complessi con parte reale nulla.
- **Non sono obbligatori**: potrebbero non essere supportati.

Il tipo void

- Tipo predefinito, che corrisponde all'insieme vuoto.
- Va utilizzato in 4 casi:
  - per dichiarare funzioni che non producono un valore;
  - per dichiarare funzioni senza argomenti;
  - per dichiarare puntatori "generici";
  - per indicare che il valore prodotto da una funzione non interessa.

Tipi enumerati

Tipi enumerati

- Sono definiti elencando i possibili valori.
- La dichiarazione ha la forma:

  `enum` [*identificatore*] `{` *lista di costanti* `}`

  - La lista di costanti è formata da identificatori separati da virgole.
  - Esempio:

```
enum colour_type { RED, BLUE, GREEN, YELLOW };
```

- Di solito, le costanti si scrivono in caratteri tutti maiuscoli.

Nome dei tipi enumerati

- Se l'identificatore manca, il tipo resta anonimo e non può essere riutilizzato altrove.
  - Serve a definire le costanti ed eventualmente una variabile. Per esempio:

```
enum { RED, BLUE, GREEN, YELLOW }
     colour;
```

- Il nome del tipo è `enum` <*identificatore*>: la parola chiave `enum` fa parte del nome.

Codifica dei tipi enumerati

- Per default sono assegnati interi consecutivi, a partire dalla prima costante, che vale zero.
- Si possono imporre codifiche differenti, anche se serve di rado farlo.
  - Esempio:

```
enum fruit
  {
  APPLE, PEAR = 3, GRAPE, LIME = 5,
  KIWI
  };
```

Uso corretto dei tipi enumerati (1)

- Il C li considera equivalenti a tipi interi (a quale tipo intero dipende dall'implementazione).
- Vanno però trattati col rigore del Pascal o di Ada, quindi le uniche operazioni corrette sono:
  - assegnamento con espressioni dello stesso tipo;
  - confronto con espressioni dello stesso tipo;
  - somma con interi;
  - differenza tra espressioni dello stesso tipo;
  - gestione tramite puntatori allo stesso tipo;
  - passaggio come argomenti a funzioni, che dichiarino argomenti formali dello stesso tipo.

Uso corretto dei tipi enumerati (2)

- **Nessun controllo** viene effettuato né sugli assegnamenti, né sulle operazioni.
- Trattare gli enumerati come interi non inficia la portabilità, ma la leggibilità del codice.

Esempi di operazioni con enumerati

```
  // Esempi di operazioni normali
colour = RED;
if (colour == YELLOW)
  …

  // Esempi insensati, ma ammessi
colour = APPLE;
colour = RED + BLUE;
colour = BLUE + LIME;
```

Variabili

Variabili

- Le variabili sono caratterizzate da:
  - un nome, fissato nella dichiarazione e immutabile;
  - un tipo, fissato nella dichiarazione e immutabile;
  - un valore;
  - una porzione di codice nella quale sono accessibili (“scope”);
  - una durata temporale.

Allocazione delle variabili

- Le variabili possono essere:
  - **statiche**, presenti per tutta la durata dell'esecuzione;
  - **automatiche**, presenti durante l'esecuzione di una funzione o di un blocco di istruzioni;
  - **dinamiche**, create e cancellate su richiesta.
    - Le variabili dinamiche non sono dichiarate e non hanno nome.

Scope delle variabili

- Per le variabili locali si estende dalla dichiarazione alla fine del blocco che la contiene.
  - Gli argomenti formali hanno come scope la funzione.
- Per le variabili globali si estende dalla dichiarazione alla fine del file che la contiene.
- Una variabile maschera nel suo scope eventuali variabili omonime di blocchi più esterni.
  - Omonimie del genere vanno comunque **evitate**.
- I nomi devono essere unici all'interno di ogni blocco.

Dichiarazioni e definizioni

- La **dichiarazione** specifica le proprietà di un oggetto.
  - Per una variabile specifica classe di memoria, nome e tipo;
  - Per una funzione specifica classe di memoria, nome, tipo, numero e tipo degli argomenti.
- La **definizione** specifica le stesse proprietà e provoca anche l'allocazione di memoria.
  - Per una funzione contiene il codice.
- Una dichiarazione non coincide necessariamente con la definizione.
  - P. es., nel caso di dichiarazioni `extern`.
- La definizione di ogni oggetto deve essere **unica**.

Dichiarazione di variabili

- Ha la forma:

  [<classe di memoria>] <tipo> <nome>`;`
- La classe di memoria non è obbligatoria.
  - Per default è `auto` se la dichiarazione compare entro una funzione, globale se al di fuori.
  - Se la classe non è `extern`, la dichiarazione diventa anche definizione.
- Si possono dichiarare più variabili con lo stesso tipo base, separando i nomi con virgole.

Inizializzazione

- Una variabile può essere inizializzata nella definizione, facendo seguire al nome l'operatore = e il valore.
  - Se la variabile è globale, il valore deve essere una costante e l'inizializzazione avviene **prima della partenza del programma**.
  - Se la variabile è automatica, il valore può essere una qualsiasi espressione e l'inizializzazione avviene **ogni volta che si entra nel blocco o funzione**.
    - Equivale a un assegnamento.

Esempi di dichiarazioni

```
int  i, j;
unsigned short k;

int i, j = 0;  // Inizializza solo j!

float f = 1.0 / 3.0;

int k = m + n; // Solo se k è automatica
int k;         // Codice equivalente
k = m + n;

const int zero = 0; // Non modificabile
```

Valore iniziale delle variabili

- Le variabili non esplicitamente inizializzate hanno valore:
  - **zero**, convertito al loro tipo, se globali;
  - **indefinito** se automatiche.
    - Può essere causa di errori difficili da trovare.