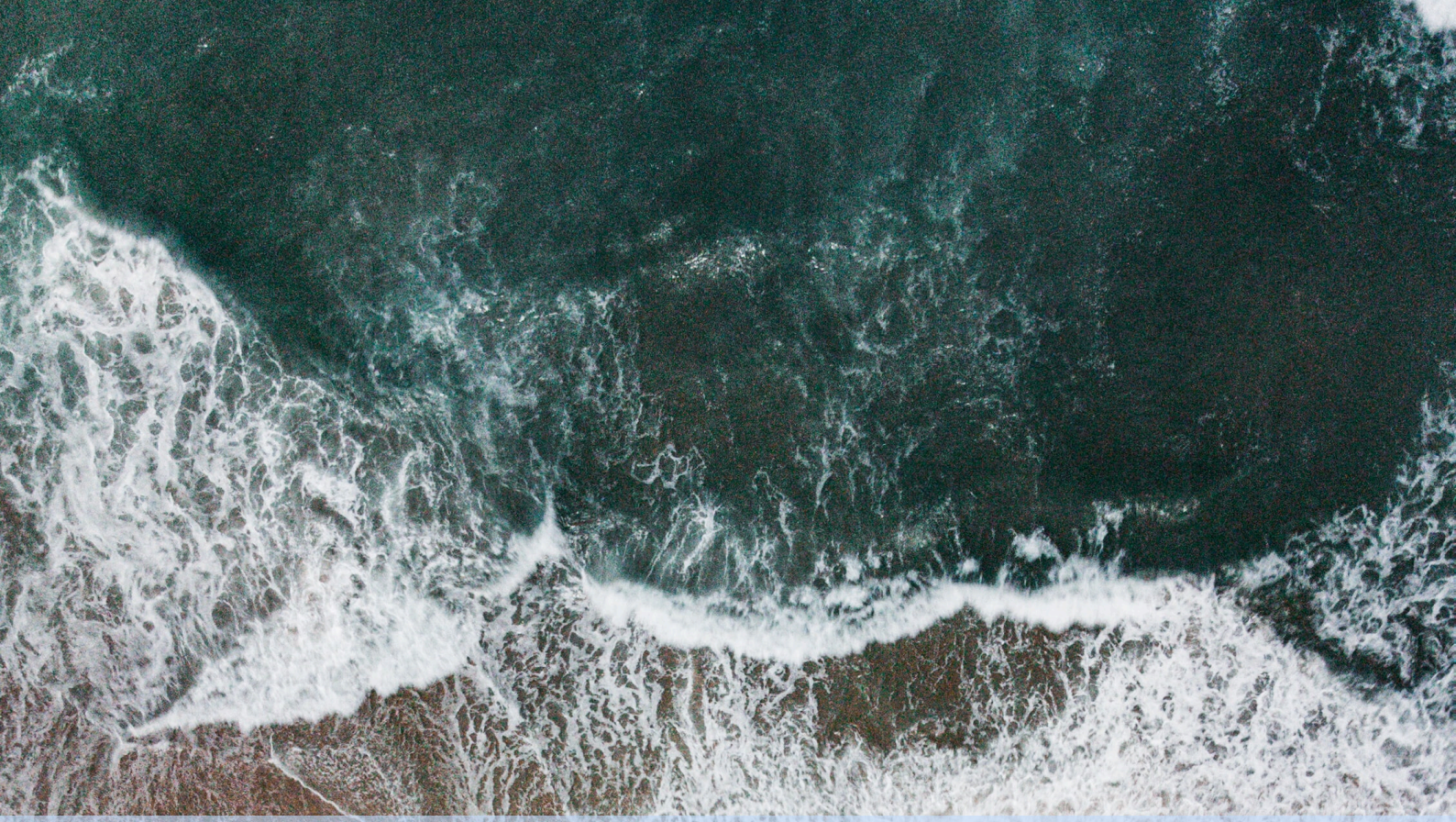


Not In The Book

Introduzione alla Programmazione

Appunti e approfondimenti

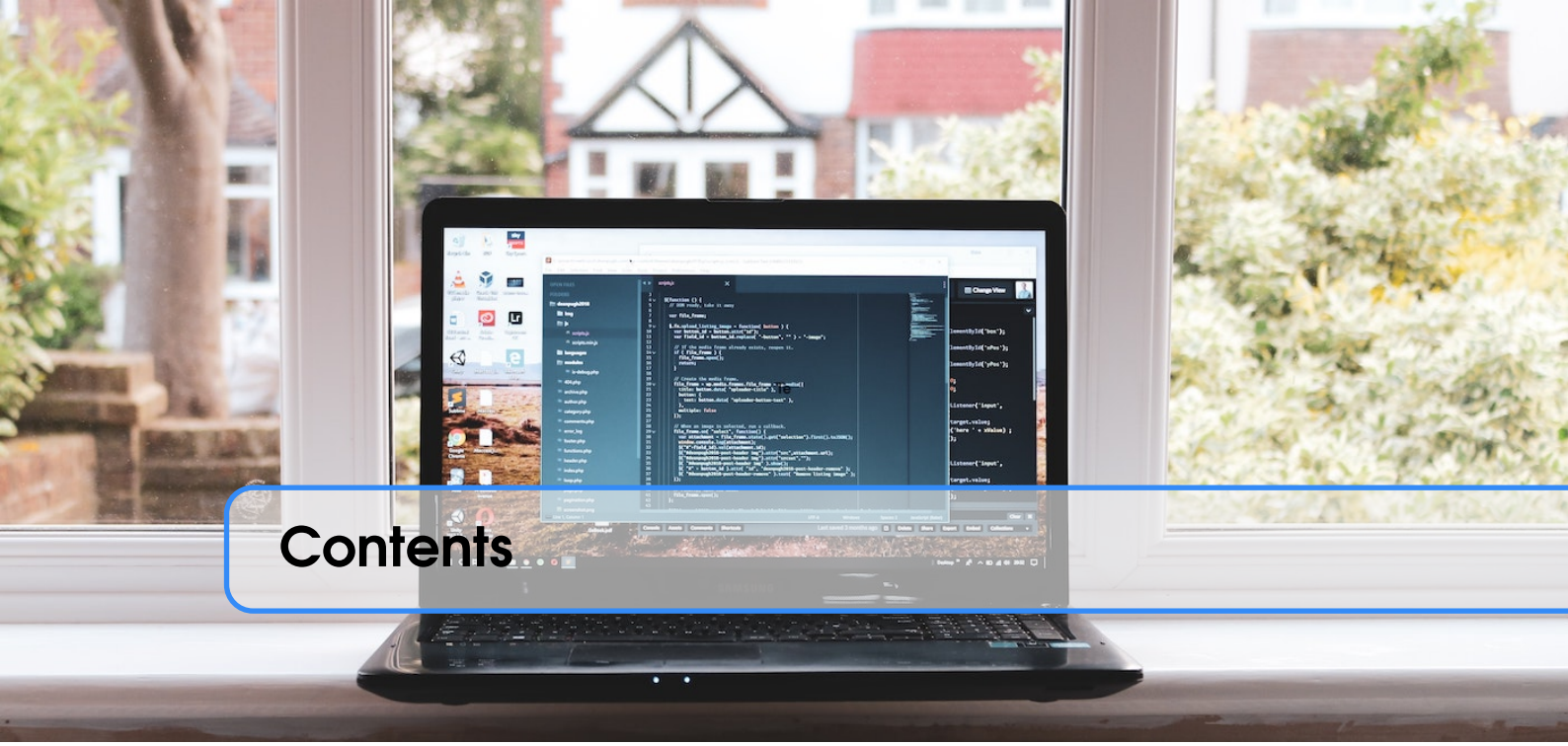
Author: Claudio Lucchese.

Last modified: September 25, 2024.

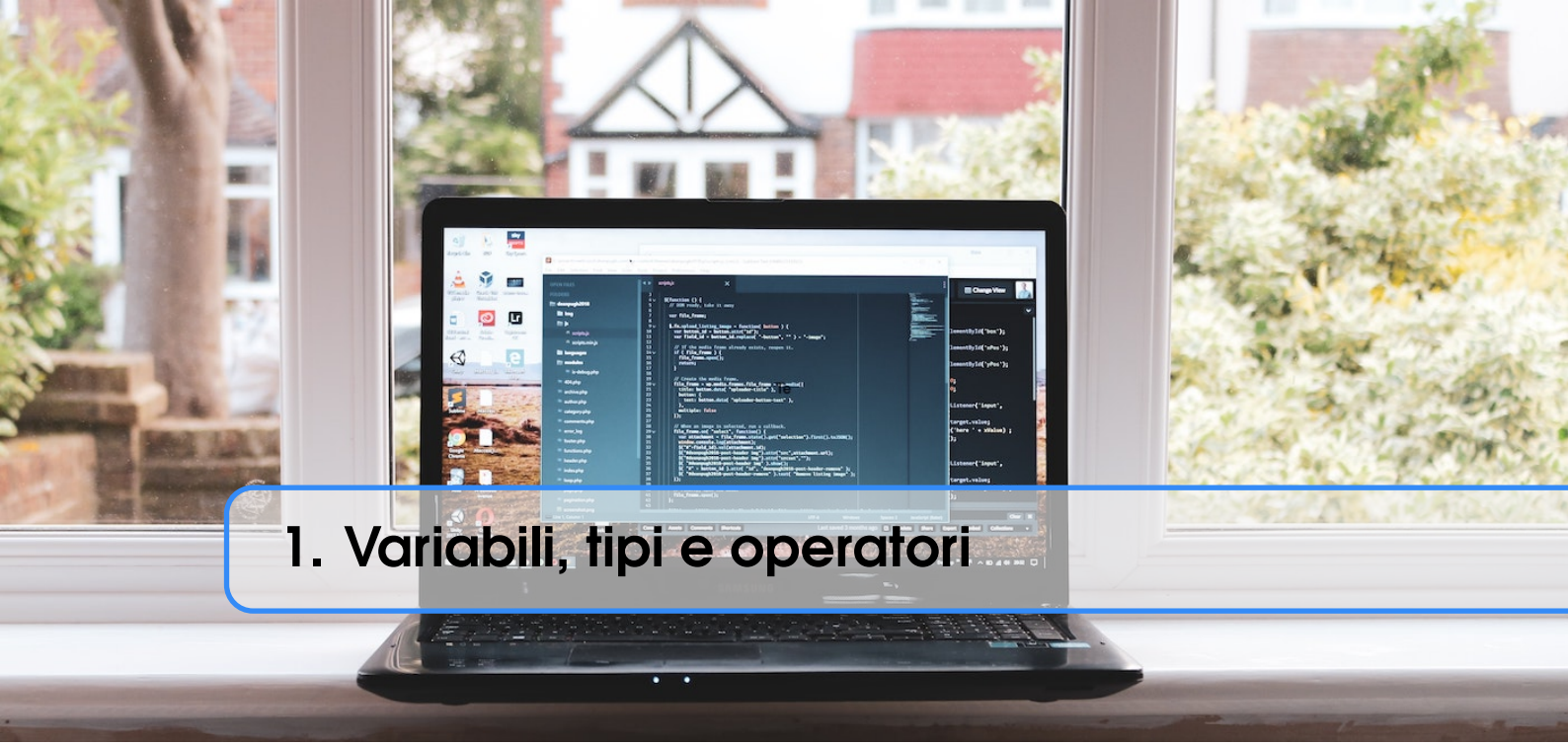
Prefazione

Questi appunti non sono da intendere come un libro di testo completo di tutti gli argomenti. Questo testo contiene degli approfondimenti specifici su alcuni argomenti che meritano una discussione più dettagliata.

Nel leggere questo testo, chiediamo quindi agli studenti di far sempre riferimento al libro di testo ed alla documentazione del linguaggio https://en.cppreference.com/w/c.



Contents



1. Variabili, tipi e operatori

1.1 Rappresentazione dell'informazione: il tipo double

Sappiamo che con il tipo `double` possiamo rappresentare un intervallo di numeri reali molto ampio. In realtà, il numero totale di numeri distinti rappresentabili è pari (circa) al numero di configurazioni diverse che possiamo costruire con 64 bit, e quindi 2^{64} (2^{32} per il tipo `float`). Questo significa che introdurremo delle approssimazioni nei nostri calcoli.

Consideriamo il codice seguente e il relativo output.

Sorgente:

```c
#include <stdio.h>

int main()
{
    if ( 0.1 + 0.1 + 0.1 == 0.3 )
        printf("Uguaglianza soddisfatta\n");
    else
        printf("Uguaglianza NON soddisfatta\n");
    return 0;
}
```

Output:

```
Uguaglianza NON soddisfatta
```

La difficoltà di rappresentazione nell'esempio sopra è dovuta all'uso della base 2 nella rappresentazione binaria. Così come non possiamo rappresentare il risultato di 1/3 (=0.3333...) in base 10 con un numero finito di cifre perché 3 non è un divisore della base 10, così non possiamo rappresentare 0.1 = 1/10 = 1/(5×2) in base due con un numero finito di cifre perché 5 non è divisore della base 2.

In questi casi, è una buona pratica non confrontare due numeri double/float con un operatore di uguaglianza, ma piuttosto misurare se la differenza tra i due numeri è inferiore ad una certa soglia di tolleranza. Ad esempio, in certe applicazioni, potremmo considerare due valori double equivalenti se la loro differenza è minore di 10^{-9}.

1.2 Operatore di assegnamento

L'assegnamento è un un'espressione così come la somma di due variabili. Questo significa che così come l'espressione `a+b` produce un valore corrispondente alla somma dei due addendi, anche l'espressione `x = z` produce un valore. Il valore prodotto corrisponde (anche nel tipo) all'espressione alla destra dell'operatore `=`. A differenza delle espressioni usuali, l'espressione di assegnamento

ha il *side-effect* (effetto collaterale) di cambiare il contenuto della memoria, memorizzando il contenuto dell'espressione a destra dell'operatore = nella locazione di memoria identificata dall'espressione a sinistra dell'operatore =.

Consideriamo il codice seguente con il relativo output.

```c
#include <stdio.h>

int main() {

    int a, b, c;

    a = b = c = 10;

    printf("a = %d \n", a);
    printf("b = %d \n", b);
    printf("c = %d \n", c);
}
```

```
a = 10
b = 10
c = 10
```

L'espressione

```c
a = b = c = 10;
```

è equivalente all'espressione:

```c
a = (b = (c = 10) );
```

perché l'operatore = è associativo da destra a sinistra. Quindi l'ordine di valutazione delle espressioni di assegnamento è quello (da destra a sinistra) corrispondente alle parentesi.

- prima viene valutata l'espressione `c = 10`, che risulta nel valore `10`
- dopo viene valutata l'espressione `b = (...)` e quindi `b = 10`, e anche questa espressione risulta nel valore `10`
- infine viene valutata l'espressione `a = (...)` e quindi `a = 10` che risulta nel valore `10`

Nota che l'espressione in ogni statement di assegnamento `x = z;` (nota il `;`) risulta in un valore che però non viene utilizzato ed è quindi ignorato.

Poiché il valore risultate dall'espressione di assegnamento è un valore valido come tutti gli altri, a questo possiamo applicare altre operazioni come nel codice seguente.

```c
#include <stdio.h>

int main() {

    int a, b, c;

    b = 20;
    a = b + ( c = 10 );

    printf("a = %d \n", a);
    printf("b = %d \n", b);
    printf("c = %d \n", c);
}
```

```
a = 30
b = 20
```

```
c = 10
```

1.3 Operatori di incremento e decremento

L'operatore di incremento/decremento può essere sia suffisso che prefisso: `i++` o `++i`. Possiamo dire che entrambe le espressioni hanno un *side-effect*, ovvero cambiano il contenuto della variabile `i` incrementandola di 1, e un *valore* prodotto che viene usato ai fini dell'espressione che contiene l'incremento. Nel caso prefisso, il valore prodotto è il valore della variabile incrementata, mentre nel caso suffisso il valore prodotto è il valore della variaible senza alcuna modifica.

Se consideriamo il codice seguente, al termine dell'esecuzione la variabile `c` contiene il valore `10`, ovvero il contenuto di `a` prima dell'incremento, mentre la variabile `d` contiene il valore `11` ovvero il contenuto di `b` dopo l'incremento.

```c
#include <stdio.h>

int main() {

    int a = 10, b = 20, c, d;
    c = a++;
    d = ++b;

    printf("a = %d \n", a);
    printf("b = %d \n", b);
    printf("c = %d \n", c);
    printf("d = %d \n", d);
}
```

```
a = 11
b = 21
c = 10
d = 21
```

Ma cosa succede nel codice seguente?

```c
i = ++i + i++;
i = i++ + 1;
```

In generale, è definito un ordinamento di valutazione per alcune espressioni del linguaggio, ma non per tutte. Nei casi sopra non è definito quale dei due incrementi venga eseguito per primo, o se l'incremento suffisso venga eseguito dopo l'assegnamento. E' una mancanza nella definizione del linguaggio. In questi casi, il comportamento dipende dall'implementazione dal compilatore o da strategie che questo applica di ottimizzazione del codice. Il risultato è quindi *non definito*: ogni compilatore e ogni esecuzione potrebbe portare a risultati diversi.

Sono assolutamente da evitare questi casi di ambiguità, e in generale un uso spinto di espressioni complesse e compatte a scapito della leggibilità del codice.

Per una trattazione dettagliata far riferimento a https://en.cppreference.com/w/c/language/eval_order.

1.4 Operatori Relazionali

Cosa calcola il codice seguente?

```c
#include <stdio.h>

int main() {
    int a=10, b=20, c=0, d=15;
```

```
5
6       int x = (a>b)*a + (a<=b)*b;
7       int y = (c>d)*c + (c<=d)*d;
8       int z = (x>y)*x + (x<=y)*y;
9
10      printf("%d", z);
11
12      return 0;
13  }
```

Il programma sopra visualizza il massimo tra le variabili `a`,`b`,`c` e `d`. qualunque sia il loro

Le espressioni relazionali come `a>b` vengono valutate in un valore Booleano pari a 0 o ad 1. Questo viene promosso a intero e poi usato per il calcolo del prodotto. Quindi l'espressione `(a>b)*a` è pari ad `a` se `a>b` ed è pari a 0 altrimenti. Quindi la variabile `x` conterrà il massimo tra `a` e `b`, mentre la variabile `y` il massimo tra `c` e `d`. In fine il massimo tra `x` e `y` viene memorizzato in `z`.