Operations involving *same* and *different* data types

- **Operations involving values from *same* data type**
    - Just like **Java**:
        - You can ***only*** perform **operations** of 2 values of the ***same* data type**
    - **Example:**

```
float A, B, C;        - Defines 3 float variables

A = 4;
B = 5;
C = A + B;
```

This is the similar situation as "adding apples and apples".

- **Operations involving values from *different* data types**
    - Just like **Java**:
        - **Operations** using values of the ***different* data type** must first **convert** one type into the other before the operation can be performed.
    - **Example:**

```
int A;       - Defines integer
float B, C;  - Defines 2 float variables

A = 4;     // Integer
B = 5;     // Float

C = A + B;  - The value of A is converted to FLOAT first
            - Then the addition is performed
```

This is the similar situation as "adding apples and oranges". We must convert the apples to oranges first before we can add

- **Converting between different types of data**
    - **Data conversion** must take place in the following **operations** involving **different data types**:
        - **Binary** (arithmetic) **operations**

          Example:

```
int   a;
float b;
```

```
a + b     // a in converted to a float
```

- **Assignment operations**

  Example:

  ```
  int   a;
  float b;

  a = b ;  // b in converted to a int
  ```

  Note:

  - **Yes**, this is **allowed** in **C**
  - In **Java**, you need to use **casting**:

    ```
    a = (int) b;    // Because b is float !
    ```

- **Warning: C is for adults *only* !!!**

  - **Fact:**

    - **C** was **designed** to **write** the **UNIX operating system** (they used to write **operating systems** in **assembler code** !!!)
    - The **designer** of **C** has **included** many ***features*** to make **system programming *easy***

      - But these **features** can ***also*** cause **major pain (harm)** to **programmers** that **do not** fully understand its **effects** !!!

  - Your **first feature**:

    - **C** will ***always*** perform an **assignment operation with *automatic* type conversion** on **non-pointer types**

      - **Even** when the **type conversion** results in ***loss* of accuracy**

    (That's because **Ritchie** hates to **type** the **casting operator**)

- **Making *automatic* conversion "explicit" with assembler programming**

  - **Fact:**

    - **Every computer (CPU)** has a number of ***conversion* instructions** (e.g., **int** ⇒ **float**, **int** ⇒ **double** and so on)

- We can **make** the **automatic convert** by the **C compiler** **explicit** by examining the ***generated*** **assembler code**:

  - The **C program**:

    ```
       int x;
       int y;

       main(int argc, char * argv[])
       {
          x = y;
       }
    ```

    will generate the following **Intel Code**:

    ```
    main:
    .LFB0:
            .cfi_startproc
            pushq   %rbp
            .cfi_def_cfa_offset 16
            .cfi_offset 6, -16
            movq    %rsp, %rbp
            .cfi_def_cfa_register 6
            movl    %edi, -4(%rbp)
            movq    %rsi, -16(%rbp)
            movl    y(%rip), %eax           move y into register "eax"
            movl    %eax, x(%rip)           move rigster "eax" to x
            leave
            .cfi_def_cfa 7, 8
            ret
            .cfi_endproc
    ```

  - A **C program** that uses ***different*** **data types**:

    ```
       int x;
       float y;

       main(int argc, char * argv[])
       {
          x = y;
       }
    ```

    will contain an ***appropriate*** **conversion instruction:**

    ```
    main:
    .LFB0:
            .cfi_startproc
            pushq   %rbp
            .cfi_def_cfa_offset 16
            .cfi_offset 6, -16
            movq    %rsp, %rbp
            .cfi_def_cfa_register 6
            movl    %edi, -4(%rbp)
            movq    %rsi, -16(%rbp)
            movss   y(%rip), %xmm0         Get y
            cvttss2si       %xmm0, %eax    Convert float to int
            movl    %eax, x(%rip)          Put in x
            leave
            .cfi_def_cfa 7, 8
            ret
            .cfi_endproc
    ```

Here's an explanation of the `cvttss2si` instruction: **click here**

- **Example Program:** (Demo above code)

  Example

  - **C program** Prog file **without** the need of conversion: click here
  - **C program** Prog file that **uses** conversion: click here

  How to run the program:

  - **Right click** on link and **save** in a scratch directory
  - To compile: `gcc -S convert?.c`
  - Examine **convert?.s**