# A taste of assembler programming

- A **single** assembler instructions will accomplish a very small amount of work.

  Each program statement in a high level language will typically require **multiple** assembler instructions to accomplish/complete.

- Example:

```
High level language statement:

   C = A + B;

Equivalent M68000 instructions:

   MOVE.L A, D0        Get integer from memory location A into D0
   MOVE.L B, D1        Get integer from memory location B into D1
   ADD.L  D0,D1        Add integers in D0 and D1
   MOVE.L D1,C         Store integer in D1 to memory location C
```

- You can see how **memory locations** are **reserved** for **variables** by the assembler when you assemble this program and look at the listing file a.lst: click here

```
a.lst:

  Address Content   Assembler source     Comment
  ======= =======   ================     ================================
   000000 2039      move.l A, d0         Get variable A into register d0
          0000
          0016
   000006 2239      move.l B, d1         Get variable B into register d1
          0000
          001A
   00000C D081      add.l  d1, d0        Add them, result is in d0
   00000E 23C0      move.l d0, C         Store result in variable C
          0000
          001E
   000014 4E71      nop

   000016 0000  A:  dc.l 4
          0004
   00001A 0000  B:  dc.l 15
          000F
   00001E ????  C:  ds.l 1
   000022           end

                        SYMBOL TABLE
                        ************
```

```
A      000016             B      00001A             C      00001E
```

- Look carefully in the above listing for:

  - Label **A** is associated with memory location **000016** (see content of the Symbol Table)
  - The instruction **`MOVE.L A,D0`** is **translated** into an instruction that ***uses*** the **memory** at address **00000016**, as given in the above listing (see **instruction** at the address 000000)

  ---

  - **Thus:**

    - the assembler will **replace** all **symbolic names** used in the program by the **associated address** !!!

  ---

  - The same is true for label **B** and **C** (I highlighted C in red)

---

- **Lesson** from this small example:

  - A statement in a high level language expresses a **unambigous** result

    C = A + B means:

    - store the sum of variables A and B in variable C

  - Assembler instructions are used to achieve this result

    We must first determine the **exact sequence of operations** that achieve the desired result !

    Steps to achieve "store the sum of variables A and B in variable C":

    - get value in variable A
    - get value in variable B
    - add them
    - put the result in variable C

- The **most** difficult part in assembler programming is **getting the value from variables**

  - It is quite easy to get values from simple variables, like

    ```
    int A;
    ```

  - It is quite complex to get values from more complex data structures, like:
    - An array

      ```
      int A[100], i, j;

      A[i]
      ```

```
A[7*i + 9*j]
```

    - Array of arrays

      ```
      int A[100], B[10], C[10], i;

      A[ B[i] ]
      A[ B[ C[i] ] ]
      ```

    - An linked list

      ```
      class List
      {
         int value;
         List next;
      };

      List head;

      head.next.next.value
      ```

  - Experience in computer building led to the understanding of a number of **address modes** that are most helpful to aid high level programming languages to **get operands** from the registers (in the CPU) and from memory.

- Addressing modes:

  - Immediate mode
    - allows the computer to get a constant as operands

  - Direct mode
    - allows the computer to get operands that are simple variables stored in a register or in memory

  - Indirect (without displacement)
    - allows the computer to get to objects through an address/reference
    - allows the computer to get to static simple variable

  - Indirect with displacement
    - allows the computer to get to local variables
    - allows the computer to get to array of simple elements
    - allows the computer to get to members in an object
    - allows the computer to get to linked lists

  - Indirect with index and displacement
    - allows the computer to get to of arrays of complex elements

- Only the first 3 addressing modes are **necessary** (immediate, direct and indirect without displacement)

  But if only these 3 addressing modes are available, you will have to compute the address "manually" using assembler instructions (ADD and MULT), which can be quite clumbersome.

  Some computers (mainly RISC - Reduced Instruction Set Computers) deliberately omit the "Indirect with index and displacement" mode because programs **rarely** need to access arrays with complexe elements. The CPU designers use the "free up" space on the CPU (which would be needed to execute the "Indirect with index and displacement" mode) for other more useful functions.

- Each computer has its own way of expressing the addressing modes.

  M68000 has all the above addressing modes....

  SPARC does not have "Indirect with index **and** displacement" only "Indirect with index"

---

We will now learn to express the addressing mode in M68000 and also explain what each addressing mode means.

Address modes are used in all assembler instructions.

To keep the focus on **addressing mode** (and not on the instructions), I will mainly use the **move** instruction to illustrate the concept of addressing mode. various

Addressing mode is the first of two hairy topics in this course (the other is ***recursion***).

Make sure you understand addressing modes VERY WELL or the rest of your CS255 experience will be misserable....