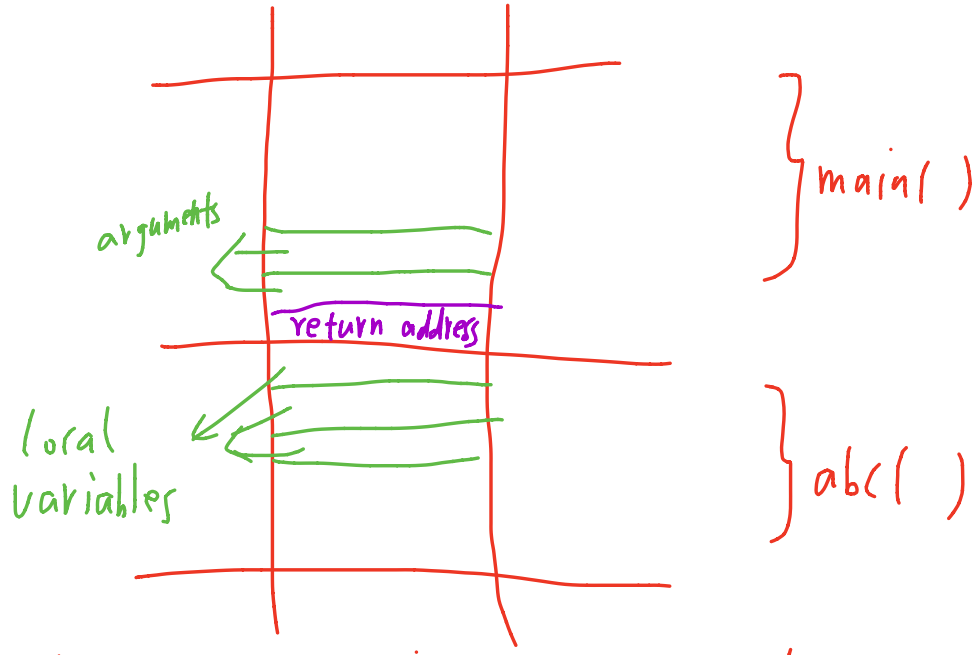


If this program is a server, it usually has root privilage,

2. gcc -c main.c

→ main.o (hasn't been linked up)

gcc -c def.c

→ def.o

gcc abc.c main.o def.o

(skip compile main.c, def.c,
directly link up main.o, def.o)

just compile abc.c, because we
just change abc.c at this time.

Save time to compile all files.

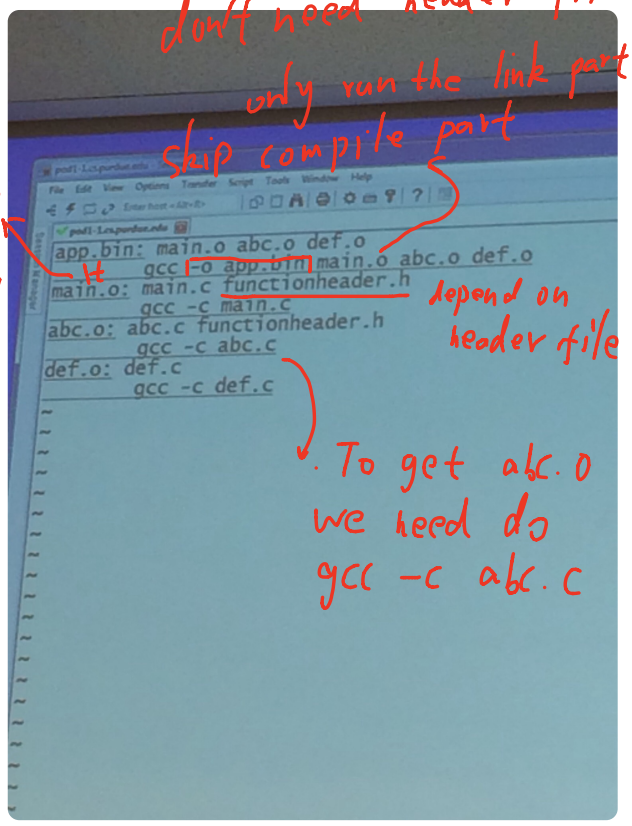
→ a.out

3. Make file

Tab

dependence

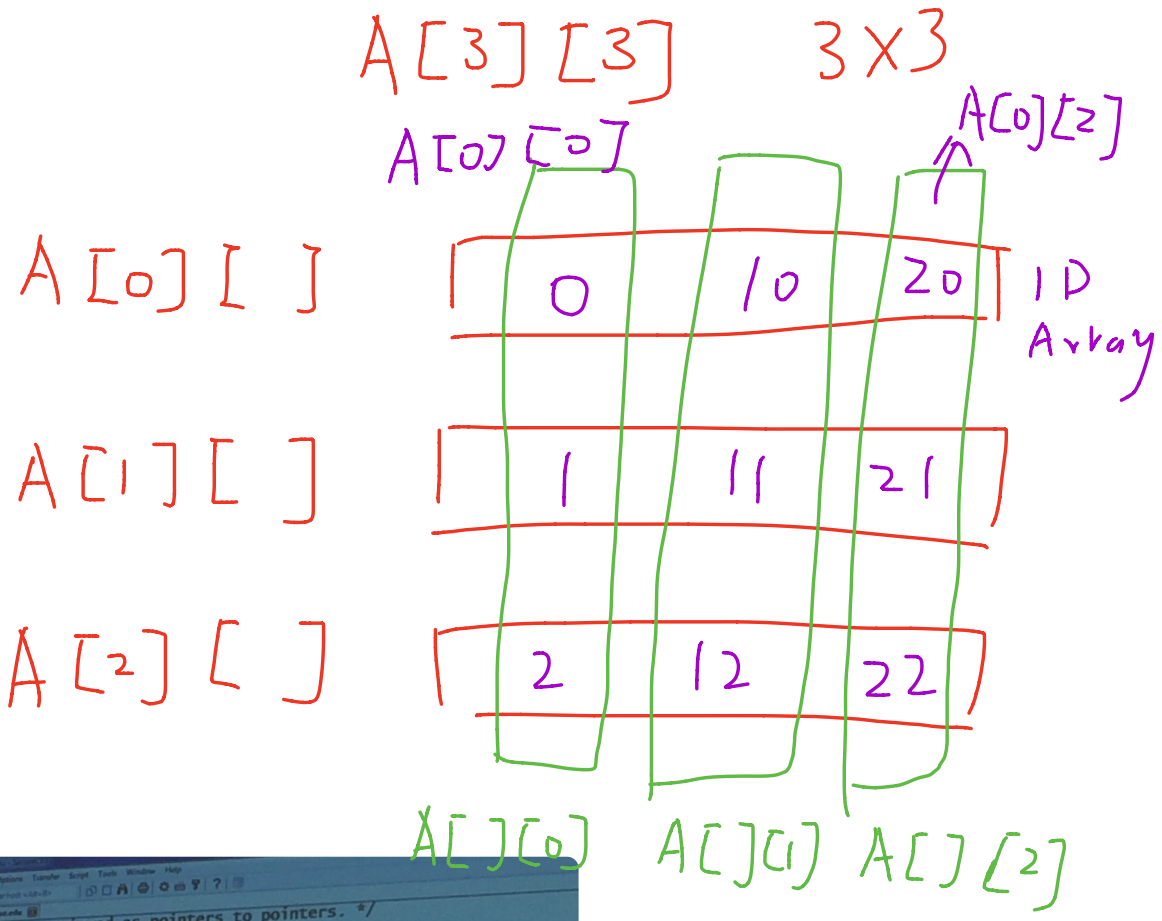
action




```
rm *.o
rm a.out
```

Make will keep track of the modify time stamp of file; recompile the changed file only

4. 2-D array

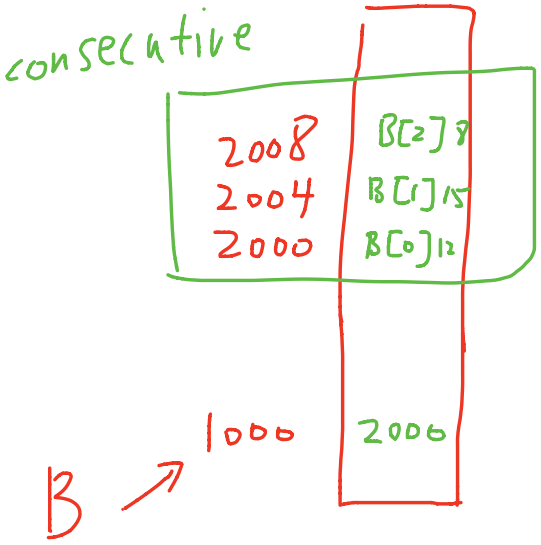



```
/* 2-D arrays viewed as pointers to pointers. */

#include <stdio.h>

void main()
{
int i, j;
int A[3][3];

// initialize 2-D array

  for (i=0; i<3; i++)
        for (j=0; j<3; j++)
          A[i][j] = i + 10 * j;

// print 2-D array
  for (i=0; i<3; i++)
        for (j=0; j<3; j++)
          if (j != 2)
                printf("A[%d][%d] = %d  ", i, j, A[i][j]);
          else
                printf("A[%d][%d] = %d\n", i, j, A[i][j]);

  printf("%d\n", **A);
  printf("%d\n", *((*A)+1));
  printf("%d\n", *(*A+2));
  printf("%d\n", **(A+1));
  printf("%d\n", **(A+2));
}
~
"main.c" 30L, 518C
```

if(j != 2) if it's not the last column just print A[i][j]

else it is the last column, print extra '\n'

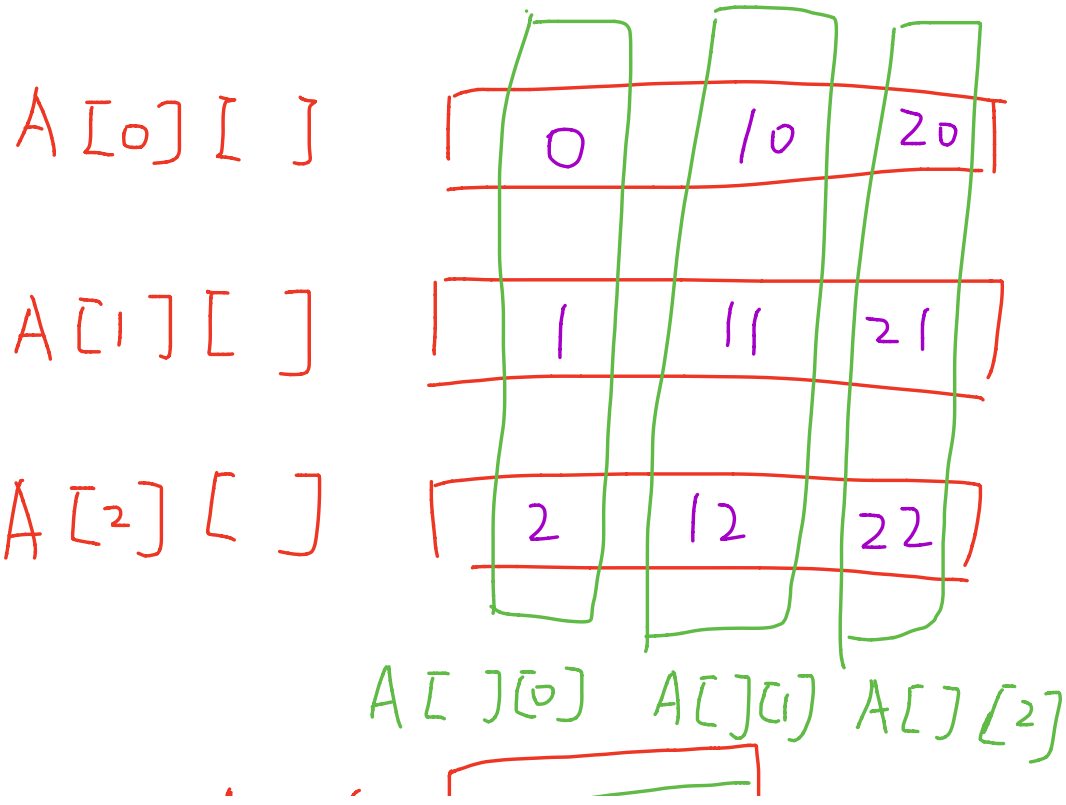
1-D int B[3] = {12, 15, 8}

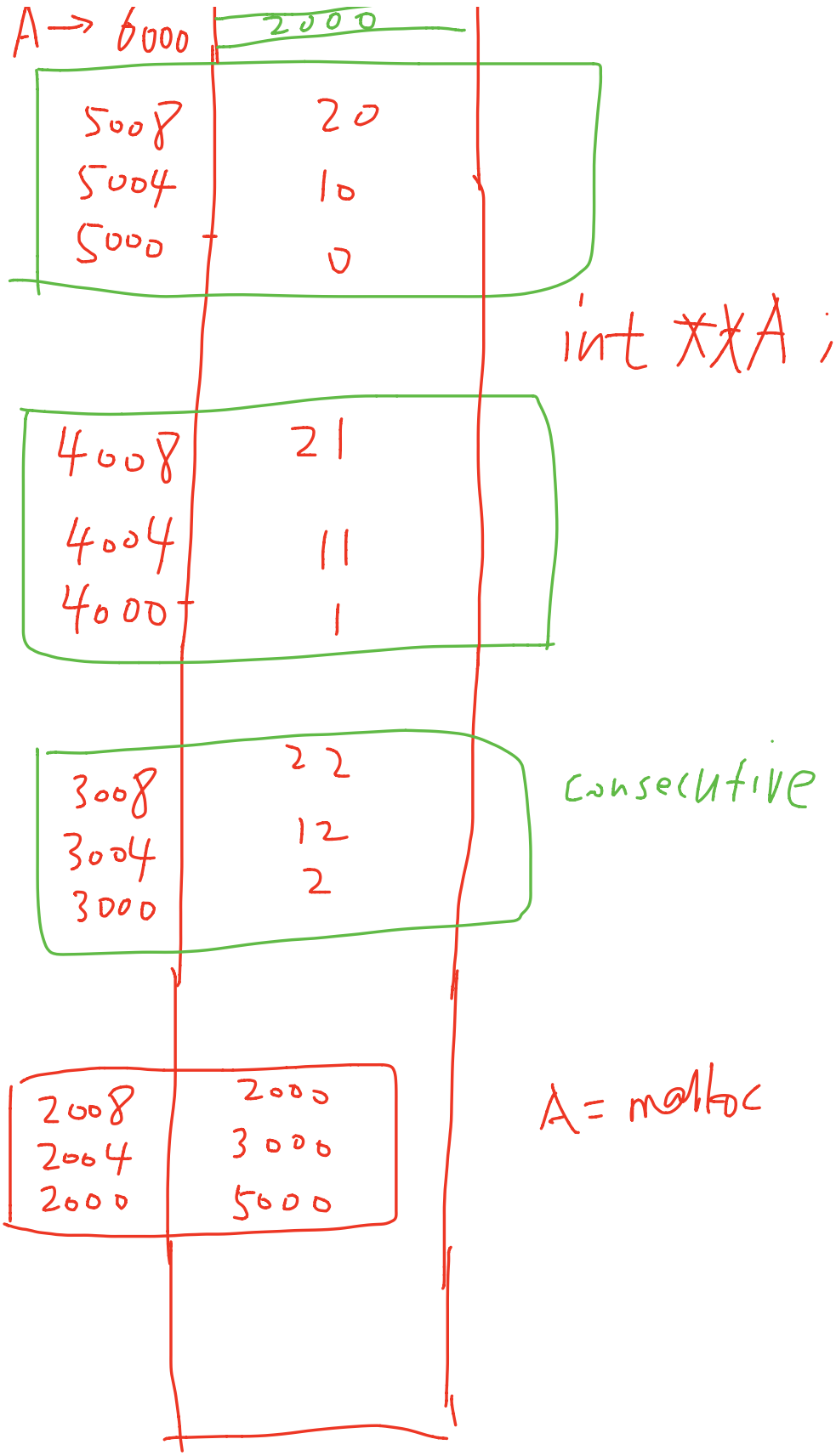


*B → *(2000) = B[0] = 12

*(B+1) → *(2000+4) = *(2004)
= B[1] = 15
since its int array

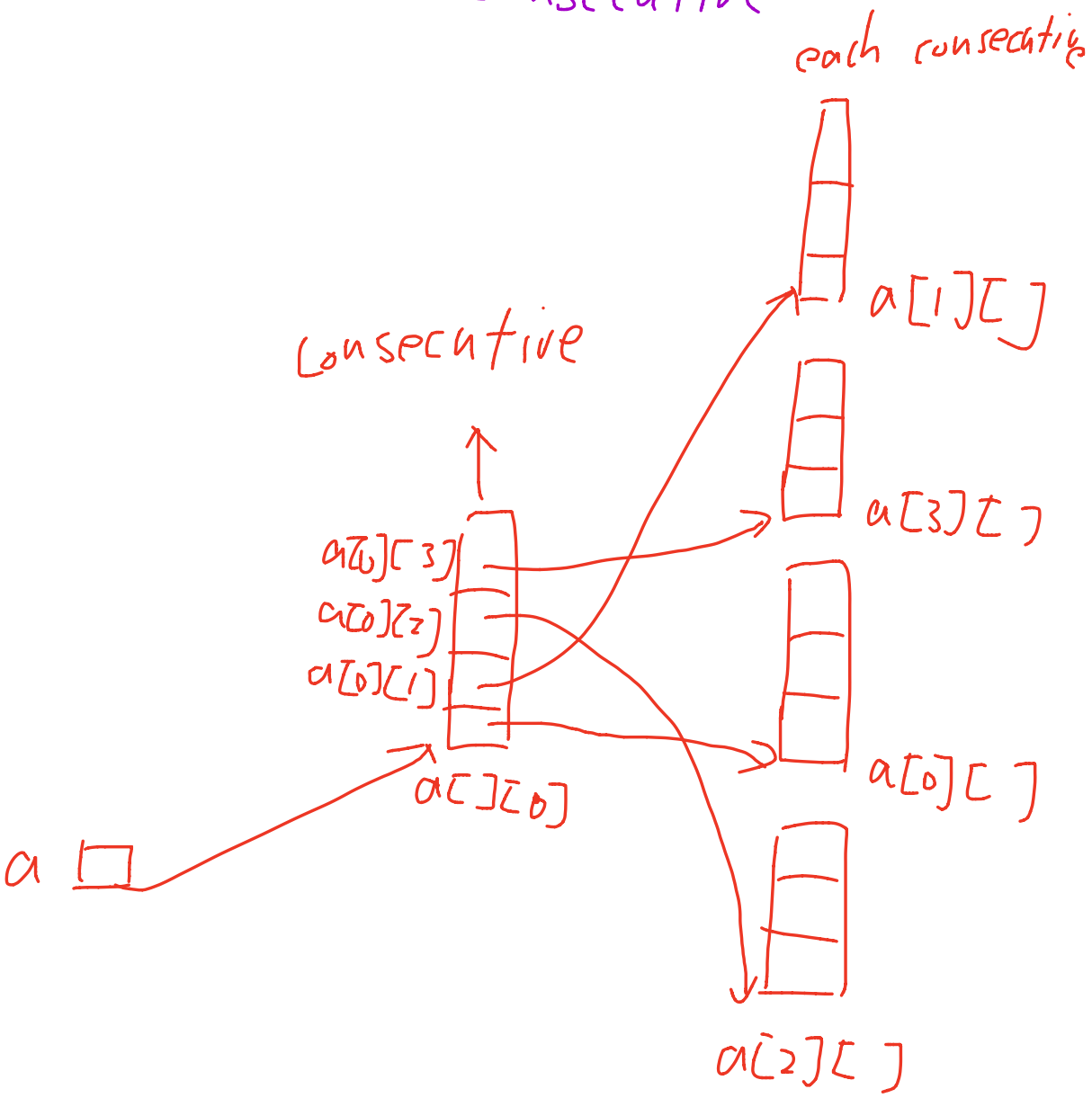
A[3][3] 3x3




A → 6000
2000
5008
5004
5000
20
10
0
int **A;
4008
4004
4000
21
11
1
3008
3004
3000
22
12
2
consecutive
2008
2004
2000
2000
3000
5000
A = malloc

A[4][3]

key point: these rows are consecutive



C++ Notes: 2-D Array Memory Layout

Two kinds of multi-dimensional arrays.

There are two basic ways of implementing 2-dimensional arrays: *rectangular, sequential, two-dimensional arrays* and *arrays of arrays*. Some languages use one method, some another, and some both. C++ allows both methods, and each has its advantages and disadvantages.

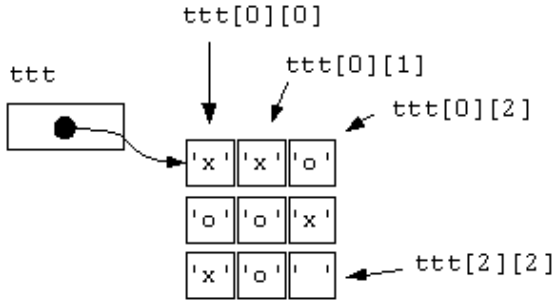
- **rectangular sequential arrays**. In this case the rows are laid out sequentially in memory. This provides simplicity in memory management and a slight speed advantage. This is the kind of array C/C++ creates by default.
- **Arrays of arrays**. In this style each element of a one-dimensional array is a *pointer* to another array. This is a common way to create an array of C-strings (a zero-terminated array of characters). This is also useful if the rows are uneven length or are dynamically allocated. The disadvantage is that memory management is more complex, often requiring dynamic allocation and deallocation. This is the only kind of multi-dimensional array in Java. See Arrays of Arrays.

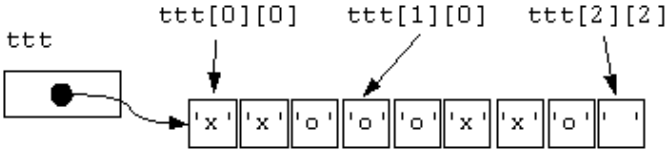
Rectangular arrays allocated in memory by row

Let's see how the memory is allocated for this array.

```
char ttt[3][3] = {{'x', 'x', 'o'},
                  {'o', 'o', 'x'},
                  {'x', 'o', ' '}
                 };
```

The memory for this array could be visualized as in the diagram to the right, which identifies a few cells by their subscripts.





Because memory is addressed linearly, a better representation is like the diagram to the left.

Computing the address of an array element

C++ must compute the memory address of each array element that it accesses. C++ does this automatically, but it helps to understand what's going on "under the hood". Assume the following declaration:

```
char a[ROWS][COLS];  // assume ROWS and COLS are const ints
```

Because arrays are laid out in memory by row, each *row length* is COLS (the number of columns is the size of a row). Let's assume that you want to find the address of `a[r][c]`. The *baseAddress* of the array is the address of the first element. The *rowSize* is `COLS` in the above example. The *elementSize* is the number of bytes required to represent the data (typically 1 for `char`, 4 for `int`, 4 for `float`, and 8 for *double*.

```
address = baseAddress + elementSize * (r*rowSize + c);
```

Note

- The number of rows (`ROWS`) is not used in the computation.
- Because the number of rows is not used, there is no need to pass it when declaring a formal array parameter for a two-dimension array.

```
/* 2-D arrays viewed as pointers to pointers. */

#include <stdio.h>

void main()
{
int i, j;
int A[3][3];

// initialize 2-D array

  for (i=0; i<3; i++)
        for (j=0; j<3; j++)
          A[i][j] = i + 10 * j;

// print 2-D array
  for (i=0; i<3; i++)
        for (j=0; j<3; j++)
          if (j != 2)
                printf("A[%d][%d] = %d  ", i, j, A[i][j]);
          else
                printf("A[%d][%d] = %d\n", i, j, A[i][j]);

  printf("%d\n", **A);
  printf("%d\n", *((*A)+1));
  printf("%d\n", *(*A+2));
  printf("%d\n", **(A+1));
  printf("%d\n", **(A+2));
}
~
~
"main.c" 30L, 518C
```

A[0][0] → 9000 + 1 = 9004 (int)

A[0][1] → 9000 + 2 = 9008 (int)

A[1][0] → 3000 + 1 = 3004 (int)

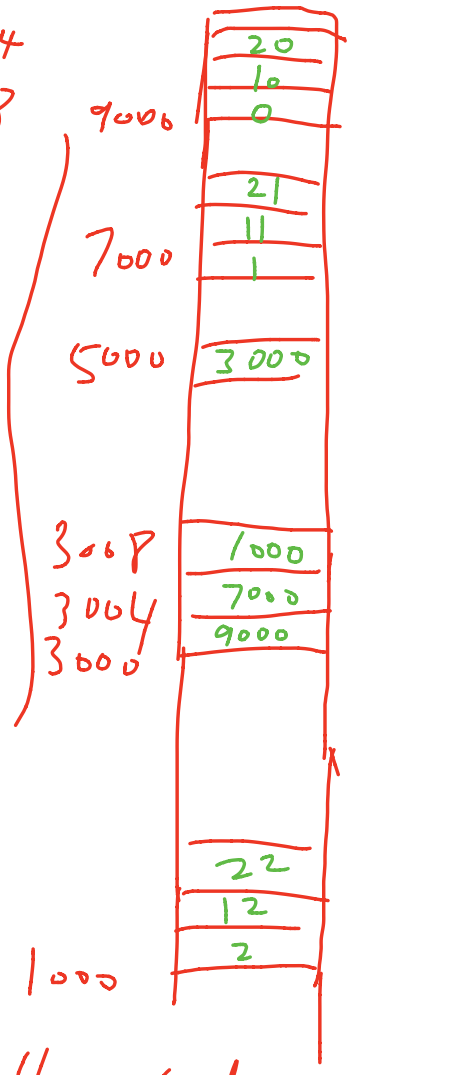
A[2][0] → 3000 + 2 = 3008

int A[3][3]

0 10 20

1 11 21

2 12 22



if You use

int A[3][3] then

9 spaces allocated

may be (9+3+1)

---

if you use

int **A,

only one space 5000 allocated

which contain a pointer