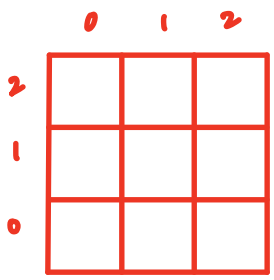
nD Arrays (Multidimensional Arrays)

Declaration: type name[#rows][#columns]

long matrix[3][3]



Array decay in multidimensional array:

Given an nD array matrix[i][j], matrix[i] decays to &matrix[i][0]

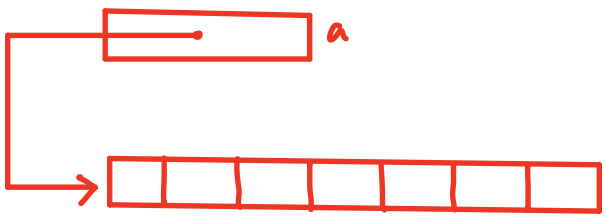
&matrix[i][0] is a pointer to long

matrix decays to &matrix[0]

&matrix[0] is a pointer to an array of long values.

notation for pointer to an array:

long (*a)[len_array]



Passing an nD array into a function (2D-array)

void foo(size_t nrows, size_t ncols, long matrix[][20])

void foo(size_t nrows, size_t ncols, long matrix[nrows][20])

```
void foo( size_t nrows, size_t ncols, long matrix[][20])
void foo( size_t nrows, size_t ncols, long (*matrix)[20])
```

exact subarray length must be passed in to the function

long (*a)[20] → points to an array of 20 elements
long *a[20] → an array of pointers to long
long *a[20] → an array of pointers to long

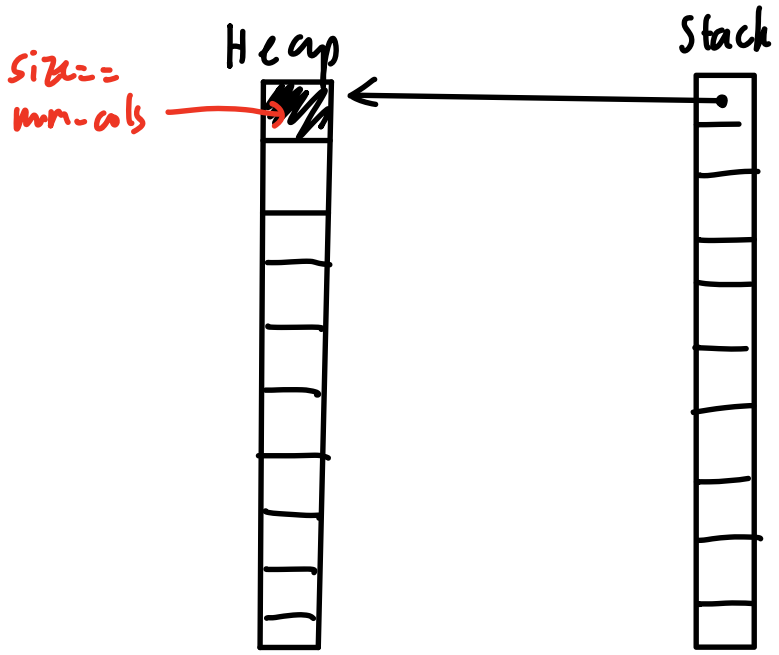
Indexing in nD arrays:
bucket[i][j] == *(bucket[i]+j)

Heap allocation of memory in nD arrays

```
long *buckets[10]
for (long i=0; i<10; i+=1){
    buckets[i]= malloc(k* sizeof(long));
    if (buckets[i]==NULL){
        cs1010_println_string("unable to allocate");
        for (long j=0; j<i , j+=1){
            free(buckets[j]); ← to free all previously allocated memory
        }
        return 1;
    }
}
```

Easier way to allocate memory

```
buckets[0] = malloc (10* num_of_cols* sizeof(double));
if (buckets[0] == null) {
        cs1010_println_string ("unable to allocate");
        return 1;
}
for (size_t[i]= 1 ; i<10 ; i += 1 ) {
      buckets[i] = buckets[i-1] + num_of_cols;
}
free(buckets[0]);
```



If # rows & # columns are unknown

```
double **canvas;
size_t num_rows;
size_t num_cols;

canvas = malloc(num_rows * sizeof(double *))
for (size_t i = 0; i < num_rows; i += 1){
    canvas[i] = malloc(num_cols * sizeof(double));
    if (canvas[i] = NULL){
        :

    }
}

for (size_t i = 0; i < num_rows; i += 1){
    free (canvas[i]);
}
free (canvas)
```

deallocate the memory that each pointer in canvas points to, to prevent people from crashing your program

C-Preprocessor

C Program → pre-process → C program → compile (etc) → Machine Code → Run

clang -E → will return C program after pre-processing

#include } pre-processing directives
#define }

#define pre-processing directive

Used to define constants

MACROS: #define fn <fn_body>

Role of macro - purely text substitution (does not check type)

Properly defining a macro:
#define SQUARE(x) (x)*(x)
↑
include parenthesis

ASSERT MACRO #include <assert.h>

macros that help user find bugs in a program

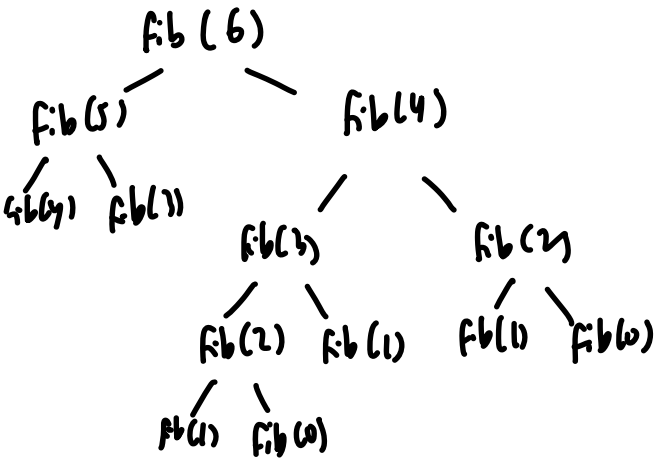
We can assert that an input/parameter must meet certain condition in the function body. It will raise in error if the assertion fails.

Program Efficiency

1) Avoid redundant work
Check program's efficiency in "WORST CASE SCENARIO". Example very large prime number in is_prime function.

2) No duplication
For example, in finding range of a list, we can combine two loops (one for max one for min) into just one loop (that finds both max & min)

Fibonacci Program (Recursive Implementation)



$O(2^n)$

Fibonacci program (Iterative implementation)

Number of steps in terms of n: $2 + 3 + 3(n-2) + 1$

- 2: boolean checks
- 3: initialising a, b & c
- $3(n-2)$: looping until result
- 1: return

$= 5 + 3n - 6 + 1$

$= 3n$ ← we can ignore the constant

The program takes $O(n)$ time.

Big-O notation

To find big O, (1) find number of steps taken by a program in terms of n, (2) drop the terms with lower rate of growth & (3) drop multiplicative constant

Example: $O(\frac{n^4}{10} + n^3 + n^2 + 2n + 1) \sim O(n^4)$

Comparing rate of growth

If $f(n)$ grows faster than $g(n)$, we can find constants n_0 and c such that $f(n) > c\,g(n)$ for all $n \geq n_0$

Example:

$f(n) = n^n$, $g(n) = 2^n$

$n_0 = 2$ $\quad f(n_0) = g(n_0)$

$n_0 = 3$ $\quad f(n_0) > g(n_0)$

Example selection sort: $O(n^2)$

we loop through the elements in the list $n-2$ times to sort

we loop through the elements in the list a further $n-2$ times to find max within the outer loop

$(n-2)*(n-2) = n^2 - 4n + 4 \quad \sim O(n^2)$