C

Binary – Decimal

7 6 5 4 3 2 1 0

10110010 b

$\Rightarrow 2^7 \cdot 1 + 2^6 \cdot 0 + 2^5 \cdot 1 + 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 0$

$= 2^7 + 2^5 + 2^4 + 2^2$

$= 128 + 32 + 16 + 4 = 180d$

Decimal → Binary:

			Remainder
371d / 2	→	185	1
185 / 2	→	92	1
92 / 2	→	46	0
46 / 2	→	23	0
23 / 2	→	11	1
11 / 2	→	5	1
5 / 2	→	2	1
2 / 2	→	1	0
1 / 2	→	0	1

101110011

1. Variables, Literals, and Constants

- Variables : To indicate storage area
  Each variables must be given data type

Naming rule: can only have alphabets, numbers, and underscore
cannot begin with number

- Literals : Fixed values that cannot be assigned different values.
  +) Integers — Decimal (0, -9, 22)
  Octal (021, 077, 033)
  Hexadecimal (0x7f, 0x2a, 0x521, ...)
  +) Floating-point (-2.0, 0.002, -0.22 E-5)

  +) Characters : created by enclosing a single character inside quotation marks ('a', 'm', 'b')

  +) Escape Sequences

Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null character

+) String ⇒ Double quote marks

※ Constants create variables whose value cannot be changed.

`const int` Light_speed = 299792458;

Light_speed = 2500 // Error! Light_speed is a constant

NOTE : Variable Type cannot be changed once it's declared.

```
int number = 5;      // integer variable
number = 5.5;        // error
double number;       // error
```

2. Data Types

In C, data types are declarations for variables. This determines the type and size of data associated with variables.

Basic Types

Type	Size (bytes)	Format Specifier
`int`	at least 2, usually 4	`%d`, `%i`
`char`	1	`%c`
`float`	4	`%f`
`double`	8	`%lf`
`short int`	2 usually	`%hd`
`unsigned int`	at least 2, usually 4	`%u`
`long int`	at least 4, usually 8	`%ld`, `%li`
`long long int`	at least 8	`%lld`, `%lli`
`unsigned long int`	at least 4	`%lu`
`unsigned long long int`	at least 8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	at least 10, usually 12 or 16	`%Lf`

+) int ⇒ We can declare multiple integer values.

int id, age = 20;

4 bytes ⇒ 32 bits ⇒ Take 2^{32} distinct states from -2^{31} to 2^{31}

+) float & double ⇒ Real numbers
We can also use 22.46e2 (= 2246)
float is 4 bytes, double is 8 bytes

Print to 6 decimals
⇒ %.nf / %.nlf
to print to n decimals

+) char : Character type variable

⍟ If we use %c, it will print z
If we use %d / %i, it will print ASCII value of z (122)

+) void : incomplete type, means "nothing" or "no type".
A function if returns nothing, will return void.
(We can create variables of void type)

+) short and long
If large number is needed, use long / long long → int
long double → float.

+) signed and unsigned are type modifiers. You can alter the data storage of a data type by using them.

signed - allows for storage of both positive and negative
unsigned - ——————————— only positive numbers.

```
// valid codes
unsigned int x = 35;
int y = -35;  // signed int
int z = 36;  // signed int

// invalid code: unsigned int cannot hold negative integers
unsigned int num = -35;
```

+) Derived data types : arrays, pointers, function types, structures,.

- bool type
- Enumerated type
- Complex type

3. C Input Output (I/O)

+) Output printf() function `printf("string: %Specifier", variable)`

- All valid C programs must contain the `main()` function. The code execution begins from the start of the main() function
- To use printf() in our program, we need to include `stdio.h` header file using the `#include <stdio.h>` statement
- The `return 0;` statement inside the main() function is the "Exit status" of the program. It's optional!

To print variables, we use format specifier corresponding to variable type (%d, %i, %f, ...)

+) Input scanf() function `scanf("%specifier", &variable)`

NO STRING LIKE PYTHON!

- We have to create variable type
- Use scanf function to take input from user
  Note the "%specifier", &variable ⇒ &variable gets the address of the variable, and value entered is stored in that address.

+) Multiple Values ⇒ User need to separate entries by space, tab, new line

```c
#include <stdio.h>
int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");

    // Taking multiple inputs
    scanf("%d%f", &a, &b);

    printf("You entered %d and %f", a, b);
    return 0;
}
```

4. C Comments

+) Single line //

+) Multiple line /* ... */

5. C Operators

NOTE • In * and / , the output — float if either operands are float
int if both operands are int.

• Modulo division % can only be used with integers

Increment and Decrement Operators . ⇒ Use only when stand alone to avoid confusion

Prefix :

++a → increase a by 1 then return the value
--a → decrease a by 1

Post fix :

a++ original value returned, then increase a by 1
a-- decrease a by 1

Assignment Operators (= , +=, -=, /=)

Relational Operators (==, >, <, !=, ...)

True → 1 False → 0 returns 0 or 1.

Logical operators

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

(More on Conditional Statements)

Precedence

e.g. If we use Arithmetic Operators, the precedence follows:

B racket
E xponents
D ivision
M ultiplication
A ddition
S ubtraction

But For assignment operators, evaluate Right to left

```
int i = 1, j = 2, k = 3;
i = j = k = 4;
```

k = 4 ①
j = 4 ②
i = 4 ③

Type Cast

We mentioned earlier if 2 types are mixed in an operation, the resultant type is more accurate type.

If we force a type:

```
int x = 5.7          → x stores 5
double z = (double) 3 / 2        → z stores 1.5
double z = (double) (3 / 2)      → z stores 1.0
                     (1)
```

Summary arithmetic BEDMAS, left to right
assignment right to left.

Math library

`# include <math.h>`

⇒ **Constants**

- `M_PI` = π
- `M_E` = e

Functions

- `sqrt (double x)` — $\sqrt{x}$

⇒ **double**

If put int, implicit conversion will occur

- `pow (double x, double y)` — x^y
- `exp (double y)` — e^y
- `log (double x)` — $\ln(x)$
- `log10 (double x)` — $\log_{10} x$
- `fabs (double x)` — $|x|$
- `sin (double x)` — $\sin(x)$ (radians)
- `cos (double x)` — $\cos(x)$ (radians)
- `fmax (double x, double y)` — larger
- `fmin` — smaller
- `floor (double x)` — largest int <= x
  ⇒ floor (-5.3) → -6
- `ceil (double x)` — smallest int >= x
- `fmod (double x, double y)` — $x - (x // y * y)$

mod % for doubles

⇒ **rint** (double x) ⇒ rounds to nearest int. Since it returns a double, it appends / converts / typecasts the int to double to return.

round (x)

Random Numbers

```
# include <stdlib.h>
```

`rand()` ⇒ random int

BUT Pseudo-random number generator !!!

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main()
{
    int c = rand();
    printf("The number is: %d\n", c);
    printf("The number is: %d\n", c);
    printf("The number is: %d\n", c);
    return 0;
}
```

⇒

```
The number is: 1804289383
The number is: 1804289383
The number is: 1804289383
```

Everytime we run the code, the same set of random numbers are generated.

⇒ If we want another set of random numbers? Using a different "seed"

In C, we pick the seed using `void srand(?)` ⇒ 1 by default

unsigned int "seed"

Now that the seed is changed, I'll have to pick it again everytime or it will fall into the same set again.

⇒

```
#include <time.h>

time_t time(time_t * t)
```

Note that time(NULL) returns time in UNIX: # of seconds since Jan 1, 1970

⇒ If we do `srand(time(NULL))`, our set of numbers is different each time.

`srand(time(NULL))`

You only need to call only once at the beginning of the code. If you call it before every rand(), it will return the same random number.

- Limiting the random numbers ⇒ Use modulo %

e.g. %5 will always return a # between 0 & 4
%2 ——————————— 0 & 1

⇒ rand() % 2 ⇒ will produce a # 0 ~ 1

For random number between 15 & 17 ?
rand() % 3 + 15 ⇒ 15 ~ 17

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    srand(time(NULL));
    int r = rand()%20;
    int k = rand()%20+123;
    printf("The numbers are: %d, %d",r,k);
}
```

⇒

```
The numbers are: 17, 124
The numbers are: 16, 142
```

Summary: To generate a random number between [MIN, MAX]

→ `rand() % (MAX - MIN + 1) + MIN`

If else Statements

General form:

```
if ( condition ) {
    statement if True ;
}
else {
    statement if False ;
}
```

- Can compare int with double
- Compare characters will compare ASCII values (A < a)

AND &&, OR ||, NOT !

⇒ Needs {} if have >= 2 statements

```
# include <stdbool.h>
```

to use bool, true, false variables

De Morgan's Law

(I) !(A && B) = !A || !B

(II) !(A || B) = !A && !B

While Loops

```
while (condion) {
    statements ;
}
```

⇒ Condition has to be satisfied so that the body can be executed.

Do - while loop

```
do {
    statements ;
} while (condions) ;
```

⇒ Body will be executed at least once regardless of condition

⇒ Helpful when checking input from users

For loops and Nested Loops

With fixed number of iterations, there is a specific structure:
(i) initialization, (ii) condition, (iii) change variables in condition
⇒ For loop

```
for (<initialization>; <condition>; <alteration>) {
    <statements> ;
}
```

NOTE That the initialized variable is only within the scope of the for loop ⇒ Not defined outside

- Initialization
  - Must still include datatype while defining
  - Can be omitted. Then, for loop takes the value in global
  - Can have complex statements

- Condition — Can form complex condition statements

- Alteration ← Can be omitted & use as statements instead.
  Can also have complex statements

IMPORTANT: You can initialize & declare 2 variables in <initialization>

```
for ( int m = 1 , n = 10 ; ... )
        same type
for ( int m = 1 , double n = 10.0 ; ... )
        different type
```

(you can also set a bool variable and switch it to end loop)

e.g. Print

```
_ _ _ _ * _ _ _ _
_ _ _ * * * _ _ _
_ _ * * * * * _ _
_ * * * * * * * _
* * * * * * * * *
```

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    max_col = 1 + (2*(n-1))
    for ( int row = 1; row <= n ; row ++ ) {
        side_space = n - row
        for (int col = 1; col <= max_col; col ++) {
            if ((col <= side_space) || (col > max_col - side_space))
                printf(" ");
            else
                printf("*");
        }
    }
}
```

Functions

1. Standard library functions

→ Built into C programming. The functions are defined in header files.

- printf() is defined in stdio.h
- sqrt() is defined in math.h

2. User-defined functions

- The execution of a C program begins from the main() function. When the compiler encounters functionName(); control of program jumps to typefunctionName();

- Structure of functions:

(2 in 1)

+) Declared & implemented before main (and then called in main)

```
returnType1 FunctionName1 (type1 para1, type2 para2...) {
    // function body
    return variable;
}
returnType2 FunctionName2 (type3 para3, type4 para4...) {
    // function body
    FunctionName1 (para1, para2);
    return variable; }
int main(void) {
    // main body
    FunctionName2 (para3, para4);
    return 0;
}
```

Orders & type

FunctionName1 & 2 cannot switch order

Declare

+) Prototype written before main, but functions are implemented after main

⇒ Function orders doesn't matter if implementation is after main

// Prototype

```
returnType functionName (type1, type2);
```

; in prototype

Can also set parameter names (but unnecessary)

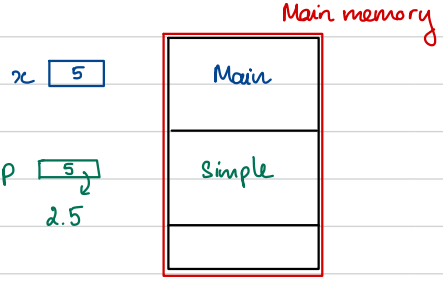
```
int main () {
    // main body
    functionName (para1, para2);
}

returnType functionName (type1 para1, type2 para2)
{
    // function body
    return variable;
}
```

NOTE Variables in functions are within function scope only.

Pointers

Recall: When passing a parameter to a function, only a copy of the value is sent, not the value itself.

e.g.
```
void FunctionName (int P)
{
    P = P / 2
    return P;
}
int main() {
    int x = 5;
    FunctionName(x);
}
```



⇒ x remains 5 since only a copy of x is sent.

⇒ What if I want to change the value of x in function? POINTERS

Note: If we have a variable var, using &var will give you the address of it in memory (while using scanf(), the value entered is stored in the address of var)

Pointers are special variables that stores the address of other variables rather than values

Declaring Pointers

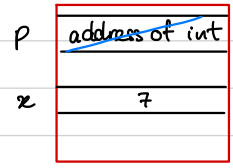
```
int * p;
```

- variable p will contain an address of an int variable.
- DataType of p is (int *) with specifier %p

1) `int * p ;`

2) `int x = 7 ;`

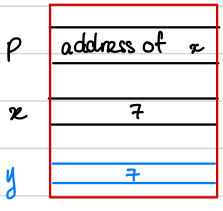
3) `p = & x ;`



Now that p is "pointing" towards the address of x, we can access the int variable (x) that the pointer (p) is pointing to using:

Dereference operator * ⇒ It operates on a pointer and gives the value stored in that pointer

4) `int y ;`
`y = *p`



⇒ Sets y to content of address p (like y = x)

Changing Value Pointed by pointers

- `int* p, c ;`
  `c = 5 ;`
  `p = &c ;`
  `c = 1 ;`
  → `*p = 1`

- `int* p, c ;`
  `c = 5 ;`
  `p = &c ;`
  `*p = 1`
  → `c = 1`

- `int* p, c, d ;`
  `c = 5 ;`
  `d = -15 ;`
  `p = &c ;` → `*p = 5`
  `p = &d ;` → `*p = -15`

This changes the value of c (value/variable pointed) but NOT pointers. Pointer still points to c and still holds &c

Use case of pointers (when we want the function to affect global variable)

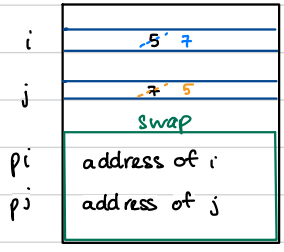
e.g. Write a function that swaps 2 variables in global scope

✗
```
void swap (int i, int j) {
    int temp = i;
    i = j;
    j = temp;
}
```
⇒ i & j swapped in the function, but not in main.

WRONG !!!

②
```
void swap (int* pi, int* pj) {
    int temp = *pi;
    *pi = *pj;
    *pj = temp;
}
int main (void) {
    int i = 5, j = 7;
    swap (&i, &j);
}
```
CORRECT ✓

Why pointers? Because we want real representations of i & j inside functions



Note: In swap() function:

```
void swap (int* pi, int* pj)
```
Declaring that we're passing addresses

... → dereferencing & changing the values of the main memory.

⇒ Pointers feed on addresses (put address to pointers)

Common mistakes

```
int c, *pc;
pc = c;        ✗  // pc is address but c is not
*pc = &c;      ✗  // &c is address but *pc is not
```

But `int c = 5;`

`int *pc = &c;` is not wrong (int* is declaring, not dereferencing)

```
int *pc;
pc = &c;
```

NULL pointer

Declaring by `int* p = NULL;`

⇒ A pointer p (8bytes), pointing to NULL p |88| → NULL
(helpful for linked list later)

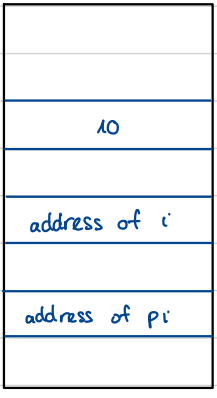
⇒ We can use this to prevent dereferencing of something invalid by using if-else.

```
if (p == NULL) {
    printf("error...")
} else {
    *p = ...
}
```

This dereferencing must be safe to conduct, meaning p != NULL
else, it will give Runtime Error

Double Pointers Pointers of type (**) holding the address of another pointer

e.g.
```
int i;
int* pi;
int** ppi;

i = 10 ;
pi = &i ;
ppi = &pi;
```



Function returning a pointer Say we want to write a function called largerLoc() that returns the actual pointer to the larger integer variable between x & y

```
int main(void) {
    int x, y;
    x=5, y=7;
int i = *largerLoc(&x, &y);
    printf("%d", i)
}
```

we are trying to print the value at the address the pointer is pointing (or, the larger number)

returning integer pointer

```
int* largerLoc(int *x, int *y) {
  if (*x > *y) {
    return x;        [&*x]
  } else {
    return y; }
}
```

(within this function, x & y are pointers)

Variable Scope

local variables defined in function will not exist outside
Variables defined in for-loop will not interfere outside for loop
... while-loop ...
... compound statement ...

Global Variables : Variables that are defined & initialized outside of any function ⇒ Can be used anywhere (even main())

or Constants

→ Automatically initialized to 0

BAD STYLE

Test Goldbach Conjecture

Question: Prompt the user for an even number that is > 2, repeatedly ask until they enter an even number. Print out the 2 prime numbers that add together to that number.

```
#include <stdio.h>
#include <stdbool.h>

int main(void){
int number = getUserInput();
testGoldbach(number);
return 0;
}
```

main body of program. As short as possible. Now create separte functions

①
```
int getUserInput(){
 int number;
 do{
    printf("Enter an even number: ");
    scanf("%d", &number);
    } while (number%2 != 0 || number <= 2);
```
bool isValid();

②
```
void testGoldbach(int number){
    int firstPart = 2, secondPart;
    do{
        secondPart = number - firstPart;
        if (isPrime(secondPart)){
            printf("%d %d", firstPart, secondPart);
            return; }
        else { firstPart = nextPrime(firstPart); }
    } while (firstPart <= secondPart);
```

③

```
bool isPrime (int num) {
    int count = 0;
    for (int i = 1; i <= num; i++) {
        if (num % i == 0) {
            count += 1;
        }
    }
return (count == 2);
}
```

④

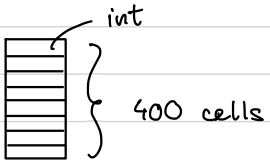
```
int nextPrime (int on) {
    if (on == 2) { return 3; }
    do {
        on += 2;
    } while (!isPrime(on));
return on;
}
```

Arrays

Why arrays ? — Represent multiple data of the same type
Vectors in machine learning

Declare arrays :

type identifier [size] ;



- We have to put in size if we need to reserve memory.

Initialize arrays : `int marks [5] = {100, 90, 80, 70, 40};`

Don't need if we're initializing

If `int marks [6] = {};` ⇒ {0, 0, 0, 0, 0, 0}
`int marks [6] ;` ⇒ Only reserve, no initialize

- We can let the size of arrays as variables, but memory will be on the Heap, not Call Stack (Dynamic Allocation)

- `int marks [6];` ⇒ Can initialize using index
```
for (int i = 0; i < 6; i++) {
    scanf("%d", &marks[i]);
}
```
⇒ from user inputs

Input to each index

NOTE : Arrays are addresses, meaning it can only be pointed at and store values, not pointing to another value (no change addr of s)
⇒ We can do `int s[6]; s[2] = 8` but NOT `s = 70`.

- Initialize using values (putting values had into arrays)

$t_1 = \frac{1}{2}$, $t_2 = \frac{1}{4}$, $t_3 = \frac{1}{8}$, $t_4 = \frac{1}{16}$, ... MAX term

```
double terms[MAX];
for (int i = 0; i < MAX; i++) {
    terms[i] = pow(2, -(i+1));
```

↑ initializing each index

- **Reverse the contents of array**

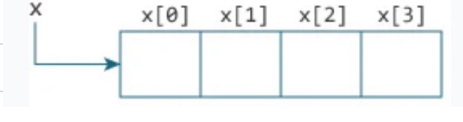
+) Create another array (but not memory efficient)
+) Doing it in-place ⇒ Using swap function

```
for (int low = 0, high = length-1;  low < (≠) high ; low++, high--) {
    swap (&marks[low], &marks[high]);
}
```

Must index to access the □

⇒ Swapping low & tail, then move to the middle

- **Meaning of Array Identifier** marks, grades, etc.



The identifier (x) points to the first element in the array

- $x = \&x[0]$
- $*x = x[0]$

⇒ We can do things like $*x = 7$, but not $x = 7$

Arrays and Functions

Say we want to read from user input (passing array into function)

```
void read (int p[], int size) {
    for (int i = 0; i < size; i++) {
        scanf ("%d", &p[i]);
    }
}
```

&p[i] = *(p+i)

```
int main (void) {
    int size = 5
    int marks [size];
    read (marks, size);
}
```

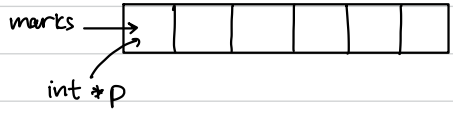
This is NOT passing array, only passing &marks[0] (address of 1st element - identifier)

This is NOT a regular array. It is a parameter

⇒ It is int *p, declaration of a pointer

It makes sense! Here, we are passing the address of the 1st element in main() to a pointer variable created in parameter.

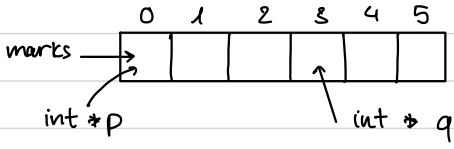
⇒ The pointer variable in the function now points to the same address of the 1st element of the array

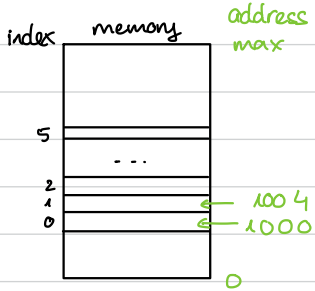


In summary, to pass an array to a function, we pass the pointer to the first element (identifier)

Note: Also need to pass "size" into function, because size is a local variable in main()

Pointer Arithmetic





- marks [0] = *p
- marks [1] = *(p+1)
- marks [2] = *(p+2)

compiler can jump address.

...

Say we do: int * q = & marks [3]

*(q-1) ⇔ marks [2]

And since p & q are just addresses, they can also be arithmetic

e.g. q - p + 1 ⇒ no. of squares from p to q

(actual address result will be scaled down for convenience)

We can do if (p <= q) { ... }

Again, pointer variables just contain addresses. We don't care about the actual addresses, we care about the position & content

EXCEPT we can do if (p != 0) {

NULL

BECAUSE NULL HAS AN ADDRESS OF NULL

Therefore, we can replace the function as :

```
void read (int *marks, int size) {
    for ( int i=0 ; i < size ; i++ ) {
        scanf ( "%d ", &marks[i] );
    }
}
```

&marks[i] → (marks + i)

NOTE Since we are using pointers & pass them into functions, modifying arrays in function will also modify array in main memory (aliasing lists).

- Example: swap cells in array through a function

```
int main (void) {
    int marks [] = {1, 2, 3, 4, 5};
    swap (marks, 1, 2);
}

void swap (int * marks, int i, int j) {
    int temp = marks[i] ;
    marks[i] = marks[j] ;
    marks[j] = temp ;
}
```

marks[i] → * (marks + i)

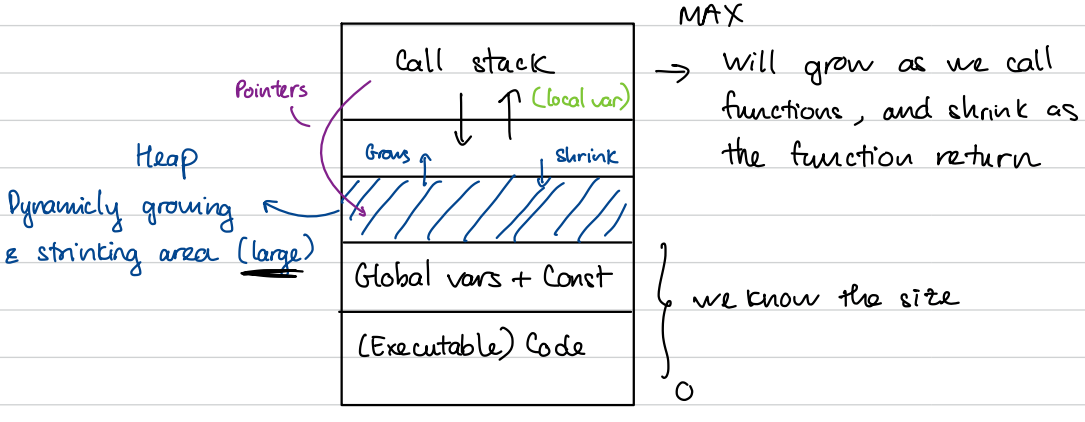
Dynamic Memory Allocation

In most cases, we don't know the size of arrays. So this may lead to excess memory. Recall that we can determine the size of the array as variables:

```c
#include <stdio.h>
int main()
{
    int size;
    scanf("%d",&size);
    int marks[size];
}
```

synthetic sugar, meaning short version

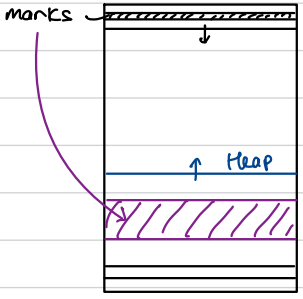
In the main memory, there are:



⇒ This is the way we organize memory

If space is not available, return NULL

Allocate memory to Heap



- We will request a chunk of memory of specific size in the Heap.
- We do that by:
  - malloc (memory allocation): a special C function from stdlib to acquire any size of memory
  - free: a function in stdlib to return memory when no longer needed, so you don't run out of memory.

→ `# include < stdlib.h >`

① Instead of creating an array on call stack, create a pointer.

`int * marks = (int *)malloc ( size * sizeof (int) );`

size of array

// malloc prototype: returns a pointer, type ⇒ void *, that points
// to 1st byte of the allocated memory
// prototype * void malloc (int size),

② Use malloc function to create space on Heap
③ Type cast the return of malloc to make sure it returns (int *)

Using allocated memory on Heap (using pointer arithmetic)

`*(marks + i)`
`marks [i] = 100;`
or `scanf ("%d", &marks [i] )`
or `(marks + i)`

Free the memory

do this after using the memory

`free (marks);`
→ only need the pointer.

2D Arrays

Motivation: Matrix, or keeping tables with rows / columns.

```
int marks [numRows][numCols];
```

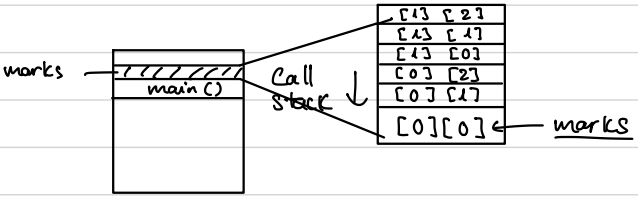
Initializing 1)

```
int marks [2][3] = {
                    { 100, 90, 80 },
                    { 90, 80, 70 }
                    }
```

2)

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        marks[i][j] = 0;
    }
}
```

How the memories are stored in memory



Go up in addresses (row-major order)

Address of [i][j] = Address of [0][0] + sizeof(int) * (i * num cols + j)

⇒ Depend on this when passing 2D Arrays into user-defined functions.

⇒ To compute the address of any cells, we need [0][0] and total number of columns

e.g - Write a function to calculate the sum of all marks in 2D array

mandatory

```
int sum ( int marks [2][3], int numRows , int numCols ) {

    for ( ... ) {
        for ( ... ) {
            sum += marks [i][j] ;
        }
    }
    return sum; }
```

```
int main (void) {
    int marks [2][3] ;
    print ("%d ", sum (marks, 2, 3));
}
```

Problem: In for-loops, we will use something like sum += marks [i][j] , so the compiler must compute the address of each cell ⇒ we need to include the number of columns

However, this requirement is very rare in real world programming, because we have to supply number of column. So instead, how do we do that?

Recall that marks is an address that we can use (although we cannot write onto it)

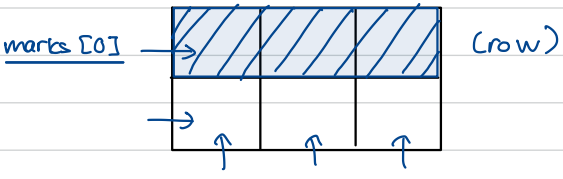
In 1D Array, *marks = ~~*~~ marks[0]

⇒ Which gives the content of the first cell

In 2D Array: marks = & marks[0]

⇒ *marks = ~~*~~ marks[0]

Which gives the address of the whole first row

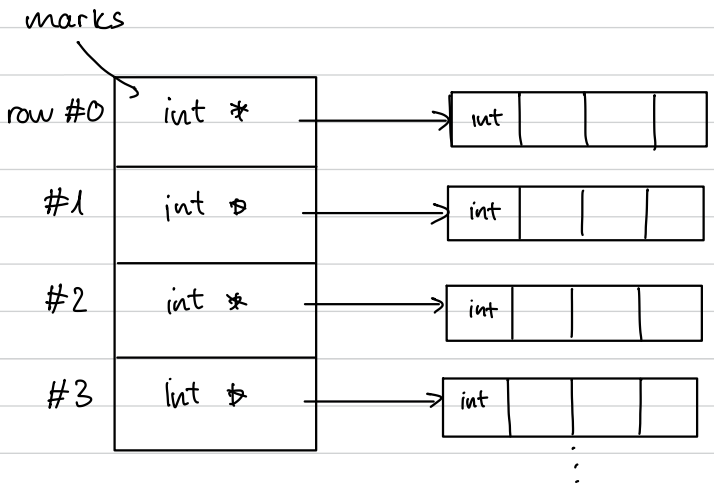


⇒ marks is address, *marks is also an address, marks[i] address

Then marks[i][j] is a value.

⇒ This affects the way we allocate memory of 2D arrays.

Dynamic Memory Allocation for 2D Arrays



⇒ We want these rows to be a pointer variable, pointing to individual rows in our 2D array.

⇒ marks is: `int ** marks`

Allocate memory

```
int ** marks = (int **) malloc ( sizeof (int *) * rows );
for (int i = 0; i < rows; i++) {
    marks [i] = (int*) malloc ( sizeof (int) * cols );
(*(marks + i))
}
```

→ allocate space for each int *

Use memory

We can use `marks [i][j]` as normal. Because:

`marks [i][j] = *(*(marks+i)+j)`

Free memory: We need to first free the arrays pointed at. THEN we can free the pointers.

```
for (int i = 0; i < rows; i++) {
    free (marks[i]);
    }
free (marks);
```

Strings

Strings are character arrays (single quote '')

Null - terminated strings (C)

Used to notify the end point of the string, must have to be considered a string.

Declare a string variable `char s[6];` (like int array)

Initialize a string `char s[] = {'H','e','l','l','o','\0'};`

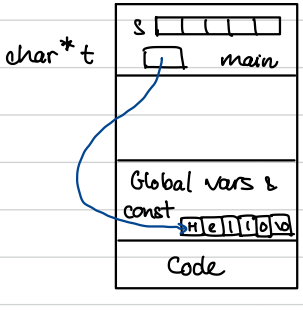
or `char s[] = "Hello";` — can add more than 6 — string variables (read & write)

→ Will initialize 6 cells including Null

Pointers and strings If we initialize the string correctly, it will be considered as a constant. ⇒ They have addresses

⇒ To initialize a pointer to string:

```
char *t = "Hello"
```



⇒ We initialize the pointer variable t with the address of "Hello"

⇒ String literal ⇒ Read - only !

char array: const address, changable char
char*pointers: changable addr, const char

ATTENTION:

CAN

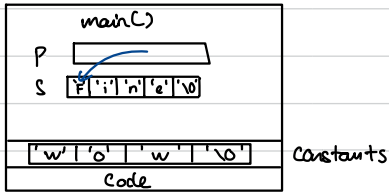
- `t = "HEllo";` ✓ will make t point to another string array
- `s[1] = 'E'` ✓

CANNOT

`t[0] = 'E';` because "Hello" is considered a constant, so you can't change elements
→ Runtime Error ←

BUT If we have t pointing to string defined through arrays

```
char *p = "Wow";
char s[] = "Fine";
p = s;
```
(cannot do `s = p;`)



CAN

`p[0] = 's';` ✓
`s[1] = 'q';` ✓

CANNOT

`s = "Hello";`, since s is an address, not pointer value, so you can't change
→ Compile Time Error ←
(array)

IN SUMMARY

+) If you declare strings as character array directly (through `s[8] = {... }` or `s[] = ""`), square brackets will be part of the call stack — s is an array

+) You can use malloc ⇒ Part of heap.
`char* t = (char*) malloc (sizeof(char) * size);` (size of strings)

+) You can declare as character pointer variable by declaring through pointers (`char *t = "Hello";`) — s is pointer

Pointer Arithmetic and Strings

```
char *p = "Sample";

printf("%s", p);        Sample
printf("%s", p+2);      mple
printf("%c", *p);       S
printf("%c", *(p+2));   m
printf("%c", *p+2);     U
```

Accessing chars in a string

e.g. Count number of spaces

```
int count = 0;
for (int i = 0; s[i] != '\0'; i++) {
    if (s[i] == ' ') {
        count++;
    }
}
```

String into Functions

```
int countBlanks (char *s) {
    :
    return count;
}
```

we don't need size because will stop when meets '\0' (for this particular func)

NOTE: If we use sizeof on any arrays on stack, it returns # of bytes. But when we pass to functions, we're passing as pointers ⇒ will need size

Side note: Making contract that the function cannot modify the string ⇒ Use `const char *s`

(s will points towards a constant string)

`const char *s` / `char const *s`	changable address, constant character
`char * const s`	changable character, constant address
`const char * const s`	→ constant character, constant address

String I/O

Output

`printf ("%s", s);` — print until meets '\0'

- `"%.5s"` → 5 characters
- `"%5s"` → at least 5 char, pass ␣ if more.

`puts(s);` → append a '\n', no need %s

print until meets '\0', replace with '\n'

Input

→ Using scanf:

```
char s[10];
scanf("%s", s);
```

no &

Why not &s ?

- s points to first element of array (= &s[0]) ⇒ use s (address of)

- Scanf will ignore white spaces before characters.
- Scanf will read until a white space (spaces, tabs, ...)
- Scanf will terminate the string array with '\0'

→ Using `gets(s);` → Reads leading white spaces and until an <enter>, it excludes '\n' character. & replace with '\0'

Problem: If we do

```
char s[8];
scanf("%s", s);
```

And type 9 characters, the character will overflow & store on stack

⇒ Unsafe

What is safe? We need to get the size. The safest one is:

```
char * fgets ( char * str, int num , stdin );
```

- `str`: string identifier
- `num`: How many characters to read
- `stdin`: Standard file where input is being saved

⇒ Does not read more, even if more is typed. (reads num - 1 char)

Tips: We can also do something like:

```
char str[10];
fgets ( str, sizeof(str), stdin);
```

(but only when array in stack)

Create our own function that safely reads from the user.

```
int main (void) {
    char s[6];
    printf ("User is entering ... %s", getstringsafely (s, 6));
    return 0;
}

char * getstringsafely (char *s, int size) {
    int i = 0;
    int c;                  // Note that it's int (getchar())
    while (i < size - 1 && (c = getchar()) != '\n') {
        s[i] = c;
        i += 1
    } s[i] = '\0';
      return s;
    }
```

Receiving and sending 1 character

1. Input : `int getchar()`

- returns ascii value.

```
char c;
c = getchar();
```

⇔ `scanf("%c", c);`

2. Output : `int putchar(int char)`

- Has a single int argument
- Writes single character
- Returns the character written cast to an int

Pay attention to indexing (s[i] or s)

String library

#include <string.h>

I. Find length

Prototype : `int strlen ( const char * );`

⇒ Counts character before '\0'

II. Copies a string from one value to another

Prototype : `char * strcpy ( char * dest, const char * src);`

But why use it ? Can't we just do :

```
char s[6] = "Hello";
char d[6] = s;   X
d = s;           X
```

(remember, cannot assign because d is just an address)

⇓

Not safe because dest may not have enough space alloted

```
char s[6] = "Hello";
char d[6];
printf ("Copied string : %s", strcpy(d, s));
```

III Copies only a selected number of characters

```
char * strncpy ( char * dest, const char * src, int n );
```

IV Concatenates 2 strings together

```
char * strcat ( char * dest, const char * src);
```

⇒ Concatenate src to end of dest, overwritting '\0'.
⇒ Unsafe, dest must have enough space.

V. Safe Concatenate

```
char * strncat ( char * dest, const char * src, int n);
```

VI. Compare 2 strings

```
int strcmp ( const char * p, const char * q);
```

Compare like Python, and will returns an int.
+) if (strcmp (a,b)) < 0) ⇒ a precedes b
+) if (strcmp (a,b)) > 0) ⇒ b precedes a
+) if (strcmp (a,b) == 0) ⇒ a & b are the same.

```
int strncmp (const char * p, const char * q, int n)
```

# of char to compare

VII find first appearance of character

```
char * strchar ( char * s , char c ) ;
```

⇒ look for first appearance of c, and return a pointer to this char in s.
⇒ If not found, return NULL

(Do pointer arithmetic to get index)

VIII find string in string

(To find others, do again from ptr1)

```
char * strstr ( char * s1 , char * s2 ) ;
```

(same rule as strchar) returns s1 if s2 has 0 length

IX Converting string to int

```
int atoi (const char * s ) ;
```

(if string only have int)

returns 0 if cannot convert

stdlib

X Converting string to float

```
double atof (const char * s ) ;
```

Function	Action	Library
int strlen()	Returns the length of the string	string.h
char *strcpy()	copies a string from src to dst	string.h
char *strcat()	writes src onto the end of dst	string.h
int strcmp()	compares two strings– then returns an int	string.h
int strncmp	compares two strings with a limit– then returns an int	string.h
char *strstr()	finds the first occurrence of a string in another	string.h
char *strchr()	finds the first occurrence of a character in a string	string.h
int atoi()	interprets the input characters as an integer	stdlib.h
double atof	interprets the input characters as a double	stdlib.h

Jagged Array

There are circumstances that we may need 2D arrays (array of strings), but different # columns each row.

```
J a n u a r y \0
...
M a r c h \0 \0 \0
```

X → We want

```
J a n u a r y \0
...
M a r c h \0
M a y \0
...
```

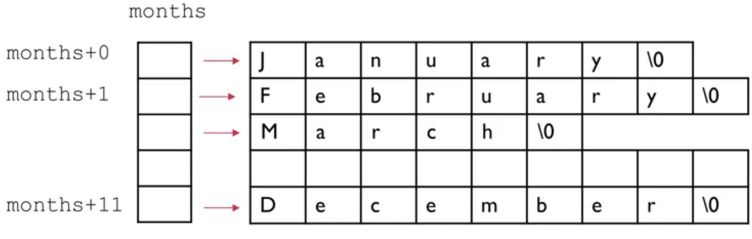
Therefore, we can create an array of pointers, each pointing to a string.

```
char * months [12];
```

→ An array of pointers type char*, can then be initialized:

```
months [0] = "January";
months [1] = "February";
...
```

⇒ Jagged array



* (months + i) [j] ≡ * (* (months + i)) ≡ months [i] [j]

Recursion

Recursion is calling a function within itself, or problem is specified in terms of itself

- A condition must be specified to stop recursion (base case)
- In recursion, all partial solutions are combined to obtain the final solution

e.g.

```
int power (int base, int exp) {
    printf ("power (%d, %d) \n", base, exp);

    if (exp == 0) {
        return 1;
    }                                  ⇒ base case

    return base * power (base, exp - 1);
```

Output:

```
power (2, 3)
power (2, 2)
power (2, 1)
power (2, 0)
→ return 8
```

Algorithm:

- Split the problem into sub-problems (pending solution) all the way to the simple case
- Solve the simple case (base case)
- Walk back to solve the pending sub problems

```
if this is the simple case
    solve it;
else
    redefine the problem using recursion
```

e.g Calculate sum of all nums from 1 to n.

```
int sum(int n) {
    if (n == 1) {
        return 1;
    }
    return (n + sum(n-1));
}
```

Greatest Common Denominator

Finding GCD using The Euclidean Algorithm (% operator)

- If A = 0, GCD (A, B) = B
- If B = 0, GCD (A, B) = A

+) GCD(A, B) = GCD (B, r) where r = A % B

⇒

```
int gcd(int a, int b) {
    if (a == 0) { return b; }
    if (b == 0) { return a; }
    return gcd(b, a % b);
}
```

- **Recursion to do math**

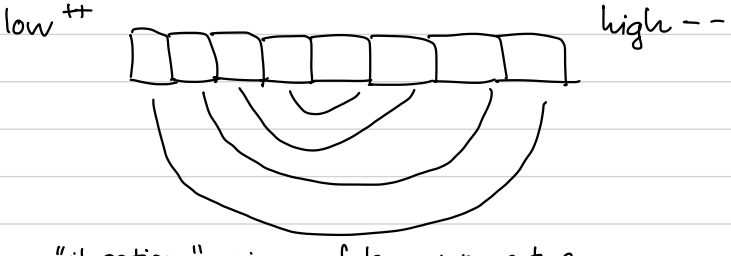
$$f(n) = \begin{cases} 2f(n-1) + 1 & n > 0 \\ 3 & n = 0 \end{cases}$$

```
if (n == 0) {
    return 3;
}
return ( 2 * f(n-1) + 1 );
```

power

Recursion and Strings

Test if a string is mirrored



If any "iteration" gives false, we return