The C Language

The C Language

- Initial version was designed by Dennis Ritchie about 1972
- Imperative (procedural) & structured
- Versions:
  - 1978, K&R C
  - 1989/1990, ANSI C and ISO C
  - 1999, C99
  - 2011, C11
  - 2017, C17

A simple C program

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Required tools

- We need some tools to compile a C source code, these are
  - a preprocessor
  - a compiler
  - an assembler
  - a linker
- In Unix-like OSes(Linux, BSD), ***GCC*** is what we are looking for; it contains
  - a preprocssor (***cpp***)
  - a compiler (***cc***)
  - an assembler (***as***)
  - a linker (***ld***)

Compilation steps and running

- Save the C program code into a file named ***myprog.c***
- Execute the following command in the shell to compile
  - *gcc myprog.c*
- The command above creates a file named ***a.out***; to run it, execute
  - ./a.out

The structure of a source file: header inclusion

- At the beginning of a source file people tend to include the header files in which the functions signatures are declared, e.g.

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

- E.g.: ***printf*** functions is declared in ***stdio.h*** file

The structure of a source file: ***main*** function

- An executable file must include a ***main*** function definitions

```
int main() {
        .
        .
        .
}
```

- The execution of a program begins from the ***main*** function

Variables

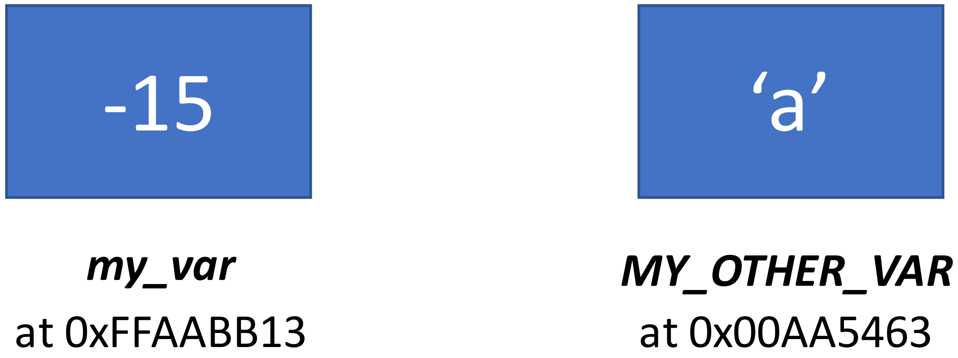
What is a variable?

- A variable is named object defined by us that has a ***value***, a ***data type***, and a ***memory address***.
- E.g.: *mychar* and *num* are variables.

```
char mychar = -15;
int num = 99;
```

A pictorial view of a variable

- You can regard a variable as a box that contains a value and has an address in the memory.



Examples

```
int main() {
    char mychar = -15;
    int num = 99;
    double PI = 3.14;
          .
          .
          .
}
```

Data Types

Data Types

- C is a statically typed language; that is, you have to specify a data type when you declare/define variables and functions (Unlike Python, Like Java).
- Every data type occupies a size of memory, and support only limited range of values; so, before choosing a data type for your variable or functions, take these into consideration.

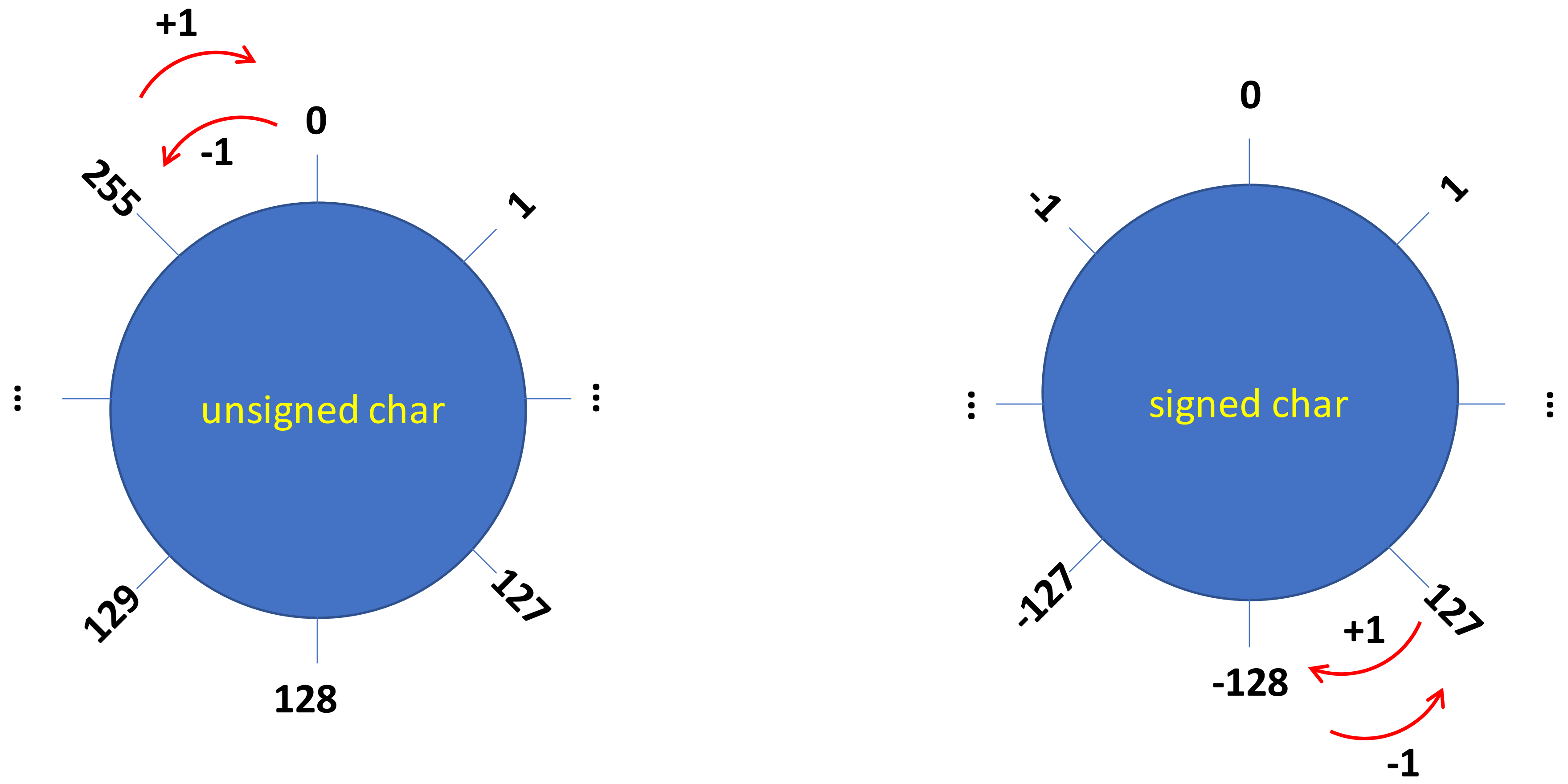
Data Types

Type	Size(in 64bit machine)	Range
char	1 byte	[-128 to 127] or [0 to 255] (*implementation specific)
signed char	1 byte	[-128 to 127]
unsigned char	1 byte	[0 to 255]
short	2 bytes	[-32768 to 32767]
unsigned short	2 bytes	[0 to 65535]
int	4 bytes	[-2,147,483,648 to 2,147,483,647]
unsigned int	4 bytes	[0 to 4,294,967,295]
long	8 bytes	[-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807]
unsigned long	8 bytes	[0 to 18,446,744,073,709,551,615]
float	4 bytes	3.4E +/- 38 (7 digits)
double	8 bytes	1.7E +/- 308 (15 digits)

Range calculation for integers

- If we have ***n*** bit spaces, then the range of an integer type is calculated as follows
  - For signed : $[-2^{n-1}, 2^{n-1} - 1]$
  - For unsigned: $[0, 2^{n} - 1]$
- E.g.
  - For 1 byte ***char*** type
    - For signed: $[-2^{7}, 2^{7} - 1] = [-128, +127]$
    - For unsigned: $[0, 2^{8} - 1] = [0, 255]$
  - For 2 bytes ***short***
    - For signed: $[-2^{15}, 2^{15} - 1] = [-32{,}768, 32{,}767]$
    - For unsigned: $[0, 2^{16} - 1] = [0, 65{,}535]$

Integer overflow: ***char*** e.g.



Operators

lvalue and rvalue

- A language object can be said an **lvalue** if it has an identifiable memory location, such as
  - Variables
  - Functions
  - Dereference of a memory location
- If we can change the value of an **lvalue**, then it is called **modifiable-lvalue**, these are
  - Non const variables
  - Dereference of **writeable** memory location
- A language object can be said an **rvalue** if it has no identifiable memory location, such as
  - Constants(3, 5, -123, ‘A’)
  - Return value of a function

Operators

- There are several operator types in the C Language
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators
  - Others
- These operators can be applied to **lvalues** or **rvalues**

Arithmetic Operators

Operator	Meaning	Usage
+	Addition	A + B
-	Subtraction	-A or A - B
*	Multiplication	A * B
/	Division	A / B
%	Modulus	A % B
++	Increment	++X or X++
--	Decrement	--X or X--

Note: X must only be a **modifiable-lvalue**, but A and B can be **lvalue** or **rvalue**.

Relational Operators

Operator	Meaning	Example
==	Equal to	A == B
!=	Not equal to	A != B
<	Less than	A < B
>	Greater than	A > B
<=	Less than or equal to	A <= B
>=	Greater than or equal to	A >= B

Note: The result of relational operator is either 1 or 0.
Note: A and B can be both **lvalue** or **rvalue**.

Logical Operators

Operator	Meaning	Example
&&	Check both side are non-zero value	A && B
||	Check either/both side are non-zero value	A || B
!	Check the expression is non-zero value	!A

A	B	A && B	A || B	!A
0	0	0	0	1
0	Non-zero	0	1	1
Non-zero	0	0	1	0
Non-zero	Non-zero	1	1	0

Evaluation table

Note: The result of relational operator is either 1 or 0.
Note: A and B can be both **lvalue** or **rvalue**.

Bitwise Operators

Operator	Meaning	Example
&	AND	A & B
|	OR	A | B
^	XOR	A ^ B
~	1's complement	~A
<<	Shift left	A<<B
>>	Shift right	A>>B

Note: A and B can be both **lvalue** or **rvalue**.

Assignment Operators

Operator	Meaning	Example
=	Assign the right-hand side to the left-hand side	A = B
+=	Add then assign	A += B (i.e. A = A + B)
-=	Subtract then assign	A -= B (i.e. A = A - B)
*=	Multiply then assign	A *= B (i.e. A = A * B)
/=	Divide then assign	A /= B (i.e. A = A / B)
%=	Taking modulo then assign	A %= B (i.e. A = A % B)
<<=	Left shift then assign	A <<= B (i.e. A = A << B)
>>=	Right shift then assign	A >>= B (i.e. A = A >> B)
&=	Applying AND then assign	A &= B (i.e. A = A & B)
|=	Applying OR then assign	A |= B (i.e. A = A | B)
^=	Applying XOR then assign	A ^= B (i.e. A = A ^ B)

Note: A must only be a **modifiable-lvalue**, but B can be **lvalue** or **rvalue**.

Flow Control

if, else if and else statements

```
if (TESTEXPR) {
    ...
} else if (TESTEXPR) {
    ...
} else if (TESTEXPR) {
    ...
}          .
           .
           .
           .
else {
    ...
}
```

Expressions(TESTEXPR) are statements that yield a value

- Constants: 3, 5.0, -3.2f, 'a'
- Literals: "HELLO WORLD"
- Arithmetic Expressions: 3 * 5 + 1 / 9.0
- Relational Expressions: x < 5, x == y
- Logical Expressions: x != 3 && y || 5
- Assignment Expressions: x = y + 5
- Function Call: myfunc(), add(3, 5)

If expressions are evaluated

- to a non-zero value, it is regarded **True**
- to 0, it is regarded **False**

if, else if and else statements: examples

```c
if(1) {
    printf("A");
} else {
    printf("B");
}
```

```c
int x = 3;
if(x) {
    printf("E");
} else {
    printf("F");
}
```

```c
int z = 5;
if(z+=-5) {
    printf("I");
} else {
    printf("J");
}
```

if, else if and else statements: examples

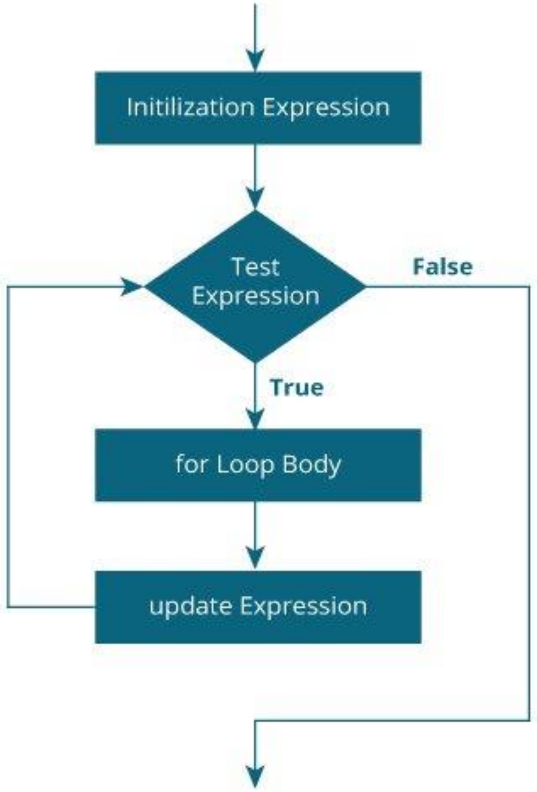
```c
if(0) {
    printf("C");
} else {
    printf("D");
}
```

```c
int y = 10;
if(y > 0) {
    printf("G");
} else {
    printf("H");
}
```

```c
char gender = 'M'; int age = 20;
if(age > 20 && gender == 'M') {
    printf("K");
} else {
    printf("L");
}
```

for loop

```
for(INIT; TESTEXPR; UPDATE) {
                .....
                .....
                .....
                .....
                .....
}
```



https://www.programiz.com/c-programming/c-for-loop

INIT: can be

- A variable definition: int counter = 0, int index = 5
- An assignment to already defined variables: i = 10

TESTEXPR: Any expression that yields a value

- If it yields to True, then loop continues to execute
- If it yields to False, the loop exits

UPDATE: should update the tested control variable

- i++, ++i, --counter, index*=2

for loop: examples

```
for(int i = 0; i < 10; i++) {
    printf("%d ", i);
}

int j;
...
for(j = 20; j > 0; j--) {
     printf("%d ", j * j);
}

for(int val = 1; val <= 32; val *= 2) {
    print("%d ", val);
}
```

for loop: examples

```
const int N_LINES = 10;
for(int i = 1; i <= N_LINES; i++) {
    for(int j = 1; j <= i; j++) {
        printf("* ");
    }
    printf("\n");
}
```

while loop

```
while(TESTEXPR) {
            ...
            ...
            ...
            ...
}
```

while loop: examples

```
int counter = 10;
while(counter > 0) {
    if(counter % 2 == 0) {
        printf("%d\n", counter);
    }
    counter--;
}
```

Loop control statements: break

- break; statement can be used to stop the current iteration and exit from the nearest loop block. It can be used inside both for and while loop blocks.
- E.g.,

```
for(int i = 1; i < 100; i++)
{
    printf("%d\n", i);
    break;
    printf("%d\n", i * i);
}
```

```
for(int i = 1; i < 100; i++) {
    printf("%d\n", i);
    if(i % 50 == 0) {
        break;
    }
    printf("%d\n", i * i);
}
```

Loop control statements: continue

- continue; statement can be used to skip the remaining statements and begins from the next iteration. It can be used inside both for and while loop blocks.
- E.g.,

```
for(int i = 1; i < 100; i++) {
    printf("%d\n", i);
    continue;
    printf("%d\n", i * i);
}
```

```
int i = 1;
while(i < 100) {
    printf("%d\n", i);
    if(i % 2 == 0) {
        continue;
    }
    printf("%d\n", i * i);
    i++;
}
```

Loop control statements: example

```c
for(int i = 0; i < 100; i++) {
    printf("%d\n", i);
}
```

Same

```c
int i = 0;
for(;;) {
    if(!(i < 100)) {
        break;
    }
    printf("%d\n", i);
    i++;
}
```

Simple I/O Operations

Including the header file

- To print something to the screen or read something from the keyboard we can use some of the functions defined in *Standard C Library.*
- The header file that contains I/O functions is **stdio.h**
- To use the functions declared in this header file, we must include it in our source file:

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

Printing

- To print something to the screen we can use ***printf*** function

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    printf("My name is Emre\n");
    printf("%d\n", 15);
    printf("%d\n", 15 * 21);
    printf("PI: %f", 3.14);
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    float PI = 3.14;
    int radius = 5;
    float area = PI * radius * radius;
    printf("Area: %f", area);
    return 0;
}
```

printf

- ***printf*** is a function that is used for formatted printing
- Declaration: `int printf(const char *format, ...)`
- We can pass these values as `*format`
  - Only string: `"This is a string"`
  - Only format specifiers: `"%d", "%f", "%c"`
  - Or both: `"The value is: %d"`
- We can print values to the screen the same number of times with format specifiers that we provide in the `*format` string

printf: format specifiers

- If you want to print a value to the screen you have to specify the format specifier in the format string as a placeholder
- Here are some:
  - %c: for characters
  - %d: for integers
  - %f: for real numbers
  - %s: for strings

Reading

- To read something from the keyboard we could use ***scanf*** function.

```c
#include <stdio.h>

int main() {
    int radius;
    printf("Radius: ");

    scanf("%d", &radius);

    printf("Area: %f", 3.14 * radius * radius);
    return 0;
}
```

scanf

- ***scanf*** is a function that is used for reading
- Declaration: `int scanf(const char *format, ...)`
- You should pass the corresponding format specifier in `*format` for your variables. If you want to read
  - an integer value, use `"%d"`
  - a real value use, `"%f"`
  - a character value, use `"%c"`
  - a string, use `"%s“`

scanf: e.g.

```
int x; scanf("%d", &x);
char y; scanf("%c", &y);
float z; scanf("%f", &z);
```

&: address-of operator

- ***scanf*** reads from keyboard into a variable. To do so, it needs the address of the variable, not its value. To pass the address of a variable into a function like ***scanf***, we use & (ampersand) operator. E.g.
  - &my_var: indicates the address of *my_var*
    - `int my_var; scanf("%d", &my_var);`
- If we know the address in advance, we can also pass it. E.g.
  - `scanf("%d", 0xFFFFCC3C);`

Functions

What is a function?

- A function is a group of statement that performs a certain task.
- Functions in C have memory addresses and introduce local scopes for variables defined in them. A function definition looks like this:

```
<return type> <function name> (<parameter1 type> <parameter1 name>,
                                <parameter2 type> <parameter2 name>,
                                …)
{


}
```

Terminology

Prototype (i.e., Signature)

Body

```
float calc_cylinder_volume(float radius, float height)
{
    float PI = 3.14;
    float volume = PI * radius * radius * height;
    return volume;
}
```

Parameters

This whole part is called “function definition”

E.g.

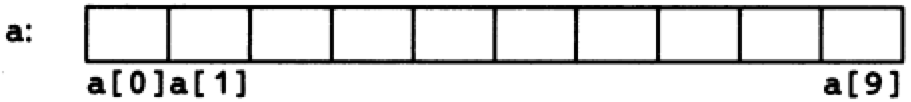
```c
#include <stdio.h>

float calculate_square(float val) {
    return val * val;
}

void calc_and_print_area(float radius) {
    float PI = 3.14;
    float area = PI * calculate_square(radius);
    printf("Area: %f", area);
}

int main() {
    float r;
    scanf("%f", &r);
    calc_and_print_area(r);
    return 0;
}
```

function calls

Arrays

What is an array?

- An array holds a set of variable that share the same data type in a contagious memory region



- E.g.: Here below is an integer array
  - ***my_array***: 95 21 -11 9 10 1005
- In C, the first element resides in 0th index
  - 0th element of my_array is 95
  - 1st element of my_array is 21
  - 2nd element of my_array is -11
  - …

Credits: The C Programming Language, Second Edition – K&R

Defining arrays

- The template for arrays stored in stack area is
  - <data type> <array name>[<number of elements>];
  - E.g.:

```
int my_array[15];
float the_holy_arr[1000];
```

- We can also initialize arrays with predefined values
  - E.g.:

```
int my_array[] = {1, 6, 9, -5, 28};
```

- Note: If you don’t provide initial values for arrays defined in local scope, they contain junk values; that is, the values can be anything in the data type domain

Accessing array elements

- We can access any element of an array by providing an index. E.g.
  - We can access the 15th element of array ***my_array*** by my_array[14]
  - The 1st element by my_array[0]
  - The nth element by my_array[n-1]
- We consider arrays' elements as variables, by doing so we can apply all operators to them as we do on variables

```
my_array[9] = 3;
my_array[2] = my_array[1] + my_array[0];
if (my_array[3] > 15) {...}
int n = 2;
my_array[n+1] = 0;
```

Pointers

What is a pointer?

- A pointer is a special type of variable that hold memory address of values rather than values
- Pointers are the most prominent features of C
- We can create a pointer-variable that points any primitive data type in the language like int, char etc. as well as custom ones that we define by using struct or union keywords.

Defining a pointer variable

- The template for pointer variable is

Pointer

```
<data type>* <variable name>;
```

- E.g.

```
int* my_ptr;
float* my_float_ptr;
```

- You can regard pointer-variables as normal variables whose data type the pointer rather than value. For example, we can say that ***my_ptr*** is a variable of integer-pointer data type.

Assigning values to pointer variables

- We can assign a memory address directly to any pointer-variables
  - E.g.

```
int* my_ptr;
my_ptr = 0xFFFFCC3C;
```

- We can assign a pointer-variable’s value to any pointer-variable
  - E.g., both ***my_ptr*** and ***my_other_pointer*** are of integer pointer type

```
int* my_ptr;
my_ptr = my_other_pointer;
```

Assigning values to pointer variables via & operator

- Recall that by using & operator we can get any variable’ address
  - E.g.

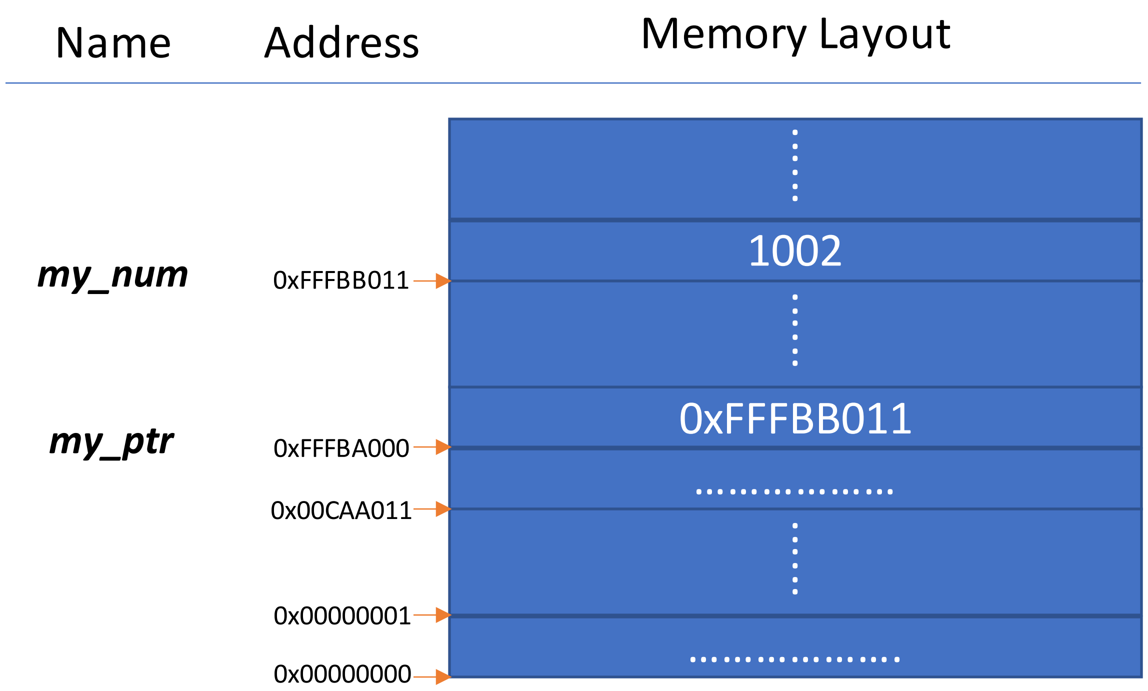
```
int my_number = 3;
int* my_ptr = &my_number;
```

- Here above we assign the address of ***my_number*** variable to ***my_ptr*** pointer-variable

A pictorial view

- Let say we have an integer variable named ***my_num*** and an integer pointer-variable named ***my_ptr***

```
.
int my_num = 1002;
.
.
.
int* my_ptr = &my_num;
.
.
```



Printing addresses

- We can print addresses of variables to the screen with ***printf***. The format specifier we use is “%p”. E.g.

```c
int my_num = 1002;
printf("%p\n", &my_num);
int* my_ptr = &my_num;
printf("%p\n", my_ptr);
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
0x7ffd95cc0f8c
0x7ffd95cc0f8c
```

Size of pointer types

- The size of pointer types depend on the CPU architecture.
  - In 32bit systems, all the pointers are of 4 bytes long
  - In 64bit systems, all the pointers are of 8 bytes long
- E.g.

```c
#include <stdio.h>

int main() {
    printf("%ld\n", sizeof(char*));
    printf("%ld\n", sizeof(int*));
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
8
8
```

Pointer Dereferencing

- We can obtain and manipulate the value at the address that a pointer holds. To do so we use dereferencing operator *
  - We can print the value of a variable through a pointer

```c
int my_num = 1002;
int* my_ptr = &my_num;
printf("The value is %d\n", *my_ptr);
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
The value is 1002
```

  - We can change the value of a variable through a pointer

```c
int my_num = 1002;
int* my_ptr = &my_num;
*my_ptr = *my_ptr * 10;
printf("The value of *my_ptr is %d\n", *my_ptr);
printf("The value of my_num is %d\n", my_num);
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
The value of *my_ptr is 10020
The value of my_num is 10020
```

Pointer Arithmetic

- We can apply increment or decrement operations on pointer-variables as well as adding or subtracting a values.
- The value operand affects the results by the size of the underlying pointer data type. E.g., if ***p*** is an integer pointer that points to address *0x04*, adding the value of *1* to it results in *0x08* if an integer takes 4 bytes.

$$p + a \rightarrow p + a * sizeof(pointed\_data\_type\_of(p))$$

Pointer Arithmetic: e.g.

```c
#include <stdio.h>

int main() {
    int my_var = 3;
    printf("sizeof(int) is %ld\n", sizeof(int));

    int* my_ptr = &my_var;
    printf("my_ptr:          %p\n", my_ptr);
    printf("my_ptr + 1:      %p\n", my_ptr + 1);
    printf("my_ptr - 3:      %p\n", my_ptr - 3);

    int a = 5;
    printf("my_ptr + a(%d):  %p\n", a, my_ptr + a);

    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
sizeof(int) is 4
my_ptr:        0x7ffce2e352e8
my_ptr + 1:    0x7ffce2e352ec
my_ptr - 3:    0x7ffce2e352dc
my_ptr + a(5): 0x7ffce2e352fc
```

Pointer and Array relationship

- The name of an array is indeed a pointer to the first element of it
  - E.g.

```c
#include <stdio.h>

int main() {
    int my_array[] = {15, -5, 36, 25};
    printf("%p\n", my_array);
    printf("%p\n", &my_array);
    printf("%p\n", &my_array[0]);
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
0x7ffdace5bff0
0x7ffdace5bff0
0x7ffdace5bff0
```

Pointer and Array relationship(II)

- You can use the name of array as pointer-variable as shown below
  - E.g.

```c
#include <stdio.h>

int main() {
    int my_array[] = {15, -5, 36, 25};
    printf("%d - %d\n", my_array[0], *my_array);
    printf("%d - %d\n", my_array[1], *(my_array + 1));
    printf("%d - %d\n", my_array[2], *(my_array + 2));
    printf("%d - %d\n", my_array[3], *(my_array + 3));
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
15 - 15
-5 - -5
36 - 36
25 - 25
```

Passing pointers to functions

- C only supports ***pass-by-value*** function calling mechanism, that is, the argument that we pass to functions are ***copied*** into function parameters.
- So, any change applied to these parameters only affects these copies
  - E.g.

```c
#include <stdio.h>
void increase(int val) {
    val = val + 5;
}
int main() {
    int x = 3;
    increase(x);
    printf("%d", x);
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
3
```

Passing pointers to functions(II)

- To make any changes on the actual argument that we pass, we have to pass its address rather than its value, then we apply any change by dereferencing the pointer. This method is called ***pass-by-pointer***
  - E.g.

```c
#include <stdio.h>
void increase(int* val) {
    *val = *val + 5;
}
int main() {
    int x = 3;
    increase(&x);
    printf("%d\n", x);
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
8
```

Passing pointers to functions as arrays

- There is no mechanism in C to pass a whole array to a function, instead we pass the first element's address and the size of the array

```c
#include <stdio.h>

void take_square(int* arr, int size) {
    for(int i = 0; i < size; i++) {
        arr[i]= arr[i] * arr[i]; // or *(arr + i) = *(arr + i) * *(arr + i);
    }
}
int main() {
    int my_arr[] = {1, 2, 3, 4, 5, 6};
    take_square(my_arr, 6); // or take_square(&my_arr[0], 6);
    for(int i = 0; i < 6; i++) {
        printf("%d ", my_arr[i]); // or printf("%d ", *(my_arr + i));
    }
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
1 4 9 16 25 36 emo@ubuntu-dev:~/Desktop/project$
```

Special Pointers(I)

- void*
  - Indicates that the address can point to any kind of data.
  - Does not allow us to apply pointer arithmetic.
- NULL
  - Indicates that pointer points to nowhere.
  - Is defined in locale.h, stddef.h, stdio.h, stdlib.h, string.h, time.h, wchar.h

Special Pointers(II): Examples

```
struct node {
    void* data;
    struct node* next;
};

void set_data(struct node* node_ptr, void*
data) {
    node_ptr->data = data;
}

int main(void) {
    struct node* my_node_ptr = ...
    ...
    int* my_int_ptr = ...;
    ...
    set_data(my_node_ptr, my_int_ptr);
    ...
    float* my_float_ptr = ...;
    ...
    set_data(my_node_ptr, my_float_ptr);
    ...
}
```

```
#include <stdlib.h>
...
...
...
if(ptr != NULL){
    // Do something
}
...
ptr = NULL;
...
```

Custom Data Types: struct

What is a struct?

- struct is a C keyword that allows us to define custom data types. We build our custom data types by using basics such as int, char, etc.
- Here below we introduce ***person*** data type by using struct keyword

```
struct person {
    char gender;
    int age;
};
```

E.g.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    return 0;
}
```

Accessing a struct elements

- We can access and manipulate the elements of a struct by using **.** (dot) operator
  - E.g.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    my_person.age += 5;
    my_person.gender = 'M';
    printf("Gender: %c\n", my_person.gender);
    printf("Age: %d\n", my_person.age);
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
Gender: M
Age: 41
```

Size of a custom data type

- We can check the size of our custom data type by using sizeof(…) operator
  - E.g.

```c
#include <stdio.h>

struct person {
    char gender;
    int age;
};

int main() {
    printf("sizeof(struct person): %ld\n", sizeof(struct person));
    return 0;
}
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
sizeof(struct person): 8
```

Pointers to struct data types

- We can define pointers for the variables whose types are our custom data types
  - E.g.

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    struct person* my_person_pointer;
    my_person_pointer = &my_person;

    return 0;
}
```

Pointers to struct data types(II)

- To access individual elements of the data through a pointer we can employ 2 different methods
  - By using dereferencing operator *
  - By using arrow operator **->**

```c
#include <stdio.h>

struct person  {
    char gender;
    int age;
};

int main() {
    struct person my_person = {'F', 36};
    struct person* my_person_pointer = &my_person;
    my_person_pointer->age = 20; // or (*my_person_pointer).age = 20;
    (*my_person_pointer).gender = 'M'; // or my_person_pointer->gender = 'M';
    return 0;
}
```

Size of a custom data type: disabling padding

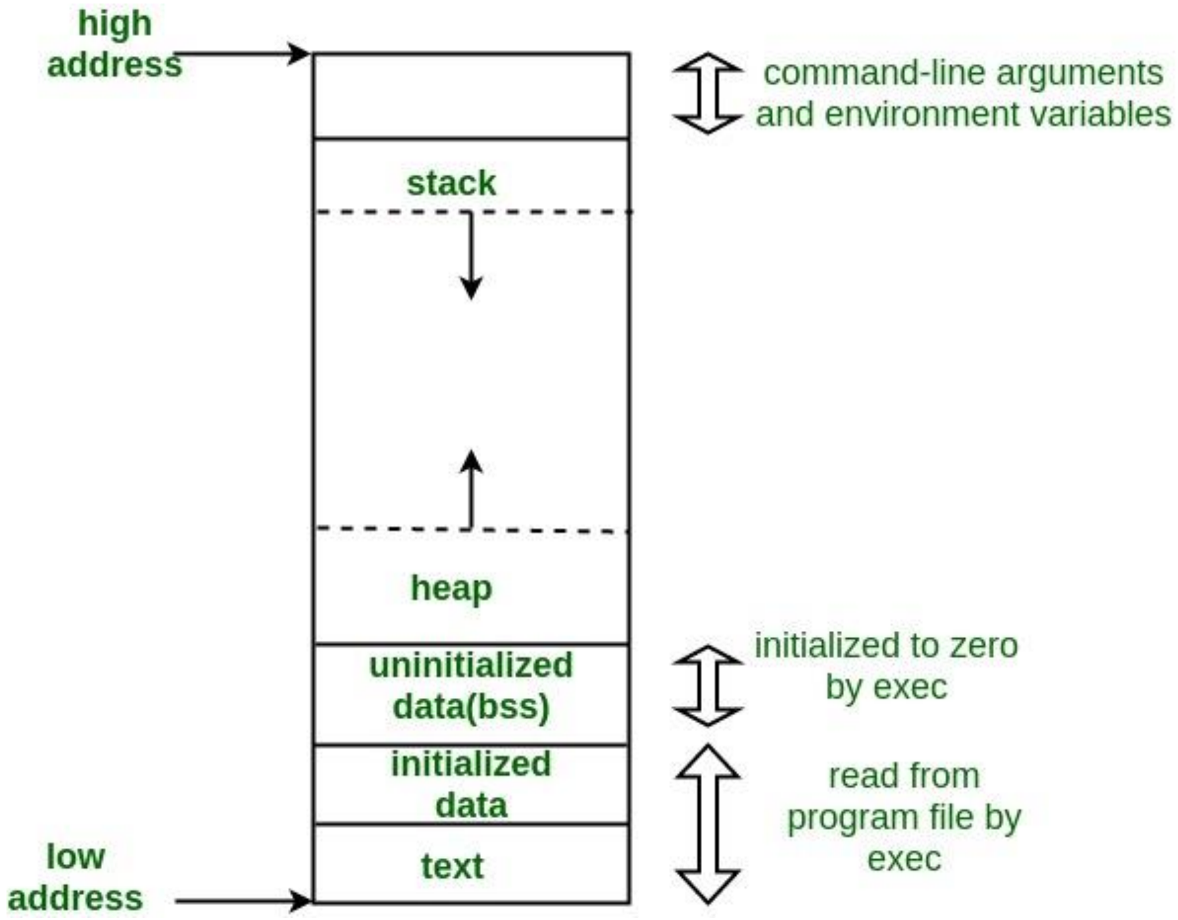
- People generally think that the size of a custom data type would be the accumulation of each individual elements' size; however, compilers are free to add some padding bytes to these custom types in order to access the data fast. To prevent the compiler adding padding bytes we can use GCC compiler directive ***__attribute__((__packed__))***
  - E.g.

```c
struct person  {
    char gender;
    int age;
} __attribute__((__packed__));
```

```
emo@ubuntu-dev:~/Desktop/project$ ./a.out
sizeof(struct person): 5
```

Dynamic Memory Management

Memory Layout



Source: https://www.geeksforgeeks.org/memory-layout-of-c-program/

Motivation

- When we define a variable or array in a function, it is allocated on stack space. And this comes with some limitations:
  - The stack space is limited by default(use 'ulimit –s' command to see in Linux)
  - The data gets destroyed after function is finished
- Solution is to allocate space on **heap** area.

Limited Stack Size

```c
#include <stdio.h>
#define BUF_SIZE (10000 * (2<<10))
int main(void) {
    char buffer[BUF_SIZE];
    for(int i = 0; i < BUF_SIZE; i++) {
        buffer[i] = i;
    }
    return 0;
}
```

```
emo@emo-pc:~/Desktop/lab$ gcc lab.c
emo@emo-pc:~/Desktop/lab$ ./a.out
Segmentation fault (core dumped)
emo@emo-pc:~/Desktop/lab$ ulimit -a
real-time non-blocking time  (microseconds, -R) unlimited
core file size              (blocks, -c) 0
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 63286
max locked memory           (kbytes, -l) 2035160
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 63286
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
```

Stack Deallocation

```c
#include <stdio.h>

int* generate_data(int size) {
    int data[size];
    for(int i = 0; i < size; i++)  {
        data[i] = i;
    }
    return data;
}


int main(void) {
    const int size = 10;
    int* data = generate_data(size);
    for (int i = 0; i < size; i++) {
        printf("%d\n", data[i]);
    }
    return 0;
}
```

```
emo@emo-pc:~/Desktop/lab$ gcc lab.c
lab.c: In function ‘generate_data’:
lab.c:8:12: warning: function returns address of local variable [-Wreturn-local-addr]
    8 |     return data;
      |            ^~~~
emo@emo-pc:~/Desktop/lab$ ./a.out
Segmentation fault (core dumped)
```

Standard Library API for Dynamic Memory Management

- The functions below defined in stdlib.h
  - `void* malloc(size_t total_size_in_bytes);`
  - `void* calloc(size_t n_elems, size_t size_of_an_elem);`
  - `void* realloc(void* ptr, size_t new_total_size_in_bytes);`
  - `void free(void* ptr);`

Allocation

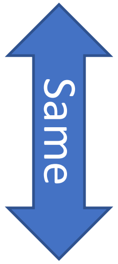
- Allocate size of bytes on heap area.
- `void* malloc(size_t total_size_in_bytes);`
  - Returns: On success, returns the beginning address of allocated space. On failure, returns NULL. The contents are undefined.
- `void* calloc(size_t n_elems, size_t size_of_an_elem);`
  - Returns: On success, returns the beginning address of allocated space, the elements of which are 0 initialized. On failure, returns NULL.
- `void* realloc(void* ptr, size_t new_total_size_in_bytes);`
  - Returns: On success, returns the beginning address of allocated/shrinked/expanded space. If the space gets expanded, the old contents remain the same and new contents are undefined. If the space get shrinked, the remaining contents remain the same. On failure, returns NULL.

Deallocation

- The allocated space remains until we deallocate it manually. To deallocate and inform the OS that we are no longer use that space, we must use **free** function. Otherwise, it still occupies space on the memory and causes memory leakage.
- `void free(void* ptr);`
  - Note: the contents of the pointer variable remains unchanged, so it still points to an address, though the address is unusable. We call this kind of pointers **dangling** pointers. To indicate that a pointer points to nothing, we can set NULL to this pointer variable.

Examples(I)

```c
#include <stdio.h>
#include <stdlib.h>
#define ARRAY_SIZE 100
int main(void) {
    int* my_int_array = (int*)malloc(ARRAY_SIZE * sizeof(int));
    for(int i = 0; i < ARRAY_SIZE; i++) {
        my_int_array[i] = 0;
    }
    ...
    free(my_int_array);
}
```




```c
#include <stdio.h>
#include <stdlib.h>
#define ARRAY_SIZE 100
int main(void) {
    int* my_int_array = (int*)calloc(ARRAY_SIZE, sizeof(int));
    ...
    free(my_int_array);
}
```

Examples(II)

```c
#include <stdio.h>
#include <stdlib.h>
#define INITIAL_SIZE 10
int main(void) {
    int size = INITIAL_SIZE;
    int* my_int_array = (int*)malloc(size * sizeof(int));
    for(int i = 0; i < size; i++) {
        my_int_array[i] = i;
    }
    printf("Addr: %p, Values: ", my_int_array);
    for(int i = 0; i < size; i++) {
        printf("%d ", my_int_array[i]);
    }
    size = 20;
    my_int_array = (int*) realloc(my_int_array, size * sizeof(int));
    printf("\nAddr: %p, Values after realloc(size increasing): ", my_int_array);
    for(int i = 0; i < size; i++) {
        printf("%d ", my_int_array[i]);
    }
    size = 5;
    my_int_array = (int*) realloc(my_int_array, size * sizeof(int));
    printf("\nAddr: %p, Values after realloc(size decreasing): ", my_int_array);
    for(int i = 0; i < size; i++) {
        printf("%d ", my_int_array[i]);
    }
    printf("\n");
    free(my_int_array);
}
```

```
emo@emo-pc:~/Desktop/lab$ gcc lab.c
emo@emo-pc:~/Desktop/lab$ ./a.out
Addr: 0x55c6941a72a0, Values: 0 1 2 3 4 5 6 7 8 9
Addr: 0x55c6941a76e0, Values after realloc(size increasing): 0 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0 0 0 0
Addr: 0x55c6941a76e0, Values after realloc(size decreasing): 0 1 2 3 4
```

The END