

Chapter 3: Pointers and Linked Lists

1. Introduction:

In the first semester, we learned that a program comprises a set of data and a set of instructions, with the data stored in memory as variables. A **variable** is a memory location characterized by an address, name, type, and value.

- **Address:** Each variable stored in memory is identified by an address, a natural number indicating its location. Typically expressed in hexadecimal (e.g., 0x5A63).
- **Name:** An identifier used by programmers to reference the stored value; the variable's name is manipulated instead of the address (e.g., "weight").
- **Type:** In computing, everything is represented in 0s and 1s. The type dictates how to interpret these binary values and specifies the size to be reserved in memory, including the number of bits and allowable operations (e.g., "int" for a 32-bit integer).
- **Value:** The content of the bits composing the variable's value, often changing during program execution (e.g., "15").

When the program encounters a variable declaration statement (e.g., `int age;`), it instructs the operating system (Windows) to allocate a memory space of size `x`, depending on the type. After reservation, the system returns the memory address usable as a variable.

To retrieve a variable's **value**, you simply use its **name**. However, to obtain its **address** (location in memory), the algorithm utilizes the "@" symbol before the variable name, and in C, the "&" symbol precedes the variable name.

Example:

```
write("value of age=", age, " its address=", @age);  
printf("value of age = %d its address = %p", age, &age);
```

Here, `age` is the variable value, and `&age` is its memory address. The `%p` format treats `&age` as a hexadecimal memory address, which can also be displayed in decimal using `%d`. It's important to note that the address may change each time the program is run.

2. Pointers

A **pointer** is a variable whose value points to an address in the computer's memory. This address can be associated with either a variable or a program. Pointers are employed for various purposes, including passing parameters by address, dynamically reserving memory, defining recursive types (such as lists, stacks, and queues), and other applications.

Example:

Memory can be conceptualized as an array numbered from 0 to the memory capacity minus one. In the following illustration, two variables have been allocated. The first is an integer named "age," situated at address 0x0276, holding the value 19. Here, the "0x" denotes that the number is expressed in the hexadecimal system (16) – specifically, 0x0276 corresponds to 630 in the decimal system.

The second variable, denoted as "p," holds the value 0x0276. This value signifies the location of the variable "age." In other words, we can state that "p" points to "age."

Variable Name	Memory address	Content
	0x0000	
	0x0001	
p	0x0002	0x0276
	0x0003	

age	0x0276	19
	0x0277	
	0x0278	

The Creation

To create a pointer variable in the algorithm, we prefix the variable type with the symbol ^. This results in the following format:

```
var p1, p2 : ^type
```

To create a pointer variable in C, we add * before the variable name. Here ^ or * indicates that the variable is of the pointer type, i.e. a memory address, while type is the type of the contents of that location.

```
Type *P1, *P2;
```

Example : We declare six variables x and y of integer type, p1 and p2 of type pointer to integer, z of type real, and pz of type pointer to real.

<pre>int x, *p1, y, *p2; float z, *pz;</pre>	<pre>Var x, y: integer p1, p2: ^integer z : real pz : ^real</pre>
--	---

When declaring a variable, it initially holds an undefined value. It is advisable to set it to NULL in uppercase, signifying that the pointer does not point anywhere (defined within stdio.h, representing the number 0).

```
p1= NULL;
```

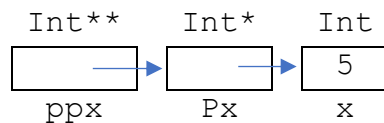
The variable p1 can take the **address** of variable x or the **value** of variable p2, but it cannot take the address of variable z, the address of p2, or the value of pz.

Valid Transactions	Invalid transactions	The Explanation

<code>p1=&x;</code>	<code>p1=x;</code>	p1 is a pointer and x is an integer
<code>p2=p1;</code>	<code>p1=&z;</code>	p1 is an integer pointer and &z is a real address
<code>pz=&z;</code>	<code>pz=p1;</code>	pz is a pointer to a real and p1 is a pointer to an integer
	<code>p2=&p1 ;</code>	P2 is a pointer to an integer, but &p1 is the address of a pointer to an integer.
	<code>p1=&(0x0276) ;</code>	Must be a variable, not a number.

It is crucial to distinguish between the address stored in the pointer and the address of the pointer itself. The pointer, being a variable, possesses an address similar to other variables. Consequently, its address can be assigned to another pointer. However, in such a scenario, the second pointer type must be the address of a pointer of the first type.

For example: x is of integer type (int), and px contains the address of x, so its type is (int*) and ppx contains the address of px, so its type is (int**) as shown in the following diagram:



It is declared as follows:

```
int x, *px, **ppx;
x=5;
px=&x;
ppx=&px;
```

typedef can be used to create new types and the above statement becomes something like this:

```
typedef int* pint;
typedef int** ppint;
pint px;
ppint ppx;
```

Usage:

It's rare that we treat memory addresses as direct numbers, but we treat them as addresses for existing variables. To get the address of a variable, we use the @ operation in the algorithm or & in the C programming language **before** the variable name, and to retrieve the value of the variable (Dereference) from its address stored in a pointer, we use the symbol ^ **after** the variable name in the algorithm and * **before the** name of the variable in the C programming language.

```
p←@x ⇒ p^ ⇔ x
p=& x ⇒ *p ⇔ x
```

Example:

C	The Algorithm	memory	The Explanation
<pre>int x, *p1, y, *p2;</pre>	<pre>Var x, y: integer p1, p2 : ^ integer</pre>		
<pre>x=3; y=4;</pre>	<pre>x←3 y←4</pre>	<p>x <input type="text" value="3"/> <input type="text"/> p1 y <input type="text" value="4"/> <input type="text"/> p2</p>	
<pre>p1=&x; p2=&y;</pre>	<pre>p1←@x p2←@y</pre>	<p>x <input type="text" value="3"/> <input type="text"/> p1 y <input type="text" value="4"/> <input type="text"/> p2</p>	Here p1 contains the address of x and p2 contains the address of y
<pre>*p1=5;</pre>	<pre>p1^←5</pre>	<p>x <input type="text" value="5"/> <input type="text"/> p1 y <input type="text" value="4"/> <input type="text"/> p2</p>	We assign the number 5 to the variable whose address is at p1, and at this point it is the variable x, as if the variable x had a second name, which is *p1 can be replaced by the x=5 statement;
<pre>p1=p2;</pre>	<pre>p1 p2←</pre>	<p>x <input type="text" value="5"/> <input type="text"/> p1 y <input type="text" value="4"/> <input type="text"/> p2</p>	We assign the value of p2, which represents the address of y, to p1, so that y, *p1, and *p2 become the same variable at that time.
<pre>*p1=6;</pre>	<pre>p1^←6</pre>	<p>x <input type="text" value="5"/> <input type="text"/> p1 y <input type="text" value="6"/> <input type="text"/> p2</p>	We assign the digit 6 to the variable whose address is in p1 and at this point it is the variable y can be replaced by the y=6 statement; or *p2=6;

Notes:

- To comprehend pointers better, it is always advisable to visually represent variables, with the pointer depicted as an arrow pointing to the variable carrying its address. Additionally, we symbolize a pointer with a value of NULL, indicating that it does not point to any location.
- A pointer is always of a simple type, whereas the variable whose address it holds can be of a complex type, such as an array or structure.
- Attempting to retrieve the value of an uninitialized pointer or a NULL value will cause the program to terminate.
 - A value (variable address) must be assigned to the pointer before attempting to retrieve the value it points to.
 - Before retrieving the value that the pointer points to, it is crucial to ensure that the pointer is not null.
- Understanding the passing of parameters by address in subroutines becomes possible with these concepts.

Example

C	memory	The Explanation
<pre>void exchange(int *x, int *y) { int t; t=*x; *x=*y; *y=t; } int a=5,b=3; exchange(&a, &b);</pre>	<p>has 5 3 b</p> <p>*x has 3 ← &a x</p> <p>b 5 ← &b y *y</p>	<p>Here x and y are two pointers and when calling the function we assign x the address of variable a i.e. x=&a and y the address of variable b i.e. y=&b and inside the function exchange to obtain the variable whose address x carries we use the operation * where *x at this moment represents the variable a and *y represents the variable b</p>

3. Pointer Operations

Suppose that P and Q are pointers and i is an integer. The following table summarizes the operations that can be performed on pointers:

Algorithm operation	Operation C	Type of 2nd Operator	Type of result	Example	Observation
+	+	Int	Pointer	P + i	Returns a pointer to the i th element after P in an array
	++		Pointer	P++	Returns a pointer to the next immediately P element in an array
-	-	Int	Pointer	P - i	Returns a pointer to the i th element before P in an array
	--		Pointer	P--	Returns a pointer to the element immediately preceding P in an array
-	-	Pointer of the same type	Int	P - Q	Returns the number of items between P and Q where P and Q should point to the same array
=	==	Pointer	Boolean	P == Q	This is true if P and Q have the same address, i.e. they point to the same place
≠	!=	Pointer	Boolean	P != Q	This is true if P and Q are different
^	*		Value Type	*P	To retrieve the value whose address it contains

4. Dynamic Memory Management

The method we've known for reserving variables in memory so far is called static reservation. In static reservation, the variable is declared at the beginning of the program, and the compiler automatically reserves the necessary memory. The variable persists until the end of the program's execution (or until the end of a subroutine in the case of a local variable). However, there are situations where we need to allocate a dynamic amount of memory, such as an array with N elements, and N is only known at runtime. In such cases, we declare a pointer, and when N becomes available, we dynamically reserve the array.

Developers have a set of functions that enable dynamic memory management during runtime.

In algorithm:

There are three procedures for dynamic memory management:

1. **allocate()**: Used to reserve an array, taking the pointer's name (array name) and the number of elements as parameters.

```
allocate(array_name, num_elements)
```

Example:

```
allocate(T, 10)
```

2. **realloc()**: Changes the size of the array, either by increasing or decreasing. It takes the pointer's name (array name) and the new number of elements (new size) as parameters. It preserves the values of the previously reserved elements and removes excess or adds new elements to the array.

```
realloc(array_name, new_size)
```

Example:

```
realloc(T, 15)
```

3. **dealloc()**: Deletes the reserved array created with allocate(). It takes the pointer's name (array name) as a parameter.

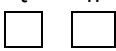

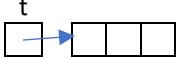
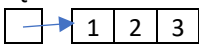
```
dealloc(array_name)
```

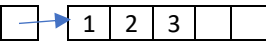
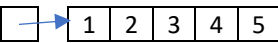
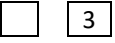
Example:

```
dealloc(T)
```

After creating an array "t" using allocate(), its elements can be accessed either by square brackets [] or by the retrieval operation ^, where the pointer "t" contains the address of the first element, i.e., $@t[0] = t$ and $t^{\wedge} = t[0]$. To get the address of the second element, "t[1]," add 1 to "t," i.e., $@t[1] = t + 1$, and $(t + 1)^{\wedge} = t[1]$. Therefore, the address of "t[i]" is $t + i$, i.e., $@t[i] = (t + i)$ and $(t + i)^{\wedge} = t[i]$.

Example:

algorithm	memory	The Explanation
var t : ^real n:integer	t n 	A pointer "t" and a variable "n" representing the number of its elements are declared
begin write("enter number of elements") read(n)	t n 	Let "n" take 3
allocate (t,n)	t 	allocate() reserves an array of three elements and sets its address to t
t[0]← 1 t[1] ← 2 t[2]←3 t^ ← 1 (t+1)^ ← 2 (t+2)^ ← 3	t 	We fill in the table where we can use the square brackets [] or use ^ where $t[i] \Leftrightarrow (t+i)^{\wedge}$

reallocate(t,n+2)	t 	Calling reallocate() resizes the array to 5
t[3←] 4 t[4] ←5 (t+3)^ 4 (t+4)^ ←5 ←	t 	We fill in the two added elements
deallocate(t)	t n 	We call deallocate() to remove the array

In C:

Memory management in C differs slightly from algorithms. Before delving further into it, we need to familiarize ourselves with "sizeof" and type casting.

4.1.The "sizeof" operation

A variable occupies more or less memory space depending on its type. For instance, a variable of type char takes up one byte, while a variable of type int requires either two or four bytes, depending on the C version. To determine the size required for a specific type, we use sizeof(), which takes the name of the variable or the name of the type as an argument and returns the number of bytes it needs in memory.

```
int sizeof type;
```

Example:

```
float t[20];
printf("char: %d bytes\n", sizeof(char));
printf("int : %d bytes\n", sizeof(int));
printf("double: %d bytes\n", sizeof(double));
printf("the size of t: %d bytes\n", sizeof(t));
printf("the size of t:%d bytes\n", 20*sizeof(float));
```

that displays on the screen

```
char: 1 byte
int: 4 bytes
Double: 8 bytes
T size: 80 bytes
T size: 80 bytes
```

The size of an array can be found by multiplying the size of a single element by the number of elements.

4.2.Type Change: Casting

Sometimes, we need to convert a specific value from one type to another. To force the compiler to change the type of a specific value, we use the following formula:

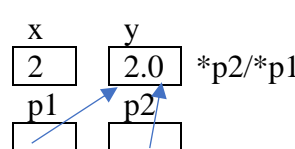
```
(type) expression
```

Where the expression is converted to type

Example 1

<code>int A=8,B=3;</code>	
<code>float R=A/B;</code>	Since operators A and B are integers, the / operation performs integer division, resulting in $R = 8/3$.
<code>printf("no casting R=%f \n",R);</code>	no casting $R=2.000000$
<code>R=(float)A/B;</code>	We convert the value of A (not the variable A) to a real number, and then we perform the division process, so the operation becomes $R = 8.0/3$.
<code>printf("with casting R=%f \n",R);</code>	with casting $R=2.6666666$

Example 2

<code>int x,*p1;</code>	An integer and a pointer to an integer
<code>float y=2,*p2;</code>	A real number and a pointer to a real number
<code>x=(int)y;</code>	It converts the value of y to an integer and puts it in x, so x takes the value 2
<code>p2=&y;</code>	p2 takes the address of y
<code>p1=(int*)p2;</code>	Converting the address of a float to the address of an int, but the address of the variable remains in both variables, which is the address of y 
<code>printf("x=%d \n",x);</code>	Displays $x=2$
<code>printf("*p2=%f \n",*p2);</code>	Displays $*p2=2.000000$ the same as y
<code>printf("*p1=%d \n",*p1);</code>	Displays $*p1=1073741824$ Because translating the bits of a real number into an integer does not give the same number

4.3.Memory Management in C

Dynamic memory management in C is done using four functions defined in the `stdlib` library:

- `malloc()` (memory allocation, meaning to reserve memory): It instructs the operating system to reserve the required amount of memory.

```
void * malloc(int size);
```

It takes the required memory size (number of bytes) as a parameter and returns a pointer to the reserved memory. If the process fails due to insufficient available size, it returns `NULL`.

Example:

```
float *t;
```

```
t=(float *)malloc(10*sizeof(float));
```


t=	(float *)	malloc(10*	sizeof(float));
Table Name	Convert to Pointer Type	To reserve the table	Number of items	The size of each element	Type of each element	

- `free()`: This function is used to return memory previously reserved by the operating system's `malloc()`, allowing it to be used by other programs.

```
void free( void * pointer );
```

It takes a previously reserved pointer as a parameter. It is recommended to set the pointer to NULL after calling `free()` to ensure that the pointer is no longer pointing to valid memory and to avoid potential errors.

Example:

```
free(t);
```

- `realloc()`: This function is used to change the size of the reserved memory, either by increasing or decreasing it.

```
void * realloc(void * pointer, int new_size);
```

Where the function calls `malloc()` to reserve a new block of memory with the size of `new_size`, then copies all the values from the "pointer" array to the new location (or deletes the extra elements if `new_size` is smaller than the old size). After that, it deletes the old reserved array by calling `free()`. If the operation succeeds, it returns a pointer to the new location; otherwise, it returns NULL.

Example:

```
t=(float*)realloc(t, 20*sizeof(float));
```

- `calloc()`: Similar to `malloc()`, but this function puts zeros in the reserved memory.

```
void * calloc(int nb_element, int element_size);
```

It takes `nb_element`, representing the number of items in the array, and `element_size`, representing the size of each element. It returns a pointer to the allocated memory with zero-initialized values.

Example:

```
t=(float*)calloc(10, sizeof(float));
```

Observation:

- In the function lesson, we learned that `void` means the function returns nothing, while `void*` means the function returns a pointer of an undefined type.
- The `void*` type needs to be converted to the specific pointer type that will hold the address. This is done by placing the pointer type in parentheses before the `malloc`, `calloc`, and `realloc` function names. However, this conversion is not necessary in C++.
- To use these functions, you need to include the `stdlib` or `alloc` library by using the following statement:

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

```
#include <alloc.h>
```

The `sizeof` operation is not a function, so parentheses can be omitted.

When reserving memory, we follow these steps:

1. Reserve memory with `malloc`.

2. Ensure that the allocation process has completed successfully by using `if (pointer != NULL)`.
3. When finished using the allocated memory, return it to the system using `free`.

Example

C	The Explanation
<code>#include <stdio.h> #include <stdlib.h></code>	Inclusion of the STDLIB library
<code>int main(void) { char *str;</code>	Declaring a char pointer
<code>str = (char *) malloc(4*sizeof char);</code>	allocating an array for 4 characters:
<code>str[0]='A'; str[1]='S'; str[2]='D'; str[3]='\0';</code>	Populating the array with the string "ASD" using [] and the symbol '\0' to indicate the end of the string:
<code>*str='A'; *(str+1)='S'; *(str+2)='D'; *(str+3)='\0';</code>	Populating the array with the literal string "ASD" using the retrieval operation * where *(str+i) ⇔ str[i]
<code>printf("String is %s\n Address is %p\n", str, str);</code>	To display the string and its address, noting that & is not used because str is already an address:
<code>str = (char*) realloc(str, 5*sizeof char);</code>	Changing the capacity of the array from 4 to 5:
<code>str[3]='2'; str[4]='\0'; *(str+3)='2'; *(str+4)='\0';</code>	Filling in the last two characters so that the string becomes "ASD2":
<code>printf("String is %s\n New address is %p\n", str, str);</code>	Displaying the string "ASD2" and its new address:
<code>free(str); return 0; }</code>	Returning reserved memory:

4.4. Pointers and matrices in C

Matrices in C are arrays in which each element is an array. We want to create an M[3][4] matrix with three rows and four columns. Suppose we have three arrays: M0, M1, and M2.

```
float M0[4], M1[4], M2[4] ;
```


These arrays can be created using pointers

```
float *M0, *M1, *M2;  
M0=(float *)malloc(4*sizeof(float));  
M1=(float *)malloc(4*sizeof(float));  
M2=(float *)malloc(4*sizeof(float));
```

Note that M0, M1 and M2 are all of the same type (float *), so they can be replaced by an array M of type (float *).

```
float * M[3];  
for(int i=0; i<3; i++)  
    M[i]=(float *)malloc(4*sizeof(float));
```

Now, pointers can be used to create the array ‘M’

C	memory	The Explanation
<code>float **M;</code>	M 	An M pointer is declared to be of type float **

<code>M=(float**) malloc(3*sizeof(float*));</code>		
		An array M is created, which contains 3 elements representing the number of rows. The type of each element is float*.
<pre>for(int i=0; i<3; i++) M[i]=(float*) malloc(4*sizeof(float));</pre>		
		We create three arrays, each representing a row in the matrix. The number of columns is 4, and the type of each column is float. *(M+i) can be used instead of M[i].

Any element of the matrix can be accessed using square brackets `[]` or by using the dereference operator `*` where:

$M[i][j] \Leftrightarrow *(M[i]+j)$

$M[i][j] \Leftrightarrow *((M+i)+j)$

using typedef

```
typedef float ** matrix;
```

```
typedef float * table;
```

```
matrix M;
```

```
M=(matrix)malloc(3* sizeof(table));
```

```
for(int i=0; i<3; i++)
```

```
  M[i]=(table) malloc(4*sizeof(float));
```

Note : A static array in C is a constant memory address that cannot be changed.

Example:

```
int *p,t[10];
```

```
p=t;
```

```
t=p;
```

Correct because t is the address of the first element
Not accepted because t is a constant that cannot be changed.