Status of This Document: WIP(Work In Progress)

DATA STRUCTURE

Dr. Hasan Mahmood Aminul Islam
Md. Mahamudur Rahman Maharaz,

Contact Person:

Md. Mahamudur Rahman Maharaz,
Student, East-West University
Email: 2022-3-60-182@std.ewubd.edu

Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory

Dr. Hasan Mahmood Aminul Islam,

Assistant Professor, Dept of CSE
East West University, Bangladesh
Specialist System-on-Chip Software (5G) Nokia
Nokia Headquarters, Espoo, Finland
D.Sc in Technology (CSE)
Aalto University, Finland
M.Sc in Computer Science
University of Helsinki, Finland
B.Sc in CSE, BUET, Bangladesh
Web Page: https://www.ewubd.edu/faculty-profile/mahmood.aminul
Personal Website: https://hmaislam.github.io/myweb/

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Table of Contents

@copyright: EWU CSE, Dhaka, Bangladesh

Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory

Variables

A variable in a program has a name and can vary in value. The compiler and linker handle this by assigning a specific block of memory within the computer to hold the value of that variable.

The size of that block depends on the range over which the variable is allowed to vary. For example, on a 32-bit PC, the size of an integer variable is 4 bytes. On older 16-bit PCs, integers were 2 bytes. In C, the size of a variable type, such as an integer, does not need to be the same on all types of machines. We have integers, long integers, and short integers, which you can read up on in any basic text on C. This document assumes a 32-bit system with 4-byte integers.

When we declare a variable, we inform the compiler of two things: the variable's name and the variable's type.

- ☐ Variables in C are used to access a single location of memory, and the memory size depends on the size of the data types.
- ☐ Therefore, a variable MUST be declared before being used. When you declare a variable, the compiler automatically allocates memory for it.

Example:

```
data_type variable_name;
```

Everything that you do or imagine is Memory

Memory is a collection of Registers

Efficient use of data types/variables will help you to be smart, skilled and competent programmer

When Loops/Iterations of Some Pieces of Tasks arise, no worries!

Whether we start or are heading to work on something new in programming, remember to "Try" to understand the semantics of the new thing, how to use it in your program, and what inherent things you need to capture in your memory.

Remember that nothing needs to be memorized. If you need to memorize something, you have some logic gaps. If you cannot find your gaps, consult your friends and teachers.

@copyright: EWU CSE, Dhaka, Bangladesh

Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory

Look at the loops: See if you find any difference in logic.

Syntax:

Same purpose

```
init ;
while( cond )
{
   Body;
   Increment/decrement;
}
```

```
init ;
do{
      Body;
      increment/decrement;
} while( cond ) ;
```

```
for( init  ; cond ; increment/decrement)
{
            Body
}
```

```
i=0 ;
while( i<10 )
{
      a = a+5; //body
      i++ ;    //increment
}
```

```
i=0 ;
do
{
      a = a+5; //body
      i++ ;
//Increment
} while( i<10 )
```

```
for( i=0 ; i<10 ; i++)
{
      a = a+5; //body
}
```

Where can we use loops?

- ☐ E.g. **H.S.C / S.S.C Mathematics** : সমান্তর ধারা, গুণোত্তর ধারা
- ☐ **Some logic:**
- ☐ **Condition** false; if the value is zero;
  true ; otherwise ;
- ☐ **for/while** → iteration number: zero or more times

Look: this is the only difference

@copyright: EWU CSE, Dhaka, Bangladesh

- **do-while** → iteration number: once or more

- **Variables vs. Arrays vs Pointers:**

  - **Variables**

    A variable in a program is something with a name, the value of which can vary. The compiler and linker handle this by assigning a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32-bit PC's, the size of an integer variable is 4 bytes. On older 16-bit PCs, integers were 2 bytes. In C, the size of a variable type, such as an integer, does not need to be the same on all kinds of machines. We have integers, long integers, and short integers, which you can read up on in any introductory text on C. This document assumes a 32-bit system with 4-byte integers. When we declare a variable, we inform the compiler of two things: the variable's name and the variable's type.

  - Variables in C are used to access a single location of memory, and the memory size depends on the size of data types. Therefore, a variable MUST be declared before using it. When you declare a variable, the compiler automatically allocates memory for that variable.

  - One frequent problem in C programming is handling similar types of data. For example, if the user wants to store the marks of 10,000 students, this can be done by creating 10,000 variables individually, but this could be more practical. These types of problems can be handled in C programming using arrays.

  - Arrays can be considered as the collection of variables where the base address of the array is the first element of the array. An array in C Programming can be defined as a number of memory locations, each storing the same data type and referenced using the same variable name.

@copyright: EWU CSE, Dhaka, Bangladesh

- Arrays and pointers are synonymous in how they are used to access memory. But, the vital difference between them is that a pointer variable can take different addresses as values, whereas it is fixed in the case of an array. In C, the name of the array always points to the first element of an array.
- An array in C Programming can be defined as a number of memory locations, each storing the same data type and referenced using the same variable name.

An important thing to remember in C arrays

Suppose you declared the array of 10 students. For example, students[10]. You can use array members from student[0] to student[9]. But what if you want to use element student[10], student[100], etc? In this case, the compiler may not show an error using these elements but may cause a fatal error during program execution.

Multidimensional Arrays

Just consider **each box** is the storage of **a simple variable**. Now you can map the logic of a variable and an element of Array in Programming.

array	Col - 0	Col -1	Col -2	Col -3	Col -4	Col -5
Row- 0	array[0][0]	array[0][1]	array[0][2]	array[0][3]	array[0][4]	array[0][5]
Row- 1		array[1][1]		array[1][3]		
Row- 2						array[2][5]

When we declare a variable, we inform the compiler of two things: the variable's name and the variable's type. For example, we declare a variable of type integer with the name k by writing:

```
int k;
```

When it sees the "int" part of this statement, the compiler sets aside 4 bytes of memory to hold the integer's value. It also sets up a symbol table. That table adds the symbol k and the relative address in memory where those 4 bytes were set aside.

@copyright: EWU CSE, Dhaka, Bangladesh

k=2

We expect that when this statement is executed at run time, the value 2 (two) will be placed in that memory location reserved for storing the value of k. In C, we refer to a variable such as an integer k as an "object."

We define a pointer variable in C by preceding its name with an asterisk. We also give our pointer a type, which refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

ptr is the name of our variable, and the '*' informs the compiler that we want a pointer variable, i.e., to set aside however many bytes are required to store an address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not be evaluated to zero since no value is assigned. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, guarantees that the pointer has become a null pointer.

Suppose we now want to store the address of our integer variable k in ptr. To do this, we use the unary & operator and write:

```
ptr = &k;
```

The "dereferencing operator" is the asterisk, and it is used as follows:

```
*ptr = 7;
```

Arrays and Pointers

Arrays and pointers are closely related in C. An array declared

as `int A[10];`

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a const int*. Since A is a constant pointer, A = NULL would be an illegal statement. Arrays and pointers are synonymous in how they are used to access memory.

@copyright: EWU CSE, Dhaka, Bangladesh

But, the vital difference between them is that a pointer variable can take different addresses as values, whereas it is fixed in the case of an array.

☐ Consider the following array:

Declaration: **int std[5];**

In C, **the name of the array** always points to the **first element of an array**. Here, the address of the first element of an array is **std[0]**. Also, **std** represents the address of the pointer where it is pointing. Hence, **&std[0]** is equivalent to **std.** Note the value inside the address **&std[0]** and address age are equal. Value in address **&std[0]** is **std[0],** and value in address std is ***std.** Hence, **std[0]** is equivalent to ***std.**

C arrays can be of any type. We define an array of ints, chars, doubles, etc. We can also define a variety of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

```
char* array[n];
```

each cell in the array A[i] is a char*, so it can point to a character. If you want to assign a string to each A[i], you can do something like this.

```
array[i] = malloc(length_of_string + 1);
```

Again, this only allocates memory for a string, and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words), you need to allocate memory for n char*'s and then allocate just the right amount for each string.

Pointer	Array

@copyright: EWU CSE, Dhaka, Bangladesh

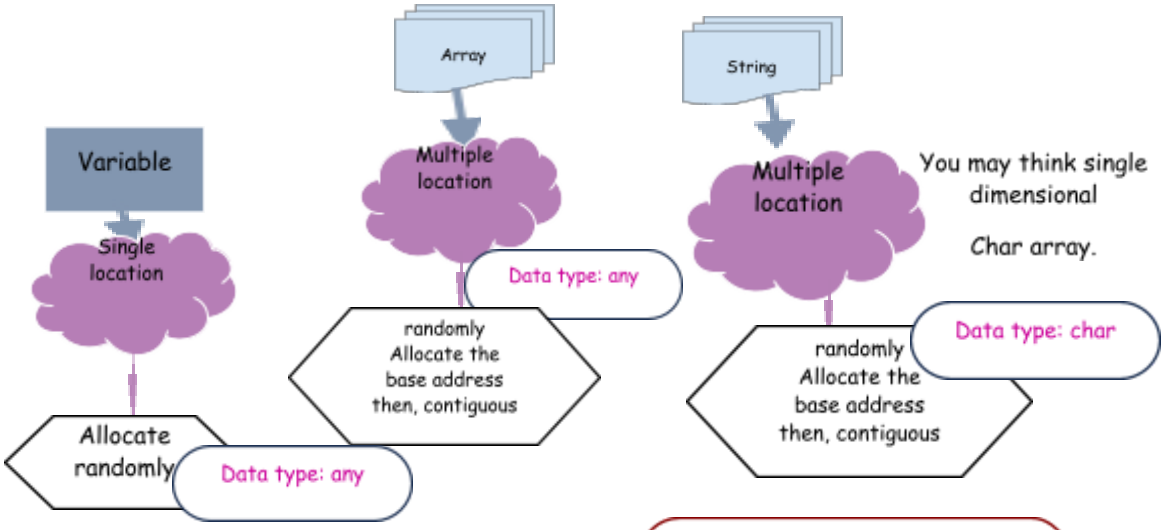
Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory

A pointer is a place in memory that keeps the address of another place inside	An array is a single, pre-allocated chunk of contiguous elements (all of the same type) fixed in size and location
A pointer is a place in memory that keeps the address of another place inside	Expression a[4] refers to the 5th element of the array a.
The pointer can't be initialized at the definition	Arrays can be initialized at definition. Example `int num[] = { 2, 4, 5}`
The pointer is dynamic. The memory allocation can be resized or freed later.	They are static. Once memory is allocated, it cannot be resized or freed dynamically.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ (MMAP) Variables vs. Arrays vs Pointers:



Type: unsigned/signed number

size → Depends on the Generation of Comp

size →

Unsigned data type:

uint8_t
uint16_t
uint32_t

8/16/32 bits

Mapping:

A single location:  e.g., `int a;`

Size: `sizeof(int)` Depending on the Computer Architecture

Multiple addresses (Contiguous), but in the case of Array, the first address is randomly allocated, and the rest of the addresses will be contiguous, and the memory size will be based on your declaration for the array. Again, note that the first address, index 0 (zero), will be allocated randomly by the compiler (the base address), and the rest of the address will follow the base address contiguously.

@copyright: EWU CSE, Dhaka, Bangladesh

Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory

Base Address: **a[0]**

Address of **a[1]** = address of **base a[0]** + sizeof(data type of array)

Figure: Each box represents one memory address whose size is defined by the array's data type. The figure has been drawn to indicate that an array is a collection of variables.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Starting to think like a C programmer

Now that we have studied and tried to understand the core of C programming, we are talking about the C language. You may have been trying to think like a Java programmer and convert that thought to C. Now, it is time to think like a C programmer. Thinking directly in C will make you a better C programmer.

☐ **Here are 15 things to remember when you start a C program from scratch.**

1. Include <stdio.h> and all other related headers in all your programs.
2. Declare functions and variables before using them
3. It is better to increment and decrement with ++ and -- operators.
4. Better to use x += 5 instead of x = x +5
5. A string is an array of characters ending with a '\0". Don't ever forget the null character.
6. An array of size n has indices from 0 to n-1. Although C will allow you to access A[n] it is hazardous
7. A character can be represented by an integer (ASCII value) and can be used as such.
8. The unary operator & produces an address
9. The unary operator * dereferences a pointer
10. Arguments to functions are always passed by value. But the argument can be addressed with just a value.
11. For efficiency, pointers can be passed to or return from a function
12. Logical false is zero, and anything else is true
13. You can do things like for(;;) or while(i++) for program efficiency and understanding
14. Use /* .. */ instead of //, it makes the code look better and readable
15. The last and most important one is always compiling your program before submitting it or showing it to someone. Don't assume that your code is compilable and contains no errors.
16. Try using -std=c99, which is the c99 standard. It's better. (Although c11 is also on its way, it is not a standard at the moment for all machines)

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Advanced Feature of C programming

Table Of Contents

- ☐ Macro
- ☐ Inline Function
- ☐ Callback Function
- ☐ Function Pointer
- ☐ Bit field in c

Reference: https://gcc.gnu.org/onlinedocs/gcc/Inline.html

@copyright: EWU CSE, Dhaka, Bangladesh

Macros

In c programming, we can use macros to repeatedly use a single value or a piece of code in our programs. By defining macros once in our program, we can use them frequently throughout the program. In addition to this, C also provides predefined macros. This article by Simplilearn will help you understand macros in depth.

What Are Macros in C?

Macro in c is defined by the #define directive. Macro is a name given to a piece of code, so whenever the compiler encounters a macro in a program, it will replace it with the Macro value. In the macro definition, the semicolon does not terminate the macros (;).

Syntax of a Macro:

```
#define macro_name macro_value;
```

Example:

```
#define pi 3.14;
```

Example Program:

```
#include <stdio.h>
#define MACRO 10 //macro definition

int main(){
 printf("the value of a is: %d", MACRO);
    return 0;
}
```

Why Do We Use Macros in C?

Macros in C programs increase program efficiency. Rather than repeatedly mentioning a piece of code in the programs, we can define the constant value once and use it frequently. Macros in C have functions like macros in which we can pass the arguments, which makes the program run faster.

@copyright: EWU CSE, Dhaka, Bangladesh

Inline Function

By declaring a function `inline`, you can direct GCC to **make calls to that function faster**. One way GCC can achieve this is by integrating that function's code into the code for its callers. This makes execution faster by **eliminating the function-call overhead;** if any of the actual argument values are constant, their known values permit simplifications at **compile time** so that not all of the inline function's code needs to be included. Depending on the case, the effect on code size is less predictable; object code may be larger or smaller with function inlining.

A C program can be optimized with **two goals: to reduce the Code size or to get the best performance (time). Usually, these are two opposite ends of the line. To reduce the code size, the program is optimized so that it compromises performance to some extent. With a compiler set to optimize for** performance in terms of execution time, **the generated binary** is usually of higher code size. Since it can be requirement-dependent, it is usually a trade-off between what fits the requirement.

Inline functions are typically seen as a means of getting higher performance, which means reduced execution times.

☐ Inline functions:

gcc.gnu.org says, https://gcc.gnu.org/onlinedocs/gcc/Inline.html

By declaring a function inline, you can direct GCC to make calls to that function faster. One way GCC can achieve this is by integrating that function's code into the code for its callers. This makes execution faster by eliminating the **function-call overhead;** if any of the actual argument values are constant, their known values permit simplifications at compile time so that not all of the inline function's code needs to be included. Depending on the case, the effect on code size is less predictable; object code may be larger or smaller with function inlining.

So, it tells the compiler to build the function into the code, which is used to improve execution time.

@copyright: EWU CSE, Dhaka, Bangladesh

If you `declare small functions` like setting/clearing a flag or some bit toggle that is repeatedly performed inline, it can make a significant performance difference `in terms of time`, but at the cost of code size.

☐ non-static inline and Static inline

Again referring to gcc.gnu.org,

When an inline function is not static, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-static inline function is always compiled in the usual fashion.

❖ `extern inline`?

Again, gcc.gnu.org, says it all:

If you specify both inline and extern in the function definition, then the definition is used only for inlining. The function is not compiled independently, even if you refer to its address explicitly. Such an address becomes an external reference as if you had only declared the function and had not defined it.

This combination of inline and extern has almost the effect of a macro. It is used to put a function definition in a header file with these keywords and put another copy of the definition (lacking inline and extern) in a library file. The definition in the header file causes most calls to the function to be inlined. If any uses of the function remain, they refer to the single copy in the library.

☐ Popular Question by Programmers

- `static inline void f(void) {}` has no practical difference with `static void f(void) {}`.
- `inline void f(void) {}` in C doesn't work the C++ way. How does it work in C?
- What does `extern inline void f(void)` do?

@copyright: EWU CSE, Dhaka, Bangladesh

Executive Summary

- For `inline void f(void){}`, `inline` definition is only valid in the current translation unit.
- For `static inline void f(void) {}` Since the storage class is `static`, the identifier has internal linkage, and the inline definition is invisible in other translation units.
- For `extern inline void f(void),` Since the storage class is `extern`, the identifier has external linkage, and the inline definition also provides the external definition.

@copyright: EWU CSE, Dhaka, Bangladesh

Callback Function

Reference:
https://stackoverflow.com/questions/142789/what-is-a-callback-in-c-and-how-are-they-implemented

There is no "callback" in C - not more than any other generic programming concept.

They're implemented using function pointers. Here's an example:

```
void populate_array(int *array, size_t arraySize,int
(*getNextValue)(void)){
   for (size_t i=0; i<arraySize; i++)
      array[i] = getNextValue();
}
int getNextRandomValue(void) {
 return rand();
}
int main(void) {
    int myarray[10];
    populate_array(myarray, 10, getNextRandomValue);
}
```

Here, the `populate_array` function takes a function pointer as its third parameter and calls it to get the values with which to populate the array. We've written the `callback getNextRandomValue`, which returns a random-ish value, and passed a pointer to `populate_array`. `populate_array` will call our callback function 10 times and assign the returned values to the elements in the given array.

Here is an example of **`callback`** in C.

Let's say you want to write some code that allows you to register a **`callback`** to be called when an event occurs.

First, define the type of function used for the **`callback:`**

```
typedef void (*event_cb_t)(const struct event *evt, void *userdata);
```

Now, define a function that is used to register a **`callback`**

```
int event_cb_register(event_cb_t cb, void *userdata);
```


@copyright: EWU CSE, Dhaka, Bangladesh

This is what code would look like that registers a **callback**

```
static void my_event_cb(const struct event *evt, void *data)
{
      /* do stuff and things with the event */
} ...

event_cb_register(my_event_cb, &my_custom_data); …
```

In the internals of the event dispatcher, the **callback** may be stored in a struct that looks something like this:

```
struct event_cb { event_cb_t cb; void *data; };
```

This is what the code looks like that executes a **callback**

```
struct event_cb *callback; ... /* Get the event_cb that you want   to execute */

 callback->cb(event, callback->data):
```

C-BitFields

@copyright: EWU CSE, Dhaka, Bangladesh

Reference:
https://learn.microsoft.com/en-us/cpp/c-language/c-bit-fields?view=msvc-170

☐ Syntax

struct-declarator:
declarator
type-specifier *Declarator*$_{opt}$ **:** *constant-expression*

The ***constant-expression*** specifies the width of the field in bits. The ***type-specifier*** for the declarator must be unsigned int, signed int, or int, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no declarator. Arrays of bit fields, pointers to bit fields, and functions returning bit fields aren't allowed. The optional declarator names the bit field. Bit fields can only be declared as part of a structure. The **address-of-operator (&)** can't be applied to bit-field components.

Unnamed **bit fields** can't be referenced, and their contents at run-time are unpredictable. They can be used as **"dummy"** fields for alignment purposes. An unnamed bit-field whose width is **0** guarantees that storage for the member following it in the ***struct-declaration-list*** begins on an int boundary.

The **number of bits** in a bit field must be **less than or equal** to the size of **the underlying type.**

For example, these two statements aren't legal:

```
short a:17;         /* Illegal! */
int long y:33;      /* Illegal! */
```

This example defines a two-dimensional array of structures named screen.

```
struct
{
    unsigned short icon : 8;
    unsigned short color : 4;
    unsigned short underline : 1;
    unsigned short blink : 1;
} screen[25][80];
```


@copyright: EWU CSE, Dhaka, Bangladesh

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: icon, color, underline, and blink. Each structure is 2 bytes in size.

Bit fields have the same semantics as the integer type. A bit field is used in expressions exactly as a variable of the same base type would be used. It doesn't matter how many bits are in the bit field.

☐ Microsoft Specific

Bit fields defined as int are treated as signed. A Microsoft extension to the ANSI C standard allows char and long types (both signed and unsigned) for bit fields. Unnamed bit fields with base type long, short, or char (signed or unsigned) force alignment to a boundary appropriate to the base type.

Bit fields are allocated within an integer from least significant to most-significant bit. In the following code:

```
struct example_bitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;
```

```
int main( void ){
    test.a = 2;
    test.b = 31;
    test.c = 0;
    return 0;
}
```

the bits of `test` would be arranged as follows:

00000001 11110010

ccccccb bbbbaaaa

Since the 8086 family of processors stores the low byte of integer values before the high byte, the integer **0x01F2** would be stored in physical memory as **0xF2** followed by **0x01**. The ISO C99 standard lets an implementation choose whether a bit field may straddle two storage instances. Consider this structure, which stores bit fields that total 64 bits:

A standard C implementation could pack these bit fields into two 32-bit integers. It might store **tricky_bits.may_straddle** as 16 bits in one 32-bit integer and 14 bits in the next 32-bit integer. The Windows ABI convention packs bit fields into single storage integers and doesn't straddle storage units. The Microsoft compiler stores each bit field in the above example to fit entirely in a 32-bit integer. In this case, the first and second are stored in one integer, and may_straddle is

@copyright: EWU CSE, Dhaka, Bangladesh

stored in a second integer, while the last is stored in a third integer. The sizeof operator returns 12 on an instance of tricky_bits.

☐ Padding and Alignment of Structure Members

ANSI 3.5.2.1 The padding and alignment of members of structures and whether a bit-field can straddle a storage-unit boundary

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address, and the last member the highest.

Every data object has an alignment requirement. The alignment requirement for all data except structures, unions, and arrays is either the object's size or the current packing size (specified with either /Zp or the `pack` pragma, whichever is less). For structures, unions, and arrays, the alignment requirement is the largest alignment requirement of its members. Every object is allocated an offset so that

```
offset % alignment-requirement == 0
```

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.

@copyright: EWU CSE, Dhaka, Bangladesh

Function Pointers in C

The function name is Nothing but a simple regular variable

Function pointers in C hold function addresses, allowing direct calls. Declared with an asterisk and function parameters, they enable callbacks and array integrations. This article covers their declaration, use, and the concept of functions as memory-resident entities.

☐ Understanding Function Pointers in C

A function pointer in C is a variable that stores the address of a function. It allows you to refer to a function by using its pointer and later call the function using that function pointer. This provides flexibility in calling functions dynamically and can be helpful in scenarios where you want to select and call different functions based on certain conditions or requirements.
In C, function pointers are declared using the asterisk (*) operator to obtain the value saved in an address.

☐ Advantages of Function Pointers

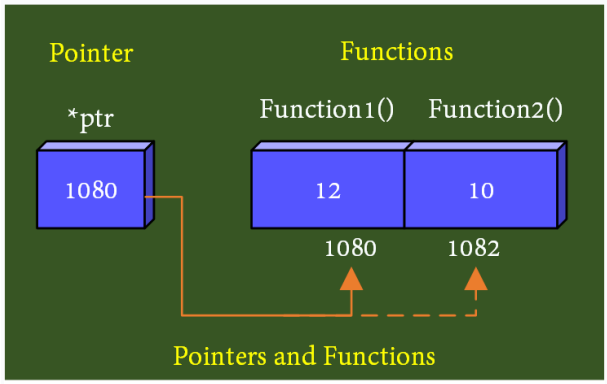
A function pointer in C is helpful to generate function calls to which they point. Hence, the programmers can pass them as arguments. The function pointers enhance the code's readability. You can access many identical functions using a single line of code.

Other prominent advantages of `function pointer in C` include `passing functions as parameters` to different functions. Therefore, you can create generic functions that can be used with any function, provided it possesses the right signature.

☐ Declaring a Function Pointer in C

Now that we know functions have a unique memory address, we can use function pointers in C to point to the first executable code inside a function body.

@copyright: EWU CSE, Dhaka, Bangladesh



For example, in the above figure, we have a function `add()` to add two integer numbers. Here, the function name points to the function's address, so we are using a function pointer `fptr` that stores the **address of the beginning** of the function `add(a, b)`, which is **1001** in this case.

> Before using a `function pointer` we need to declare them to tell the compiler the `type of function` a `pointer` can `point to.`

☐ Syntax of Function Pointer in C

The general syntax of a function pointer is,

```
return_type (* pointer_name) (datatype_arg_1, datatype_arg_1, ...);
```

Declaring a function pointer in C is comparable to declaring a function, except that when a function pointer is declared, we prefix its name with, we need to declare it to tell the compiler what `type of function it` an Asterisk * symbol.

For example, if a function has the declaration

```
float foo (int, int);
```

Declaration of a function pointer in C for the function foo will be

```
// function pointer declaration
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
float (*foo_pointer) (int, int);


/*
assigning the address of the function (foo)
to function pointer
*/

foo_pointer = foo;
```

Here, pointer `*foo_pointer` is a function pointer that stores the memory address of a function foo, takes two arguments of type int, and returns a value of data type float.

Tricky example

```
void *(*fp) (int *, int *);
```

At first glance, this example seems complex, but the trick to understanding such declarations is to read them inside out. Here, `(*fp)` is a function pointer like a regular one as like int `*ptr`. This `function pointer (*fp)` can point to functions with two int * type arguments and has a return type of `void *`, as we can see from its declaration where `(int *, int *)` explains the type and number of arguments and `void *` is the return type from the pointed function.

Note: It is essential to declare a function before assigning its address to a function pointer in C.

☐ Calling a Function Through a Function Pointer in C

Calling a function using a pointer is similar to calling a function in the usual way using `the function's name.`

Suppose we declare a function and its pointer as given below.

```
int (*pointer) (int); // function pointer declaration

int areaSquare (int); // function declaration

pointer = areaSquare;
```

To call the `function areaSquare,` we can **create a function call** using any of the **three ways:**

@copyright: EWU CSE, Dhaka, Bangladesh

```
int length = 5;

// Different ways to call the function//

1.    using function name

      int area = areaSquare(length);

2.    using function pointer (a)

      int area = (*pointer)(length);

3.    using function pointer (b)

int area = pointer(length);
```

The effect of calling `functions` using `pointers or their names` is the same. It is not compulsory to call the function with the indirection `operator (*)` as shown in the second case. Still, it is good practice to use the indirection operator to clear out that the function is called using a pointer as `(*pointer)()` is more readable when compared to calling function from pointers with parentheses `pointer()`.

☐ Example for `function Ptr`

Now that we know the syntax for declaring and using function pointers in C to create a function call, let's see an example of creating a function pointer to call the function that returns the area of a rectangle.

@copyright: EWU CSE, Dhaka, Bangladesh

```c
#include<stdio.h>
// function declaration
int areaRectangle(int, int);

int main()
{
    int length, breadth, area;

    // function pointer declaration
    // note that our pointer declaration has identical
    // arguments as the function it will point to
    int (*fp)(int, int);

    printf("Enter length and breadth of a rectangle\n");
    scanf("%d %d", &length, &breadth);

    // pointing the pointer to functions memory address
    fp = areaRectangle;

    // calling the function using function pointer
    area = (*fp)(length, breadth);

    printf("Area of rectangle = %d", area);
    return 0;
}
// function definition
int areaRectangle(int l, int b) {
    int area_of_rectangle = l * b;
    return area_of_rectangle;
}
```

Output

```
Enter length and breadth of a rectangle

5 9
Area of rectangle = 45
```

Here, we have defined a function `areaRectangle()` that takes two integer inputs and returns the area of the rectangle. To store the reference of the function, we use a `function pointer (*fp)` that has a declaration similar to the function it points to.

To point the function address to the pointer, we don't need to use the `&` symbol as the function name areaRectangle also represents the function's address. To call the function, we pass parameters inside the parenthesis `((*fp)(length, breadth))`, and the return value is stored in the variable area.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Example: Array of function pointers

Arrays are data structures that store collections of identical data types. Like any other data type, we can create an array to store function pointers in C. Function pointers can be accessed from their indexes like we access normal array values arr[i]. This way, we create an array of function pointers, where each array element stores a function pointer pointing to different functions.

This approach is useful when we do not know in advance which function is called, as shown in the example.

```c
#include<stdio.h>

float add(int, int);
float multiply(int,int);
float divide(int,int);
float subtract(int,int);

int main() {

    int a, b;

    float (*operation[4])(int, int);

    operation[0] = add;
    operation[1] = subtract;
    operation[2] = multiply;
    operation[3] = divide;

    printf("Enter two values ");
    scanf("%d%d", &a, &b);

    float result = (*operation[0])(a, b);
    printf("Addition (a+b) = %.1f\n", result);

    result = (*operation[1])(a, b);
    printf("Subtraction (a-b) = %.1f\n", result);

    result = (*operation[2])(a, b);
    printf("Multiplication (a*b) = %.1f\n", result);

    result = (*operation[3])(a, b);
    printf("Division (a/b) = %.1f\n", result);

    return 0;
}
float add(int a, int b) {
    return a + b;
}
float subtract(int a, int b) {
    return a - b;
}
float multiply(int a, int b) {
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
    return a * b;
}
float divide(int a, int b) {
    return a / (b * 1.0);
}
```

Output

```
Enter two values 3 2
Addition (a+b) = 5.0
Subtraction (a-b) = 1.0
Multiplication (a*b) = 6.0
Division (a/b) = 1.5
```

Here, we have stored the addresses of four functions in an array of function pointers. We used this array to call the required function using the function pointer stored in this array.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Data Structure ADT

Abstraction is generally The process of working with ideas rather than their implementation.

Abstraction in OOP is one of the essential vital features, which hides its internal and other implementation details and cleanly separates its interface from the implementation of the different modules. In other words, the user will have just the knowledge of what an entity is doing instead of its internal workings.

How to Achieve Abstraction in OOP?

In OOP, we can achieve Data Abstraction using Abstract classes and interfaces. Interfaces allow 100% abstraction (complete abstraction). Interfaces will enable you to abstract the implementation thoroughly.
Abstract classes allow 0 to 100% abstraction (partial to complete abstraction) because they can contain concrete methods whose implementation results in a partial abstraction.

Abstract Classes:

- An **Abstract class** is a class whose objects can't be created. An **Abstract** class is made through the use of the **abstract keyword**. It is used to represent a concept.
- An abstract class can have abstract methods (**methods** without the body) and non-abstract or concrete methods (methods with the body). A non-abstract class cannot have abstract methods.
- The class must be declared **abstract** if it contains at least one abstract method.
- **An abstract class** does not allow you to create objects of its type. In this case, we can only use the objects of its subclass.
- Using an **abstract class**, we can achieve 0 to 100% abstraction.
- There is always a **default constructor** in an abstract class; it can also have a parameterized constructor.
- The **abstract class** can also contain final and static methods.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Syntax of declaring an abstract class:

```
abstract class ClassName{
    //class body
}
```

☐ Abstract Methods in OOP

- **Abstract methods** are methods with **no implementation and without a method body.**
- An **abstract** method is declared with an **abstract** keyword.
- The **child classes** that **inherit** the **abstract** must implement these inherited **abstract methods.**

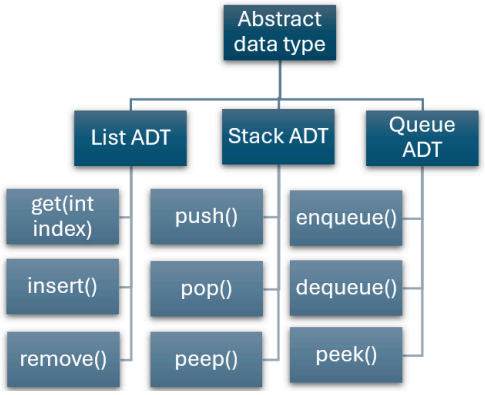
1. List ADT

The List Abstract Data Type is a type of list that contains similar elements in sequential order. The list ADT is a collection of elements that have a linear relationship with each other. A linear relationship means that each list element has a unique successor.

The List ADT is an interface; other classes give the actual implementation of the data type. For example, Array Data Structure internally implements the ArrayList class, while List Data Structure internally implements the LinkedList class.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ List of Abstract Data Type (ADT)



The List *interface of the Java* library specifies 25 different operations/methods. Following are some of the operations that we can perform on the list:

- **`get(int index)`:** Returns an element at the specified index from the list.
- **`insert()`:** Inserts an element at any position.
- **`remove()`:** Removes the first occurrence of any component from a list.
- **`removeAt()`:** Removes the element at a predefined area from a non-empty list.
- **`Replace()`:** Replaces an element with another element.
- **`size()`:** Returns the number of elements of the list.
- **`isEmpty()`:** Returns true if the list is empty; otherwise, it returns false.
- **`isFull()`:** Returns true if the list is full; else, returns false.

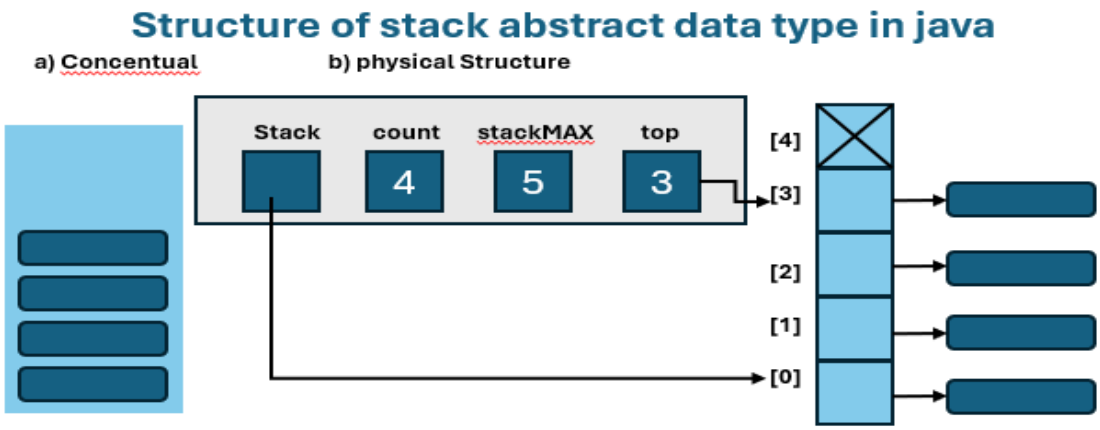
☐ Stack ADT

A stack is a LIFO ("Last In, First Out") data structure with similar elements arranged in an ordered sequence. All the operations in the stack take place at the top of the stack.

- Stack ADT is a collection of homogeneous data items (elements) in which all insertions and deletions occur at one end, called the top of the stack.
- In Stack ADT Implementation, there is a pointer to the data instead of storing the data at each node.
- The program allocates the memory for the data and passes the address to the stack ADT.
- The start node and the data nodes encapsulate together in the ADT. Only the pointer to the stack is visible to the calling function.
- The stack head structure also contains a pointer to the top and the count of the number of entries currently in the stack.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ The below diagram shows the whole structure of the Stack ADT:



We can perform the following operations on the stack –

- ❖ **push():** It inserts an element at the top of the stack if it is not full.
- ❖ **pop():** It removes or pops an element from the top of the stack if it is not empty.
- ❖ **peep():** Returns the top element of the stack without removing it.
- ❖ **size():** Returns the size of the stack.
- ❖ **isEmpty():** If the stack is empty, it returns true; otherwise, it returns false.
- ❖ **isFull():** If the stack is full, it returns true; else, it returns false.

☐ Queue ADT

A Queue is a FIFO ("First In, First Out") data structure containing similar elements arranged sequentially. We can perform the operations on a queue at both ends; insertion takes place at the rear end, and deletion takes place at the front end.

Queue ADT is a collection in which the arrangement of the elements of the same type is sequential.

- ❖ The Queue abstract data type (ADT) design is the same as the basic design of the Stack ADT.
- ❖ Each node of the queue contains a void pointer to the data and a link pointer to the next element of the queue. The program allocates the memory for storing the data.

Operations performed on the queue are as follows:

- ❖ **enqueue():** It inserts or adds an element at the end of the queue.

@copyright: EWU CSE, Dhaka, Bangladesh

- **`dequeue()`** : Removes an element from the front side of the queue.
- **`peek()`** : Returns the starting element of the queue without removing it.

- **`size()`:** This function returns the number of elements in the queue.
- **`isEmpty()`:** If the queue is empty, it returns true; otherwise it returns false.
- **`isFull()`:** If the queue is full, it returns true; otherwise, it returns false.

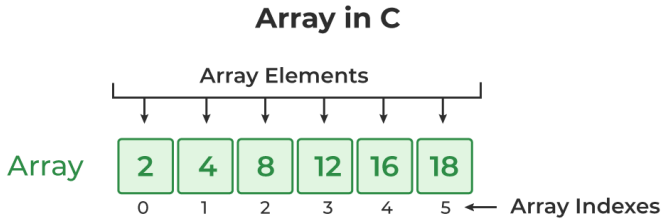
Designing an Abstract Data Type:

We must choose good operations to design an abstract data type and determine how they should behave. Here are a few rules for developing an ADT.

- Combining simple and few operations in powerful ways rather than many complex operations is better.
- Each operation in an Abstract Data Type should have a clear purpose and logical behavior rather than a range of exceptional cases. All the special cases would make the operation difficult to understand and use.
- The set of operations should be adequate so that there are enough kinds of computations that users likely want to do.
- The type may be either generic, such as a graph, list, or set, or domain-specific, such as an employee database, a street map, a phone book, etc. However, it should not combine generic and domain-specific features.

☐ Prerequisite:

- **Array:** An array is a variable that can store multiple values.



- Arrays have 0 as the first index, not 1. In this example, array[0] is the first element. Array[0] = 2, 2 is the value of the 1st index.
- If the size of an array is n, to access the last element, the n-1 index is used. In this example, our array size is 6, but the last element is array[5]

@copyright: EWU CSE, Dhaka, Bangladesh

- Suppose the starting address of Array[0] is 2120d. Then, the address of the array[1] will be 2124d. Similarly, the address of Array[2] will be 2128d, and so on.

- This is because the size of an int is 4 bytes. (address according to the data type)

A simple example of an integer array :

```c
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>
int main() {
  int values[5];
  printf("Enter 5 integers: ");
  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }
  printf("Displaying integers: ");
  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```

Output:

```
Enter 5 integers: 1
-3
34
0
3
Displaying integers: 1
-3
34
0
3
```


@copyright: EWU CSE, Dhaka, Bangladesh

☐ Pointer:

A pointer is a programming concept used in computer science to reference or point to a memory location that stores a value or an object. It is a variable that stores the memory address of another variable or data structure rather than storing the data itself.

The variable -> var
It's address -> &var

```c
#include <stdio.h>

int main(){
 int var;  // initialization of a variable.
 var =5;  // declaration of a variable.
 printf("value of the variable = %d \n",var);
 printf("address of the variable = %d \n",&var);
 return 0;
}
```

Output:

```
value of the variable = 5
address of the variable = 6422296
```

Now, we can access the address of a variable through a pointer.

After initializing, `int *ptr;` (ptr is a `pointer variable` which points to an address which holds the value of `type integer`)

`ptr -> `	`ptr` is a pointer variable that holds the address.	`ptr == &var`
`*ptr -> `	This holds the value of the pointing address.	`*ptr == var`
`* -> `	The * operator denotes the pointer's underlying value. This is known as "dereferencing" or "indirect"	

❖ 1st declare a pointer:

```c
// pointers in various ways, such as
 int *ptr;
 int* ptr;
 int * ptr;
```


@copyright: EWU CSE, Dhaka, Bangladesh

- 2nd assign address to the pointer;

```
// right way of assigning an address to a pointer.
 ptr = &var; //both ptr and &var are addresses.
   *ptr = var; // both c and *pc are values.

// wrong way of assigning value to the pointer.
*ptr = &var;  // &c is address but *pc is not.
ptr = var;  // pc is the address, but c is not.
```

- 3rd, manipulate the value (print, change, etc.)

We can print a variable's value and address using a pointer.

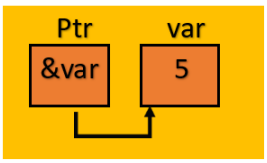
```
#include <stdio.h>
int main()
{
//We can declare pointers in various ways, such as
 int *ptr;
 int var;
 ptr = &var; //correct assigning.

    printf ("value of the variable = %d \n",var);
    printf ("address of the variable = %d \n\n",&var);
    printf ("value of the variable through pointer = %d \n",*ptr);
    printf ("address of the variable through pointer = %d \n\n", ptr);
 return 0;
}
```

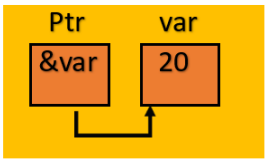
Output:

```
value of the variable = 5
address of the variable = 6422296

value of the variable through pointer = 5
address of the variable through pointer = 6422296
```

We can change the value of a variable, and after changing, the pointer will still show the correct value, as the pointer (ptr) only takes and stores the address of the variable.

@copyright: EWU CSE, Dhaka, Bangladesh





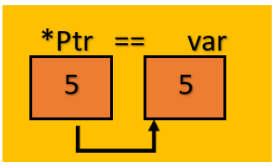

```c
#include <stdio.h>
int main()
{
//We can declare pointers in various ways, such as
    int *ptr, var;
    var = 5;  ptr = &var;

    printf ("value of the variable through pointer = %d \n",*ptr);
    var = 20; //changing the value of the variable.
    printf ("value of the variable after change = %d \n",*ptr);
 return 0;
}
```

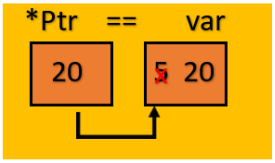
Output:

```
value of the variable  = 20
value of the variable after incrementation = 21
value of the variable after decrementation = 20
```

```
value of the variable through pointer = 5
value of the variable after change = 20
```

We can change the value of a variable through the pointer as *ptr holds the value.






```c
    int *ptr;
    int var;
    var = 5;
    ptr = &var;

    printf ("value of the variable through pointer = %d \n",*ptr);
    *ptr = 20; //changing the variable's value by pointer.
     printf ("value of the variable after change = %d \n",*ptr);//pointer shows the
value after change.
```


@copyright: EWU CSE, Dhaka, Bangladesh

Output:

```
value of the variable through pointer = 5
value of the variable after change = 20
```

Increment and decrement of a pointer.

`++*ptr` -> means incrementing the value of the pointing variable

`--*ptr` -> means decrementing the value of the pointing variable

```c
#include <stdio.h>
int main()
{
    int *ptr, var;
    var = 20;
    ptr = &var;
    printf ("value of the variable  = %d \n",*ptr);
    ++*ptr; //incrementing the value of the variable.
    printf ("value of the variable after incrementation = %d \n",*ptr);

    --*ptr; //decrementing the value of the variable.
    printf ("value of the variable after decrementation = %d \n",*ptr);
    return 0;
}
```

Introduction to data structure

@copyright: EWU CSE, Dhaka, Bangladesh

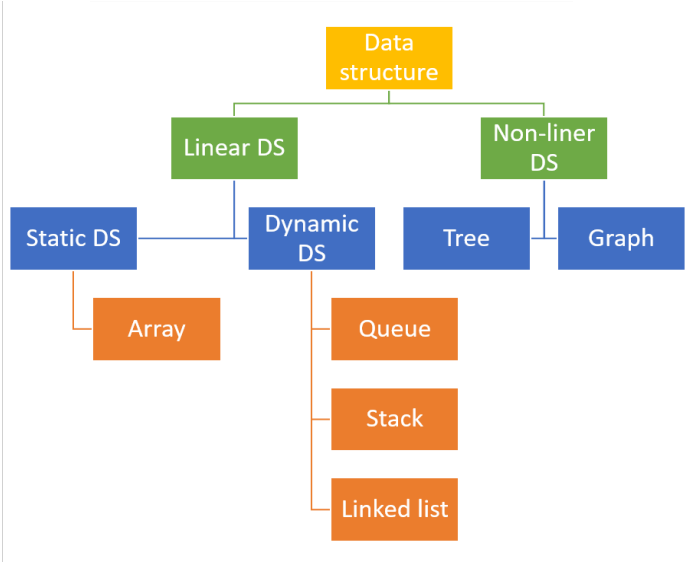
What is Data Structure?

A data structure is a way to organize and store information in computer memory to be used efficiently.

It's like different types of containers or organizers for your data, just like a bag for your books or a shelf for your clothes.

Data structures help organize and manage different types of information in a computer. They make it easier to access, search, and manipulate data depending on what you need to do with it.

Classification of data structure:



☐ Linked List

@copyright: EWU CSE, Dhaka, Bangladesh

❖ What is a linked list?

A linked list is like a chain, where each link (or node) holds an item and a reference (or link) to the next item in the list. This allows you to connect many items together in a sequence, making it easy to add, remove, or rearrange them as needed.

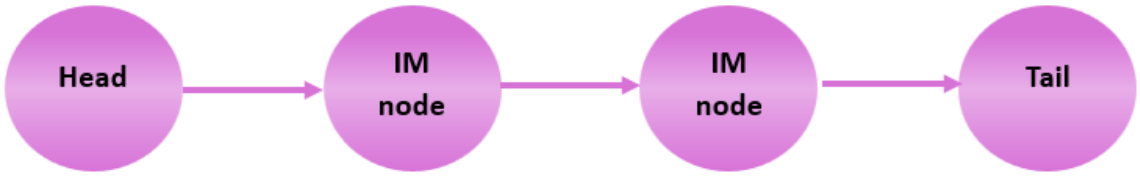
Types of linked lists:

There is a lot of variation; here are the three mainly used

❖ **Singly Linked List:**

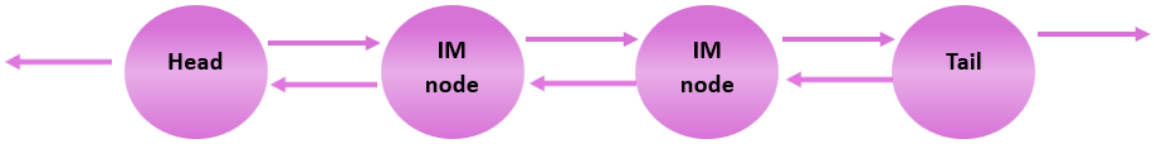
Each node points to the next node.

`IM` – **intermediate**



❖ **Doubly Linked List:**

Each node points to the next and the previous node, traversing quickly in both directions.



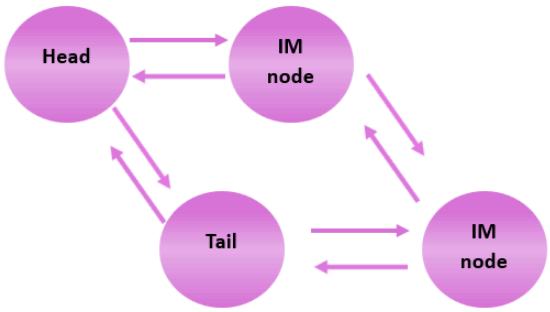
❖ **Circular Linked List:**

The last node connects to the first node, creating a closed loop.

@copyright: EWU CSE, Dhaka, Bangladesh

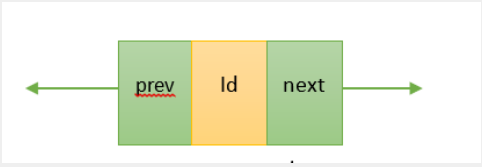
Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory



Sample:

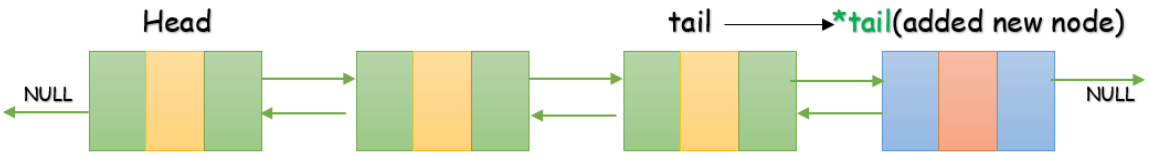
```
struct student
{
      uint32_t id;
      struct student *next;
      struct student *prev;

};
struct student *head, *std, *tail;
      std = (struct
student*)malloc(sizeof(struct
student));
      head = std;
      head -> id = 1;
      head -> next = NULL;
      head -> prev = NULL;
```



☐ Let's learn straightforward operations we can do on a linked list

1. Addition

Add a new node after the list

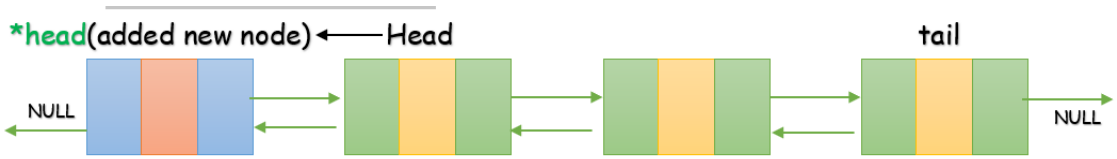


@copyright: EWU CSE, Dhaka, Bangladesh

To add a new node after the list we have to make the node tail, as it's now the last element of the list.

```
newNode = (struct student*)malloc(sizeof(struct student));
newNode -> prev = tail ->next;
newNode -> next = NULL;
tail = newNode;
```

☐ Add a new node before the list.

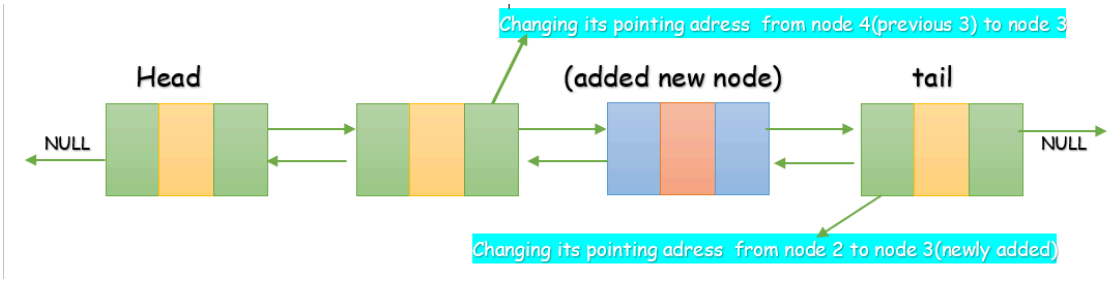


To add a new node before the list, we have to make the node head, as it's now the first element of the list.

```
newNode = (struct student*)malloc(sizeof(struct student));
newNode -> prev = NULL;
newNode -> next = head ->prev;
head = newNode;
```

Add a new node middle of the list

@copyright: EWU CSE, Dhaka, Bangladesh

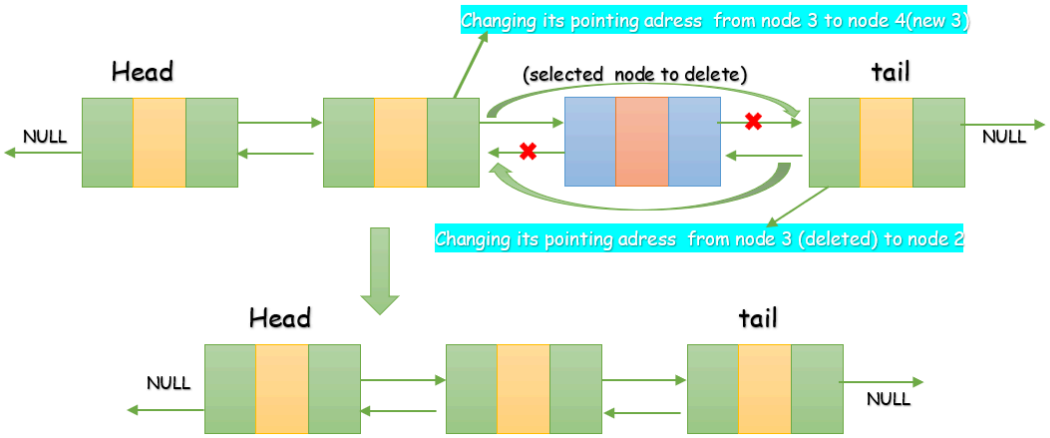


We must connect the node with its previous and next nodes to add a new node in the middle of the list.

*Some information will be provided on where to add the new node(index or after a value). *

```
newNode = (struct student*)malloc(sizeof(struct student));
struct student *curr // temporary variable to keep the track.
Curr = head;//then traverse through the list and reach the target
           newNode->prev = curr;
           newNode->next = curr->next;
           curr->next->prev = newNode;
curr->next = newNode;
```

☐ Delete a node from the middle of the `list`



@copyright: EWU CSE, Dhaka, Bangladesh

To delete a new node in the middle of the list, we have to make the node's previous and next nodes connect with each other's.

*Some information will be provided on where to delete the node (index or after a value).

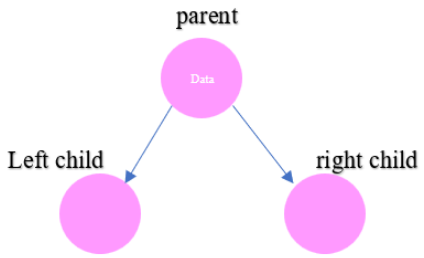
```
struct student       // temporary variable to keep the track.

Curr = head ; //then traverse through the list and reach the target
curr->prev->next = curr->next;

//now the node before the deleted node will point the node after the deleted
node.

curr->next->prev = curr->prev;
free(curr);
```

☐ TREE Data Structure

Binary tree:

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

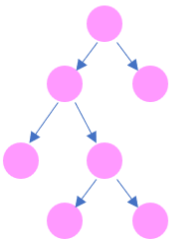
- ❖ data item
- ❖ address of the left child
- ❖ address of the right child



❖ Full Binary Tree

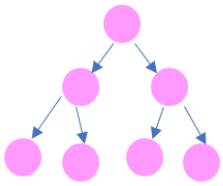
A full Binary tree is a particular type of binary tree in which every parent node/internal node has two or no children.

@copyright: EWU CSE, Dhaka, Bangladesh



❖ Perfect Binary Tree

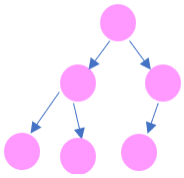
A perfect binary tree is one in which every internal node has exactly two child nodes, and all the leaf nodes are at the same level.



❖ Complete Binary Tree

A complete binary tree is just like a full binary tree but with two significant differences.

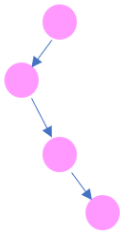
- Every level must be filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling; i.e., a complete binary tree doesn't have to be a full binary tree.



❖ Degenerate or Pathological Tree

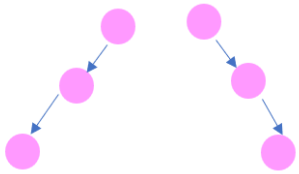
A degenerate or pathological tree has a single child, either left or right.

@copyright: EWU CSE, Dhaka, Bangladesh



❖ Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the left or right nodes dominate the tree. Thus, there are two skewed binary trees: left-skewed binary trees and right-skewed binary trees.



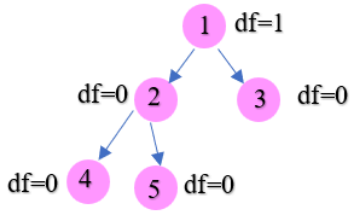
❖ Balanced Binary Tree:

It is a binary tree in which the difference between the height of the left and the right subtree for each node is 0 or 1.

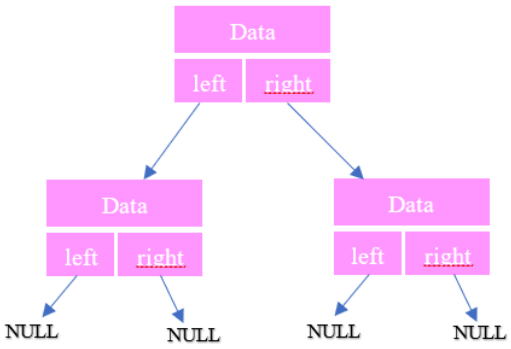
@copyright: EWU CSE, Dhaka, Bangladesh

Towards the Core of Computer Science:

Storing, organizing and retrieving the data from the memory



In terms of code nodes store data like



☐ Let's do some code:

1. Create a structure of a node:

```
struct binarytree {
  int item;
  struct binarytree* left; //pointer to keep track of left items
  struct binarytree* right; //pointer to keep track of left items
};
```

2. Create a binary tree:

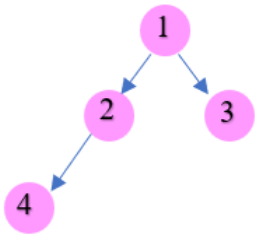
```
struct binarytree* createNode(value) {
  struct binarytree* newNode = malloc(sizeof(struct binarytree));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}
// Insert on the left of the binarytree
struct binarytree* insertLeft(struct binarytree* root, int value) {
  root->left = createNode(value);
  return root->left;
}
// Insert on the right of the binarytree
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
struct binarytree* insertRight(struct binarytree* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct binarytree* root = createNode(1);
  insertLeft(root, 2);
  insertRight(root, 3);
  insertLeft(root->left, 4);

}
```

Output:

- We created a tree like this



- We have three different ways to traverse and print the value:

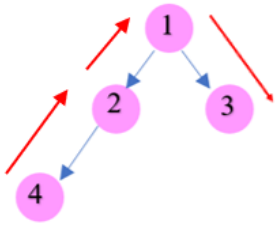
  Traverse last left child(leaf) to last right child(leaf)

```
// Inorder traversal
void inorderTraversal(struct binarytree* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);}
```

Output:

```
Inorder traversal
4 ->2 ->1 ->3 ->
```


@copyright: EWU CSE, Dhaka, Bangladesh

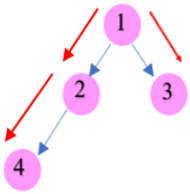


2. Traverse root to left child till leaf, then last right child till leaf

```
// Preorder traversal

void preorderTraversal(struct binarytree* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);

  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

Output:

```
Preorder traversal
1 ->2 ->4 ->3 ->
```



3. Traverse the last left child(leaf) to root, then the last right child(leaf) to root, and print the root at the end.

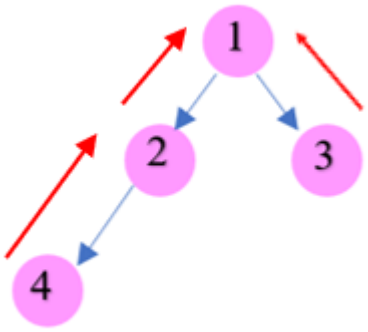
```
// Postorder traversal

void postorderTraversal(struct binarytree* root) {
  if (root == NULL) return;
   postorderTraversal(root->left);
  postorderTraversal(root->right);
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
  printf("%d ->", root->item);

}
```

Output:

```
Postorder traversal
4 ->2 ->3 ->1 ->
```



@copyright: EWU CSE, Dhaka, Bangladesh

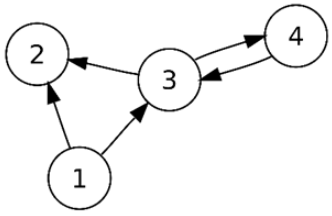
☐ Graph Theory

☐ Directed graph:

In mathematics, and more specifically in graph theory, A directed graph (or digraph) is a graph made up of vertices connected by directed edges, often called arcs.

A directed graph is an ordered pair $G = (V, A)$ where

- V is a set whose elements are called *vertices*, *nodes*, or *points*;
- A is a set of ordered pairs of vertices, called *arcs*, *directed edges* (sometimes simply *edges* with the corresponding set named E instead of A), *arrows*, or *directed lines*.

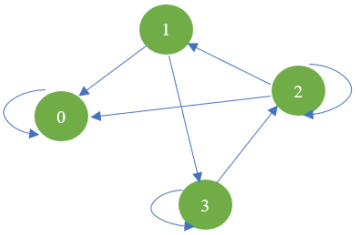


☐ Here is a code to show a graph by adjacency Matix:

```c
#include<stdio.h>
int main(){
   int a[4][4];// 2D array declaration as a adjacency matix
   int i, j;
   for(i=0; i<4; i++) {
      for(j=0;j<4;j++) {
         printf("Enter value for array[%d][%d]:", i, j);
         scanf("%d", &a[i][j]);
      }
   }
   printf("Here is the adjacency matix for given graph:\n");
   for(i=0; i<4; i++) {
      for(j=0;j<4;j++) {
         printf("<%d, %d, %d >\n",i, j, a[i][j]);
      }
   }
   return 0;
}
```

Suppose the given graph is the following:

@copyright: EWU CSE, Dhaka, Bangladesh



If the value of any element `a[i][j]` is 1, it represents an edge connecting vertex i and vertex j.

```
Enter value for array[0][0]:1
Enter value for array[0][1]:0
Enter value for array[0][2]:0
Enter value for array[0][3]:0
Enter value for array[1][0]:1
Enter value for array[1][1]:0
Enter value for array[1][2]:0
Enter value for array[1][3]:1
Enter value for array[2][0]:1
Enter value for array[2][1]:1
Enter value for array[2][2]:1
Enter value for array[2][3]:0
Enter value for array[3][0]:0
Enter value for array[3][1]:0
Enter value for array[3][2]:1
Enter value for array[3][3]:1
Here is the adjacency matix for given graph:
<0, 0, 1 >
<0, 1, 0 >
<0, 2, 0 >
<0, 3, 0 >
<1, 0, 1 >
<1, 1, 0 >
<1, 2, 0 >
<1, 3, 1 >
<2, 0, 1 >
<2, 1, 1 >
<2, 2, 1 >
<2, 3, 0 >
<3, 0, 0 >
<3, 1, 0 >
<3, 2, 1 >
<3, 3, 1 >
```

☐ Graph Traversal:

Graph Traversal means visiting every vertex and edge exactly once in a well-defined order. When using specific graph algorithms, this must be ensured. The order in which the vertices are visited is essential and may depend upon the algorithm or question. Tracking which vertices have been visited during a traversal is necessary. The most common way of tracking vertices is to mark them.

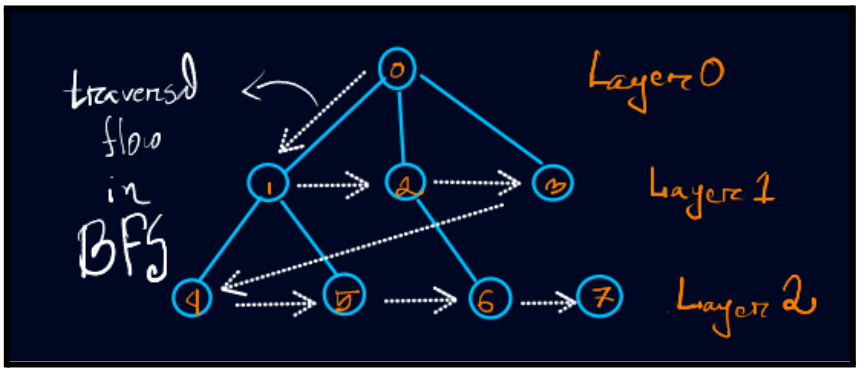
@copyright: EWU CSE, Dhaka, Bangladesh

☐ Breadth First Search (BFS):

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm that starts by traversing from a selected node (source or starting node). It then traverses the graph layerwise, exploring the neighbor nodes (nodes directly connected to the source node). Finally, it must move towards the next-level neighbor nodes.

The name BFS suggests to traverse the graph breadthwise as follows:



- First, move horizontally and visit all the nodes of the current layer.
- Move to the next layer.

Unlike trees, Graphs may contain cycles so we may return to the same node. We divide the vertices into two categories to avoid processing a node more than once.

- Visited and
- Non visited

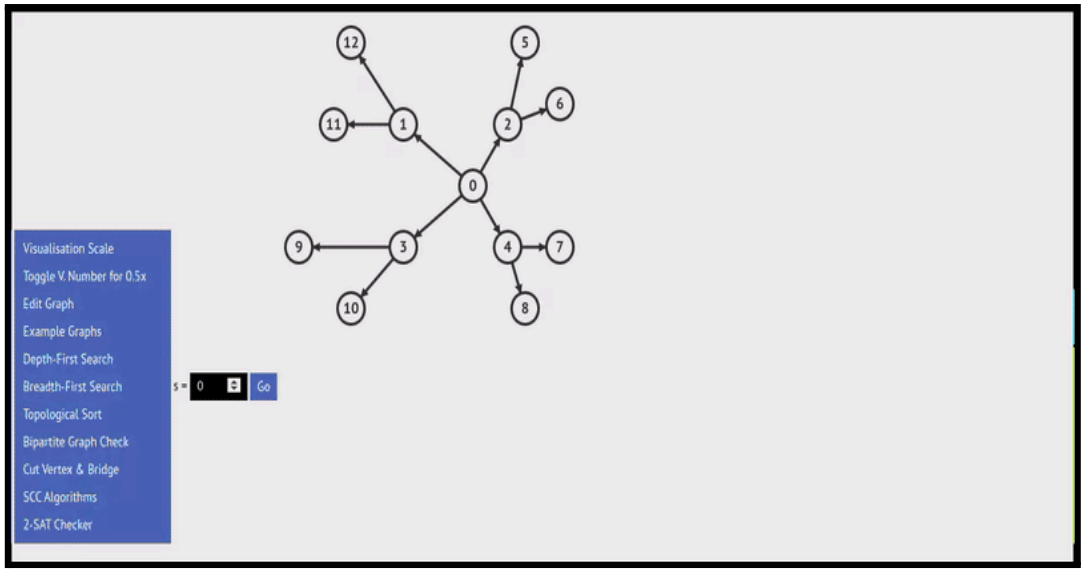
A boolean visited array is used to mark the visited vertices.

Pseudo Code:

```
BFS (G, s)                  //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )            //Stores w in Q to further visit its neighbour
                         mark w as visited.
```

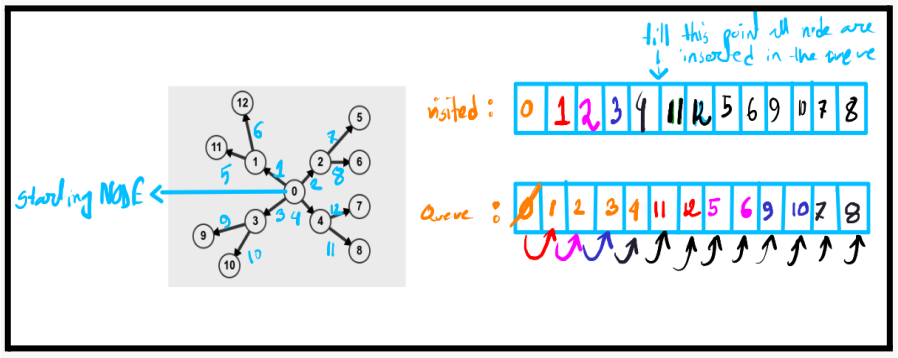

@copyright: EWU CSE, Dhaka, Bangladesh

Analysis:



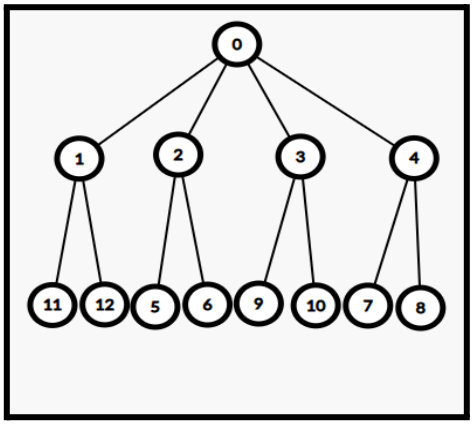
☐ Simulation of BFS:

Here is the illustration drawn with the help of a queue and visited array.



@copyright: EWU CSE, Dhaka, Bangladesh

Figure: BFS Spanning Tree of the Graph




```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
using std::vector;
#define MAX_VERTICES 20

/*!
   Structure for creating Queue
*/
struct Q_Node
{
   int v;
   struct Q_Node *next;
};

struct Q_Node *head, *tail, *curr;

int s = 0;

void enqueue(int v);
void dequeue();
int peek();
int sizee();
bool isEmpty();

/*!
   Declaring vector and array to store the graph and keep track to the visited
status.
*/
vector<int> v[MAX_VERTICES];
int level[MAX_VERTICES];
bool vis[MAX_VERTICES];

vector<int> visList;

/*!
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
   @param int s represents the starting node.
   Every time, the first element of the queue is accessed and traversed to all
adjacent nodes.
   Every time we visit a newNode, adding it to the visit vector
*/
void BFS(int s)
{
   enqueue(s);
   level[s] = 0;
   vis[s] = true;
   visList.push_back(s);

   while (!isEmpty()){
       int p = peek();
       dequeue();
       for (int i = 0; i < v[p].size(); ++i){
           if (vis[v[p][i]] == false){
               level[v[p][i]] = level[p] + 1;

               enqueue(v[p][i]);
               vis[v[p][i]] = true;
               visList.push_back(v[p][i]);
           }
       }
   }
}

int main()
{
   v[0].push_back(1);
   v[0].push_back(2);
   v[0].push_back(3);
   v[0].push_back(4);

   v[1].push_back(11);
   v[1].push_back(12);
   v[1].push_back(0);

   v[2].push_back(5);
   v[2].push_back(6);
   v[2].push_back(0);

   v[3].push_back(9);
   v[3].push_back(10);
   v[3].push_back(0);

   v[4].push_back(7);
   v[4].push_back(8);
   v[4].push_back(0);

   v[11].push_back(1);
   v[12].push_back(1);

   v[5].push_back(2);
   v[6].push_back(2);

   v[9].push_back(3);
   v[10].push_back(3);
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
    v[7].push_back(4);
    v[8].push_back(4);

    v[7].push_back(15);
    v[15].push_back(7);

    BFS(0);
    printf("Visited List Order : ");

    for (int i = 0; i < visList.size(); ++i) {
        printf("%d ", visList[i]);
    }
    printf("\n");
}
/*!
   @param int val represents the value of the current node.
  Inserts a node at the end.
*/
void enqueue(int val)
{
    if (s == 0)
    {
        struct Q_Node *temp = (struct Q_Node *)malloc(sizeof(struct Q_Node));
        if (temp == NULL){
            exit(0);
        }
        temp->v = val;
        temp->next = NULL;
        head = temp;
        curr = temp;
        tail = temp;
        s++;
    } else {
        struct Q_Node *temp = (struct Q_Node *)malloc(sizeof(struct Q_Node));
        if (temp == NULL){
            exit(0);
        }
        temp->v = val;
        temp->next = NULL;
        curr->next = temp;
        curr = temp;
        tail = temp;
        s++;
    }
}
/*!
  Removes the node from the start
*/
void dequeue()
{
    if (s == 0) {
        return;
    }
    struct Q_Node *temp = head;
    head = head->next;
    free(temp);
    s--;
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
}
/*!
  Returns the value of the first element of the queue
*/
int peek()
{
   return head->v;
}

/*!
  Returns the size of  the queue
*/
int sizee()
{
   return s;
}

/*!
  Returns status of the queue Emptiness
*/
bool isEmpty()
{
   if (s == 0)
   {
       return true;
   }
   else
   {
       return false;
   }
}
```

Output:

```
mehraj@mehraj-B550GTA:~/Desktop/ewu/dsa209/Data_Structure_207$ g++ BFS.cpp -o ou
tputBfs
mehraj@mehraj-B550GTA:~/Desktop/ewu/dsa209/Data_Structure_207$ ./outputBfs
Visited List Order : 0 1 2 3 4 11 12 5 6 9 10 7 8 15
mehraj@mehraj-B550GTA:~/Desktop/ewu/dsa209/Data_Structure_207$
```


@copyright: EWU CSE, Dhaka, Bangladesh

☐ Depth First Search (DFS):

Depth First Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure.

DFS Pseudocode:

```
DFS (G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

Init () {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS (G, u)
}
```


@copyright: EWU CSE, Dhaka, Bangladesh

Advanced C Programming For Data Structure

Table Of Contents

- ☐ Macro
- ☐ Inline Function
- ☐ Callback Function
- ☐ Function Pointer
- ☐ Bit field in c

Reference: https://gcc.gnu.org/onlinedocs/gcc/Inline.html

@copyright: EWU CSE, Dhaka, Bangladesh

Macros

In c programming, we can use macros to repeatedly use a single value or a piece of code in our programs. By defining macros once in our program, we can use them frequently throughout the program. In addition to this, C also provides predefined macros. This article by Simplilearn will help you understand macros in depth.

What Are Macros in C?

Macro in c is defined by the #define directive. Macro is a name given to a piece of code, so whenever the compiler encounters a macro in a program, it will replace it with the Macro value. In the macro definition, the semicolon does not terminate the macros (;).

Syntax of a Macro:

```
#define macro_name macro_value;
```

Example:

```
#define pi 3.14;
```

Example Program:

```
#include <stdio.h>
#define MACRO 10 //macro definition

int main(){
 printf("the value of a is: %d", MACRO);
    return 0;
}
```

Why Do We Use Macros in C?

Macros in C programs increase program efficiency. Rather than repeatedly mentioning a piece of code in the programs, we can define the constant value once and use it frequently. Macros in C have functions like macros in which we can pass the arguments, which makes the program run faster.

@copyright: EWU CSE, Dhaka, Bangladesh

Inline Function

By declaring a function `inline`, you can direct GCC to make calls to that function faster. One way GCC can achieve this is by integrating that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; if any of the actual argument values are constant, their known values permit simplifications at compile time so that not all of the inline function's code needs to be included. Depending on the case, the effect on code size is less predictable; object code may be larger or smaller with function inlining.

A C program can be optimized with two goals: to reduce the Code size or to get the best performance (time). **Usually, these are two opposite ends of the line. To reduce the code size, the program is optimized so that it compromises performance to an extent. With a compiler set to optimize for** performance in terms of execution time, the generated binary is usually of higher code size. Since it can be requirement-dependent, it is usually a trade-off between what fits the requirement.

Inline functions are typically seen as a means of getting higher performance, which means reduced execution times.

☐ `inline functions:`

gcc.gnu.org says, https://gcc.gnu.org/onlinedocs/gcc/Inline.html

By declaring a function inline, you can direct GCC to make calls to that function faster. One way GCC can achieve this is by integrating that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; if any of the actual argument values are constant, their known values permit simplifications at compile time so that not all of the inline function's code needs to be included. Depending on the case, the effect on code size is less predictable; object code may be larger or smaller with function inlining.

So, it tells the compiler to build the function into the code where it is used to improve execution time.

@copyright: EWU CSE, Dhaka, Bangladesh

If you `declare small functions` like setting/clearing a flag or some bit toggle performed repeatedly inline, it can make a significant performance difference `with respect to time`, but at the cost of code size.

☐ non-static inline and Static inline

Again referring to gcc.gnu.org,

When an inline function is not static, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-static inline function is always compiled in the usual fashion.

❖ `extern inline?`

Again, gcc.gnu.org says it all:

If you specify both inline and extern in the function definition, then the definition is used only for inlining. The function is not compiled independently, even if you refer to its address explicitly. Such an address becomes an external reference as if you had only declared the function and had not defined it.

This combination of inline and extern has almost the effect of a macro. It is used to put a function definition in a header file with these keywords and put another copy of the definition (lacking inline and extern) in a library file. The definition in the header file causes most calls to the function to be inlined. If any uses of the function remain, they refer to the single copy in the library.

☐ Popular Question by Programmers

- `static inline void f(void) {}` has no practical difference with `static void f(void) {}`.
- `inline void f(void) {}` in C doesn't work the C++ way. How does it work in C?
- What does `extern inline void f(void);` do?

☐ Executive Summary

@copyright: EWU CSE, Dhaka, Bangladesh

1. For `inline void f(void){}`, `inline` definition is only valid in the current translation unit.
2. For `static inline void f(void) {}` Since the storage class is `static`, the identifier has internal linkage, and the inline definition is invisible in other translation units.
3. For `extern inline void f(void),` Since the storage class is `extern`, the identifier has external linkage, and the inline definition also provides the external definition.

@copyright: EWU CSE, Dhaka, Bangladesh

Callback Function

https://stackoverflow.com/questions/142789/what-is-a-callback-in-c-and-how-are-they-implemented

There is no "callback" in C - not more than any other generic programming concept.

They're implemented using function pointers. Here's an example:

```
void populate_array(int *array, size_t arraySize,int
(*getNextValue)(void)){
     for (size_t i=0; i<arraySize; i++)
        array[i] = getNextValue();
}
int getNextRandomValue(void) {
   return rand();
}
int main(void) {
      int myarray[10];
      populate_array(myarray, 10, getNextRandomValue);
}
```

Here, the `populate_array` function takes a function pointer as its third parameter, and calls it to get the values to populate the array with. We've written the `callback getNextRandomValue`, which returns a random-ish value, and passed a pointer to it to `populate_array`. `populate_array` will call our callback function 10 times and assign the returned values to the elements in the given array.

Here is an example of **`callback`** in C.

You want to write some code that allows registering **`callback`** to be called when some event occurs. First, define the type of function used for the **`callback:`**

```
typedef void (*event_cb_t)(const struct event *evt, void *userdata);
```

Now, define a function that is used to register a **`callback`**

```
int event_cb_register(event_cb_t cb, void *userdata);
```

This is what the code would look like that registers a **`callback`**

```
static void my_event_cb(const struct event *evt, void *data)
{
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
        /* do stuff and things with the event */
} ...

event_cb_register(my_event_cb, &my_custom_data); …
```

In the internals of the event dispatcher, the **callback** may be stored in a struct that looks something like this:

```
struct event_cb { event_cb_t cb; void *data; };
```

This is what the code looks like that executes a **callback**

```
struct event_cb *callback; ... /* Get the event_cb that you want to
execute */

callback->cb(event, callback->data):
```


@copyright: EWU CSE, Dhaka, Bangladesh

C-BitFields

Reference: https://learn.microsoft.com/en-us/cpp/c-language/c-bit-fields?view=msvc-170

Syntax

struct-declarator:
  declarator
  type-specifier $Declarator_{opt}$: *constant-expression*

The *constant-expression* specifies the width of the field in bits. The *type-specifier* for the declarator must be unsigned int, signed int, or int, and the *constant-expression* must be a nonnegative integer value. If the value is zero, the declaration has no declarator. Arrays of bit fields, pointers to bit fields, and functions returning bit fields aren't allowed. The optional declarator names the bit field. Bit fields can only be declared as part of a structure. The **address-of operator (&)** can't be applied to bit-field components.

Unnamed **bitfields** can't be referenced, and their contents at run-time are unpredictable. They can be used as **"dummy"** fields for alignment purposes. An unnamed bit-field whose width is **0** guarantees that storage for the member following it in the *struct-declaration-list* begins on an int boundary.

The **number of bits** in a bit field must be **less than or equal** to the size of the underlying type.

For example, these two statements aren't legal:

```
short a:17;         /* Illegal! */
int long y:33;      /* Illegal! */
```

This example defines a two-dimensional array of structures named screen.

```
struct
{
    unsigned short icon: 8;
    unsigned short color: 4;
    unsigned short underline: 1;
    unsigned short blink: 1;
} screen[25][80];
```

The array contains 2,000 elements. Each element is an individual structure containing four bit-field members: icon, color, underline, and blink. Each structure is 2 bytes in size.

@copyright: EWU CSE, Dhaka, Bangladesh

Bit fields have the same semantics as the integer type. A bit field is used in expressions exactly as a variable of the same base type would be used. It doesn't matter how many bits are in the bit field.

☐ Microsoft Specific

Bit fields defined as int are treated as signed. A Microsoft extension to the ANSI C standard allows char and long types (both signed and unsigned) for bit fields. Unnamed bit fields with base type long, short, or char (signed or unsigned) force alignment to a boundary appropriate to the base type.

Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code:

```
struct example_bitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;
```

```
int main( void ){
    test.a = 2;
    test.b = 31;
    test.c = 0;
    return 0;
}
```

the bits of `test` would be arranged as follows:
00000001 11110010
cccccccb bbbbaaaa

Since the 8086 family of processors stores the low byte of integer values before the high byte, the integer **0x01F2** would be stored in physical memory as **0xF2** followed by **0x01**. The ISO C99 standard lets an implementation choose whether a bit field may straddle two storage instances. Consider this structure, which stores bit fields that total 64 bits:

A standard C implementation could pack these bit fields into two 32-bit integers. It might store `tricky_bits.may_straddle` as **16 bits in one 32-bit integer and 14 bits in the next** 32-bit integer. The Windows ABI convention packs bit fields into single storage integers and doesn't straddle storage units. The Microsoft compiler stores each bit field in the above example to fit entirely in a 32-bit integer. In this case, the first and second are stored in one integer, may_straddle is stored in a second integer, and the last is stored in a third integer. The sizeof operator returns 12 on an instance of tricky_bits.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Padding and Alignment of Structure Members

ANSI 3.5.2.1 The padding and alignment of members of structures and whether a bit-field can straddle a storage-unit boundary

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address, and the last member has the highest.

Every data object has an alignment requirement. The alignment requirement for all data except structures, unions, and arrays is either the object's or the current packing size (specified with either /Zp or the `pack` pragma, whichever is less). For structures, unions, and arrays, the alignment requirement is the largest alignment requirement of its members. Every object is allocated an offset so that

```
offset % alignment-requirement == 0
```

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the following bit field fits into the current allocation unit without crossing the boundary imposed by the standard alignment requirements of the bit fields.

@copyright: EWU CSE, Dhaka, Bangladesh

Function Pointers in C

The function name is Nothing but a simple regular variable

Function pointers in C hold function addresses, allowing direct calls. Declared with an asterisk and function parameters, they enable callbacks and array integrations. This article covers their declaration, use, and the concept of functions as memory-resident entities.

☐ Understanding Function Pointers in C

A function pointer in C is a variable that stores the address of a function. It allows you to refer to a function by using its pointer and later call the function using that function pointer. This provides flexibility in calling functions dynamically and can be useful in scenarios where you want to select and call different functions based on certain conditions or requirements.
The function pointers in C are declared using the asterisk (*) operator to obtain the value saved in an address.

☐ Advantages of Function Pointers

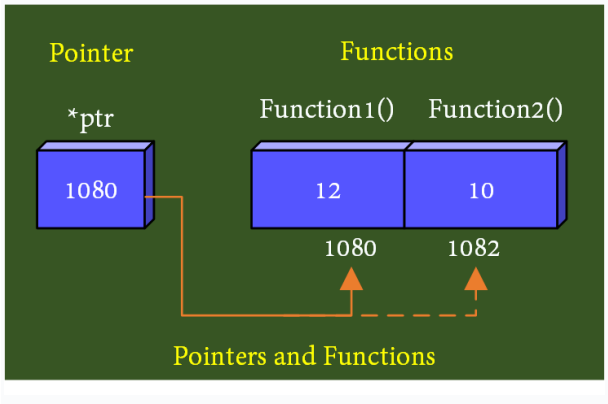
A function pointer in C helps generate function calls to which they point. Hence, the programmers can pass them as arguments. The function pointers enhance the code's readability. You can access many identical functions using a single line of code.

Other prominent advantages of `function pointer in C` include `passing functions as parameters` to different functions. Therefore, you can create generic functions that can be used with any function, provided it possesses the right signature.

☐ Declaring a Function Pointer in C

Now that we know that functions have a unique memory address, we can use function pointers in C that point to the first executable code inside a function body.

@copyright: EWU CSE, Dhaka, Bangladesh



For example, in the above figure, we have a function **add()** to add two integer numbers. Here, the function name points to the function's address, so we use a function pointer **fptr** that stores the **address of the beginning** of the function **add(a, b)**, which is **1001** in this case.

> Before using a **function pointer**, we need to declare them to tell the compiler the **type of function a pointer** can **point to.**

☐ Syntax of Function Pointer in C

The general syntax of a function pointer is,

```
return_type (* pointer_name) (datatype_arg_1, datatype_arg_1, ...);
```

Declaring a function pointer in C is comparable to declaring a function, except when a function pointer is declared, we prefix its name, which is an Asterisk * symbol.

For example, if a function has the declaration

```
float foo (int, int);
```

Declaration of a function pointer in C for the function foo will be

```
// function pointer declaration
float (*foo_pointer) (int, int);
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
/*
assigning the address of the function (foo)
to function pointer
*/

foo_pointer = foo;
```

Here, pointer `*foo_pointer` is a function pointer that stores the memory address of a function foo, takes two arguments of type int, and returns a value of data type float.

Tricky example

```
void *(*fp) (int *, int *);
```

At first glance, this example seems complex, but the trick to understanding such declarations is to read them inside out. Here, `(*fp)` is a function pointer like a normal C pointer, like int `*ptr`. This `function pointer (*fp)` can point to functions with two int * type arguments and has a return type of `void *`, as we can see from its declaration where `(int *, int *)` explains the type and number of arguments and `void *` is the return type from the pointed function.

Note: It is important to declare a function before assigning its address to a function pointer in C.

☐ Calling a Function Through a Function Pointer in C

Calling a function using a pointer is similar to calling a function in the usual way using `the function's name.`

Suppose we declare a function and its pointer as given below.

```
int (*pointer) (int); // function pointer declaration

int areaSquare (int); // function declaration

pointer = areaSquare;
```

To call the `function areaSquare,` we can **create a function call** using any of the **three ways:**

@copyright: EWU CSE, Dhaka, Bangladesh

```
int length = 5;

// Different ways to call the function//

   1. using function name

      int area = areaSquare(length);

   2. using function pointer (a)

      int area = (*pointer)(length);

   3. using function pointer (b)

int area = pointer(length);
```

The effect of calling `functions` using `pointers or their names` is the same. It is not compulsory to call the function with the indirection `operator (*),` as shown in the second case. Still, it is good practice to use the indirection operator to clear out that the function is called using a pointer as `(*pointer)()` is more readable when compared to calling function from pointers with parentheses `pointer()`.

☐ Example for `function Ptr`

Now that we know the syntax of how function pointers in C are declared and used to create a function call. Let us see an example where we are creating a function pointer to call the function that returns the area of a rectangle.

@copyright: EWU CSE, Dhaka, Bangladesh

```c
#include<stdio.h>
// function declaration
int areaRectangle(int, int);

int main()
{
    int length, breadth, area;

    // function pointer declaration
    // note that our pointer declaration has identical
    // arguments as the function it will point to
    int (*fp)(int, int);

    printf("Enter length and breadth of a rectangle\n");
    scanf("%d %d", &length, &breadth);

    // pointing the pointer to functions memory address
    fp = areaRectangle;

    // calling the function using function pointer
    area = (*fp)(length, breadth);

    printf("Area of rectangle = %d", area);
    return 0;
}
// function definition
int areaRectangle(int l, int b) {
    int area_of_rectangle = l * b;
    return area_of_rectangle;
}
```

Output

```
Enter length and breadth of a rectangle

5 9
Area of rectangle = 45
```

Here, we have defined a function `areaRectangle()` that takes two integer inputs and returns the area of the rectangle. To store the reference of the function, we use a `function pointer (*fp)` that has a declaration similar to the function it points to. To point the function address to the pointer, we don't need to use & symbol as the function name areaRectangle also represents the function's address. To call the function, we pass parameters inside the parenthesis **((*fp)(length, breadth))**, and the return value is stored in the variable area.

@copyright: EWU CSE, Dhaka, Bangladesh

☐ Example: Array of function pointers

Arrays are data structures that store collections of identical data types. Like any other data type we can create an array to store function pointers in C. Function pointers can be accessed from their indexes like we access normal array values arr[i]. This way, we create an array of function pointers, where each array element stores a function pointer pointing to different functions.

This approach is useful when we do not know in advance which function is called, as shown in the example.

```c
#include<stdio.h>

float add(int, int);
float multiply(int,int);
float divide(int,int);
float subtract(int,int);

int main() {

    int a, b;

    float (*operation[4])(int, int);

    operation[0] = add;
    operation[1] = subtract;
    operation[2] = multiply;
    operation[3] = divide;

    printf("Enter two values ");
    scanf("%d%d", &a, &b);

    float result = (*operation[0])(a, b);
    printf("Addition (a+b) = %.1f\n", result);

    result = (*operation[1])(a, b);
    printf("Subtraction (a-b) = %.1f\n", result);

    result = (*operation[2])(a, b);
    printf("Multiplication (a*b) = %.1f\n", result);

    result = (*operation[3])(a, b);
    printf("Division (a/b) = %.1f\n", result);

    return 0;
}
```


@copyright: EWU CSE, Dhaka, Bangladesh

```
float add(int a, int b) {
    return a + b;
}
float subtract(int a, int b) {
    return a - b;
}
float multiply(int a, int b) {
    return a * b;
}
float divide(int a, int b) {
    return a / (b * 1.0);
}
```

Output

```
Enter two values 3 2
Addition (a+b) = 5.0
Subtraction (a-b) = 1.0
Multiplication (a*b) = 6.0
Division (a/b) = 1.5
```

Here, we have stored the addresses of four functions in an array of function pointers. We used this array to call the required function using the function pointer stored in this array.

To be continued … …

DStructCSD Compiler

@copyright: EWU CSE, Dhaka, Bangladesh

Dr. Hasan Mahmood Aminul Islam

Assistant Professor, Dept of CSE

East West University, Bangladesh

Specialist System-on-Chip Software (5G) Nokia (2019-2022)

Nokia Headquarters, Espoo, Finland

D.Sc in Technology (CSE) (2013-2018)

Aalto University, Finland

M.Sc in Computer Science

University of Helsinki, Finland

B.Sc in CSE, BUET, Bangladesh

Web Page: https://www.ewubd.edu/faculty-profile/mahmood.aminul

Personal Website: https://hmaislam.github.io/myweb/

@copyright: EWU CSE, Dhaka, Bangladesh