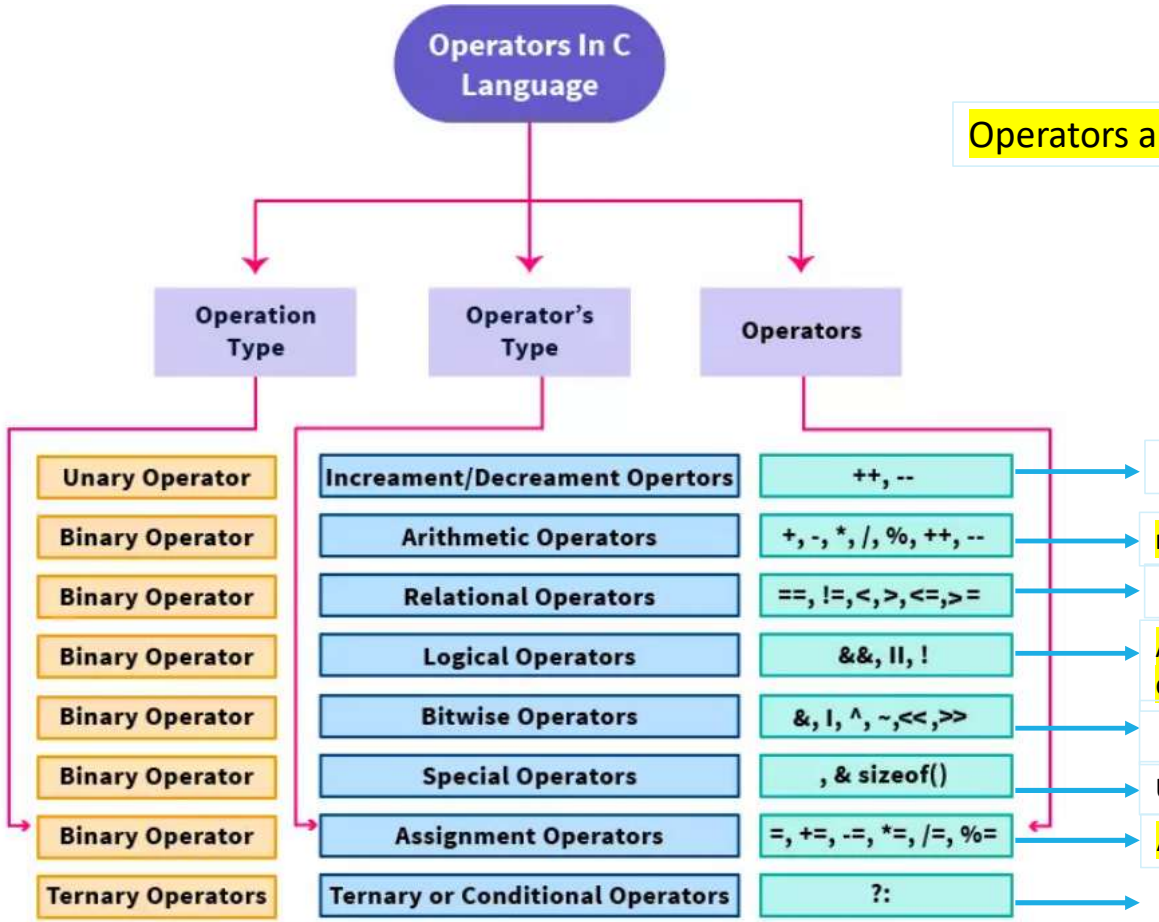
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   int a, b, temp;

   // Convert command line arguments to integers
   a = atoi(argv[1]);
   b = atoi(argv[2]);

   // Swap using a temporary variable
   temp = a;
   a = b;
   b = temp;

   // Print swapped values
   printf("a=%d b=%d\n", a, b);

   return 0;
}
```

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

int main(int argc, char *argv[]) {
   int a, b;

   // Convert command line arguments to integers
   a = atoi(argv[1]);
   b = atoi(argv[2]);

   // Swap without using a third variable
   a = a + b;
   b = a - b;
   a = a - b;

   // Print swapped values
   printf("a=%d b=%d\n", a, b);

   return 0;
}
```

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

int main(int argc, char *argv[]) {
   int a, b, c;

   // Convert three command line arguments to integers
   a = atoi(argv[1]);
   b = atoi(argv[2]);
   c = atoi(argv[3]);

   // Average using integer division
    printf("a = %d, b = %d, c = %d, argc = %d\n", a, b, c, argc);
    printf("the average is = %d\n", (a + b + c) / (argc-1));

   return 0;
}
```

The key point: **the counting happens before your program even runs**. The compiler doesn't do it — the OS does.

Operators in C



Operators are used to perform operations on variables and values.

increments/ decrements the values by 1

mathematical operations such as add, sub, etc.

checks the relationship between two operands

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false.

Performs bit-level operations

Unary operator that returns the size of data

Assignment operator is used for assigning a value to a variable

Increment/Decrement Operators

```
#include <stdio.h>

int main() {
   int a = 5, b;

   printf("Initial value of a = %d\n\n", a);

   // Post-increment: value is used first, then incremented
   b = a++;
   printf("After b = a++:\n");
   printf("a = %d, b = %d\n\n", a, b);

   // Reset a
   a = 5;

   // Pre-increment: value is incremented first, then used
   b = ++a;
   printf("After b = ++a:\n");
   printf("a = %d, b = %d\n", a, b);

   return 0;
}
```

Issues

```
#include <stdio.h>

int main() {
   int a = 5;

   printf("a++ = %d, a = %d, ++a = %d\n", a++, a, ++a);

   return 0;
}
```

```
#include <stdio.h>

int main() {
   int a = 5, b;

   a = a++;
   printf("%d\n", a);
}
```

Arithmetic Operators

```c
#include <stdio.h>

int main() {
    int a, b;

    // Input two numbers
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    // Arithmetic operations
    printf("\nArithmetic Operations:\n");
    printf("%d + %d = %d\n", a, b, a + b);   // Addition
    printf("%d - %d = %d\n", a, b, a - b);   // Subtraction
    printf("%d * %d = %d\n", a, b, a * b);   // Multiplication
    printf("%d / %d = %d\n", a, b, a / b);   // Division (integer division)
    printf("%d %% %d = %d\n", a, b, a % b);  // Modulus (remainder)

    return 0;
}
```

Relational Operator

```
#include <stdio.h>

int main() {
   int a = 10, b = 20;

   printf("a = %d, b = %d\n\n", a, b);

   // Relational operators
   printf("a == b : %d\n", a == b);   // Equal to
   printf("a != b : %d\n", a != b);   // Not equal to
   printf("a > b  : %d\n", a > b);    // Greater than
   printf("a < b  : %d\n", a < b);    // Less than
   printf("a >= b : %d\n", a >= b);   // Greater than or equal to
   printf("a <= b : %d\n", a <= b);   // Less than or equal to

   return 0;
}
```

Assignment (shorthand) Operators

```
#include <stdio.h>
int main() {
   int a = 10, b = 3;
   printf("Initial values: a = %d, b = %d\n\n", a, b);

   a += b;   // a = a + b
   printf("After a += b  → a = %d\n", a);

   a -= b;   // a = a - b
   printf("After a -= b  → a = %d\n", a);

   a *= b;   // a = a * b
   printf("After a *= b  → a = %d\n", a);

   a /= b;   // a = a / b
   printf("After a /= b  → a = %d\n", a);

   a %= b;   // a = a % b
   printf("After a %%= b  → a = %d\n", a);
   return 0;
}
```

Ternary Operators

```
#include <stdio.h>

int main() {
   int num = 10;

   // Check if number is even or odd using ternary operator
   (num % 2 == 0) ? printf("%d is Even\n", num) : printf("%d is Odd\n", num);

   return 0;
}
```

```
condition ? expression_if_true : expression_if_false;
```

The ternary operator is a three-operand operator (?:) that evaluates a condition and returns one of two values depending on whether the condition is true or false.

Quick exercise for class

Write a C program to find the maximum of three numbers using ternary operator.

```c
#include <stdio.h>

int main() {
    int a, b, c, max;

    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    // Using ternary operator to find maximum
    max = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);

    printf("The maximum number is: %d\n", max);

    return 0;
}
```

sizeof Operator

```
#include <stdio.h>

int main() {
   int a;
   double b;
   char c;

   printf("Size of int = %zu bytes\n", sizeof(int));
   printf("Size of a   = %zu bytes\n", sizeof a); //show a+1, (a+1)
   printf("Size of double = %zu bytes\n", sizeof(b));
   printf("Size of char   = %zu bytes\n", sizeof(c));
   //printf("Size of pointer to int = %zu bytes\n", sizeof(int*));

   return 0;
}
```

The sizeof operator in C returns the size in bytes of a data type or a variable.

- Its return type is size_t (an unsigned integer type).
- The size depends on the system and compiler.
- When you use it with a **variable or expression**, parentheses are optional (unless needed for precedence):
- %zu is used for size_t return type.

Bitwise Operators

Operator	Symbol	Meaning
AND	&	Sets bit to 1 if both bits are 1
OR	|	Sets bit to 1 if at least one bit is 1
XOR	^	Sets bit to 1 if bits are different
NOT	~	Flips all bits (1→0, 0→1)
Left Shift	<<	Shifts bits left (×2 per shift)
Right Shift	>>	Shifts bits right (÷2 per shift)

```c
#include <stdio.h>
int main() {
   unsigned int a = 5;   // 0101 in binary
   unsigned int b = 9;   // 1001 in binary
   printf("a = %u, b = %u\n", a, b);
   // AND (&)
   printf("a & b = %u\n", a & b);   // 0101 & 1001 = 0001 (1)

   // OR (|)
   printf("a | b = %u\n", a | b);   // 0101 | 1001 = 1101 (13)

   // XOR (^)
   printf("a ^ b = %u\n", a ^ b);   // 0101 ^ 1001 = 1100 (12)

   // NOT (~)
   printf("~a = %u\n", ~a);        // Bitwise negation

   // Left Shift (<<)
   printf("a << 1 = %u\n", a << 1); // 0101 << 1 = 1010 (10)

   // Right Shift (>>)
   printf("b >> 1 = %u\n", b >> 1); // 1001 >> 1 = 0100 (4)
}
```

Control Statements

Allow different sets of instructions to be executed depending on whether certain conditions is evaluated to be true or false.

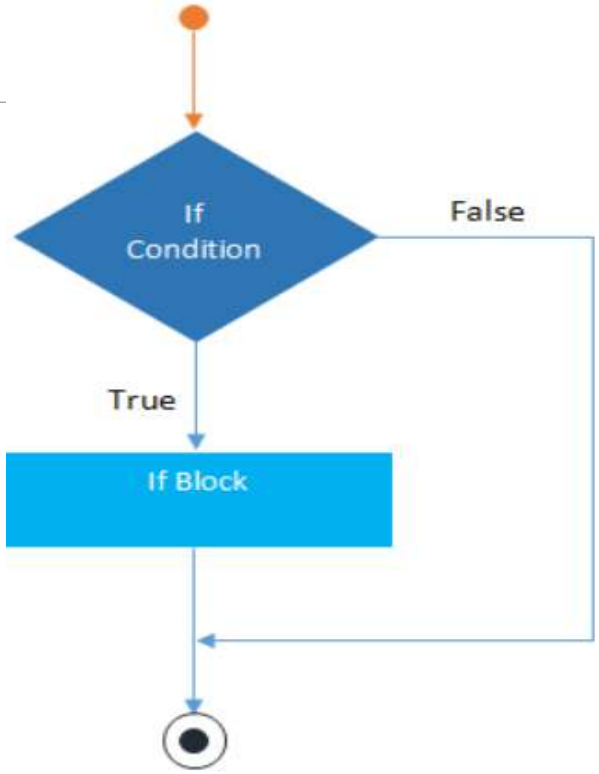
Also called branching.

A Condition can an expression

- Expression = non-zero value ---> **True**
- Expression = Zero value ------> **False**

Syntax

```
if(expression)
{
        //code to be executed
}
```

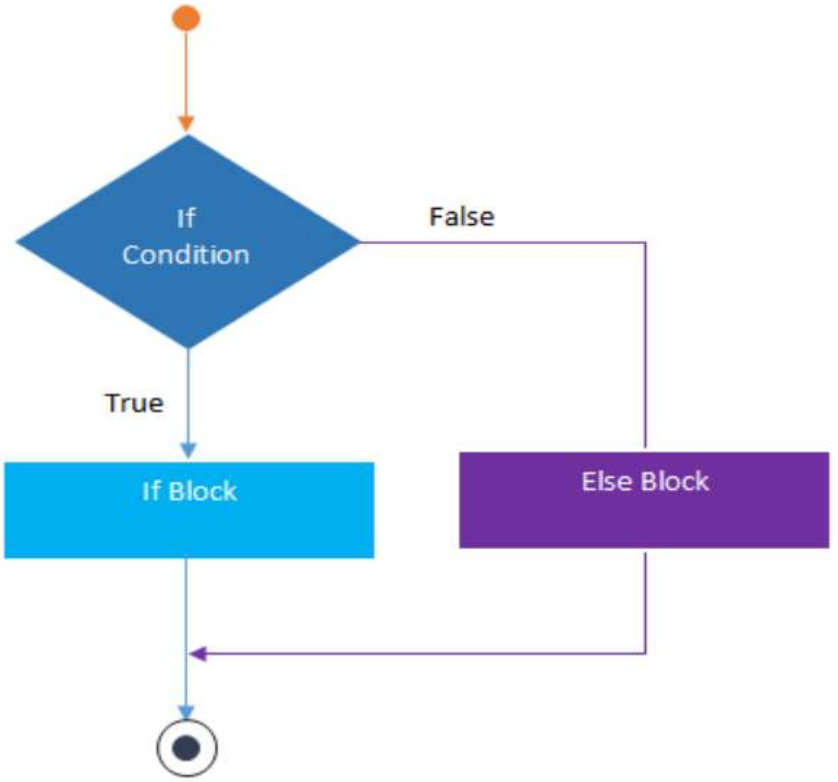


If statement

If-else: Control Statement

Syntax

```
if(expression)
{
        //Statements
}
else
{
        //Statements
}
```



If-Else statement

- IF without a else is possible.
- Else without a if is not possible.

simpleif1.c

```c
#include <stdio.h>

int main() {
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);

    // Check if number is positive
    if (number > 0) {
        printf("The number is positive.\n");
    }
    // Check if number is negative
    if (number < 0) {
        printf("The number is negative.\n");
    }
    // Check if number is zero
    if (number == 0) {
        printf("The number is zero.\n");
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int age;

    printf("Enter your age: ");
    scanf("%d", &age);

    if (age >= 18) {
        printf("You are eligible to vote.\n");
    } else {
        printf("You are not eligible to vote yet.\n");
    }

    return 0;
}
```

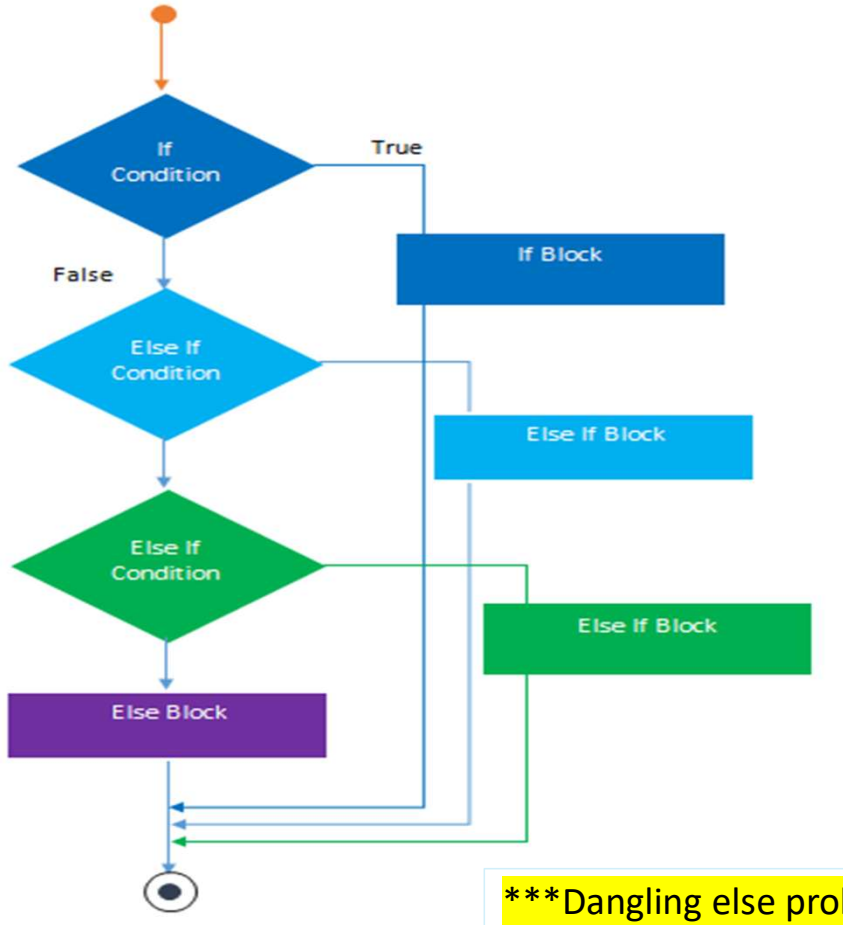
Nested if-else: Control Statements (Cont.…)

```
if( expression )
{
        if( expression1 )
        {
                statement-block1;
        }
        else
        {
                statement-block 2;
        }
}
else
{
        statement-block 3;
}
```

Any levels of nested if-else is possible.

if-else-if ladder: Control Statements (Cont....)

```
if(condition1)
{
	//statements
}
else if(condition2)
{
	//statements
}
else if(condition3)
{
	//statements
}
else
{
	//statements
}
```



If-Else-If Ladder

***Dangling else problem

Difference

```c
#include <stdio.h>
int main() {
    int marks = 85;

    if (marks >= 90) {
        printf("Grade A\n");
    }
    if (marks >= 75) {
        printf("Grade B\n");
    }
    if (marks >= 50) {
        printf("Grade C\n");
    } else {
        printf("Fail\n");
    }

    return 0;
}
```

```c
#include <stdio.h>
int main() {
    int marks = 85;

    if (marks >= 90) {
        printf("Grade A\n");
    } else if (marks >= 75) {
        printf("Grade B\n");
    } else if (marks >= 50) {
        printf("Grade C\n");
    } else {
        printf("Fail\n");
    }

    return 0;
}
```

Quick exercise for class

- Input a 3-digit number. Check whether it is a palindrome.
- Write a program to check whether a given year is a leap year or not.
- Input a character and check whether it is a vowel or a consonant.

```c
#include <stdio.h>

int main() {
    int num, first, last;

    printf("Enter a 3-digit number: ");
    scanf("%d", &num);

    if(num < 100 || num > 999) {
        printf("Not a 3-digit number!\n");
    } else {
        first = num / 100;    // first digit
        last = num % 10;      // last digit

        if(first == last) {
            printf("%d is a Palindrome number.\n", num);
        } else {
            printf("%d is NOT a Palindrome number.\n", num);
        }
    }
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);

    if((year % 400 == 0) || (year % 4 == 0 && year % 100 != 0)) {
        printf("%d is a Leap Year.\n", year);
    } else {
        printf("%d is NOT a Leap Year.\n", year);
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <ctype.h>   // for isalpha()

int main() {
   char ch;

   printf("Enter a character: ");
   scanf("%c", &ch);

   if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u' ||
     ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U') {
      printf("%c is a Vowel.\n", ch);
   }
   else if(isalpha(ch)) {
      printf("%c is a Consonant.\n", ch);
   }
   else {
      printf("%c is not an alphabet.\n", ch);
   }
   return 0;
}
```

```c
#include <stdio.h>

int main() {
   char ch;

   printf("Enter a character: ");
   scanf("%c", &ch);

   // check if vowel
   if(ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u' ||
     ch=='A' || ch=='E' || ch=='I' || ch=='O' || ch=='U') {
      printf("%c is a Vowel.\n", ch);
   }
   // check if alphabet using ASCII ranges
   else if((ch >= 65 && ch <= 90) || (ch >= 97 && ch <= 122)) {
      printf("%c is a Consonant.\n", ch);
   }
   else {
      printf("%c is not an alphabet.\n", ch);
   }
   return 0;
}
```

What is ASCII?

- **ASCII** = *American Standard Code for Information Interchange*.
- It's a character encoding standard where **each character is represented by a number** (0–127).
- For example:
  - `'A'` → 65
  - `'a'` → 97
  - `'0'` → 48
  - space `' '` → 32

So, in C, when you type a character, under the hood it's actually stored as its ASCII integer value.

ASCII Table (0–127)

Decimal	Char	Description	Decimal	Char	Description
0	NUL	Null char	32	(space)	Space
9	TAB	Horizontal Tab	48–57	0–9	Digits
10	LF	Line Feed (newline)	65–90	A–Z	Uppercase letters
13	CR	Carriage Return	97–122	a–z	Lowercase letters

```c
#include <stdio.h>

int main() {
    char ch;
    printf("Enter a character: ");
    scanf("%c", &ch);

    printf("The ASCII value of %c is %d\n", ch, ch);

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    int ascii;
    printf("Enter an ASCII value (0-127): ");
    scanf("%d", &ascii);

    printf("The character for ASCII %d is %c\n", ascii, ascii);

    return 0;
}
```

Some questions

- If ASCII is an integer, then how can it be stored in a char (1 byte)?
- If the binary for 65 (01000001) represents both the integer 65 and the character ‘A’, how does the system distinguish between them?

Dangling Else Problem

```
#include <stdio.h>

int main() {
    int x = -5, y = 10;

    if (x > 0)
        if (y > 0)
            printf("Both positive\n");
    else
        printf("x is not positive\n");  // Dangling else confusion

    return 0;
}
```

```
#include <stdio.h>

int main() {
    int x = -5, y = 10;

    if (x > 0) {
        if (y > 0)
            printf("Both positive\n");
    }
    else
        printf("x is not positive\n");  // Dangling else confusion

    return 0;
}
```

In C, an else always binds to the nearest unmatched if. To avoid confusion, use braces {}.

Loops in C

Loop is used to execute the block of code several times according to the condition given in the loop.

while

for

do-while

while loop

Why do we need a loop?
I want to print “hi” 5 times….

```
main ()
{
        printf (“hi ”);
        printf (“hi ”);
        printf (“hi ”);
        printf (“hi ”);
        printf (“hi ”);

}
```

Syntax

```
while(condition) {
  statement(s);
}
```

A smart solution could be

```
main ()
{
        int i = 0; // initialization
        while (i < 5)
        {
                printf (“hi ”);
                i++;  // increment or decrement
        }

}
```

```
main ()
{
        int i = 5; // initialization
        while (i > 0)
        {
                printf (“hi ”);
                i--;  // increment or decrement
        }

}
```

While loop

```
int main ()
{
  int i = 1, n, sum = 0, mul = 1;
  printf ("Enter the range:");
  scanf("%d", &n);
  printf("\nThe series is = ");
  /*while (i <= n)
  {
    printf("%d ", i);
    sum = sum + i;
    mul = mul * i;
    i  += 2;
  }*/
  while (i <= n)
  {
    if (i%2 == 1){
      printf("%d ", i);
      sum = sum + i;
      mul = mul * i;
      //i++;
    }
    i++; // what if I put it in if ?
  }
  printf("\nThe sum and mul is : %d %d\n", sum, mul);
}
```

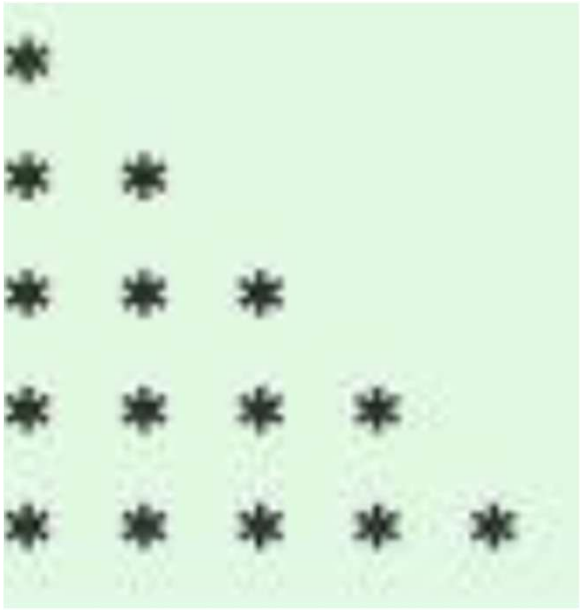
Try to solve it

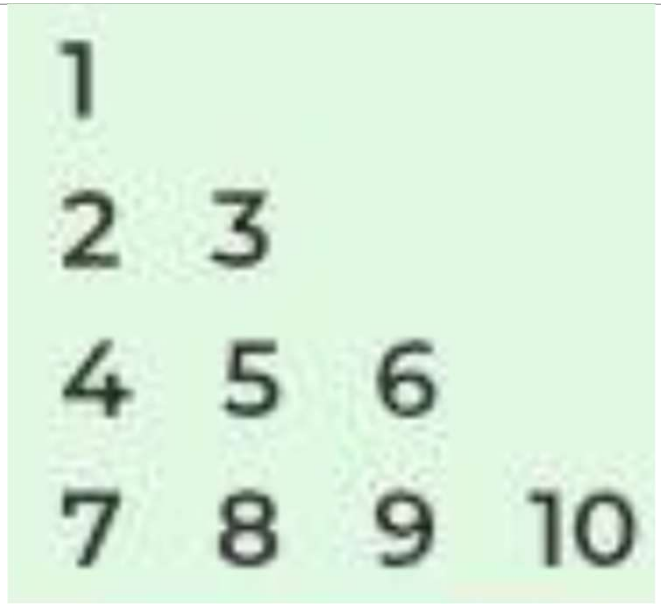
- Some of the digits of a number.

While loop

```
int main()
{
	int num = 0, sum = 0, rem;
	printf("Enter a number greater than 0:");
	scanf("%d", &num);
	while(num != 0)
	{	rem = num % 10;
		printf("%d ", rem);
		num = num / 10;
		sum = sum + rem;
	}
	printf("The sum of digits is : %d\n", sum);
}
```

Print the following two patterns

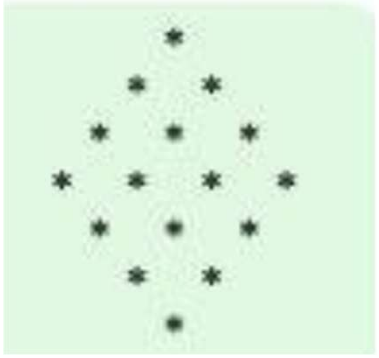


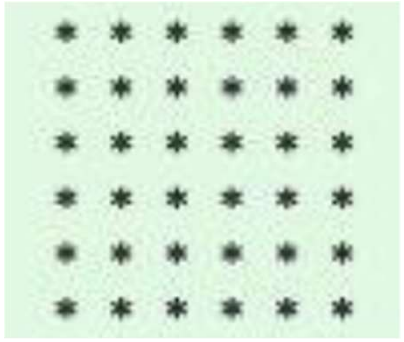


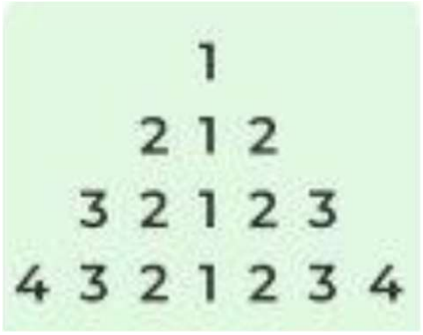
While loop

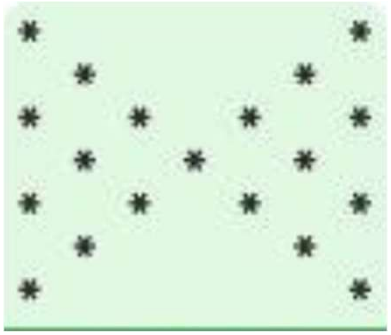
```
void main()
{
	int i=0, j=0, n, c=1;
	printf("Enter the number of lines:");
	scanf("%d", &n);
	while(i < n)
	{
		while (j <= i)
		{
			printf("* ");
			//printf("%d ", c);
			//c++;
			j++;
		}
		printf("\n");
		j=0;
		i++;
	}
}
```

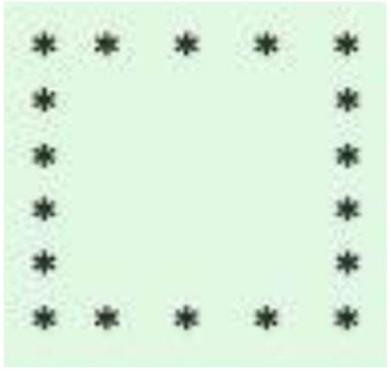
Homework (Line number is user input)











Quick exercise for class

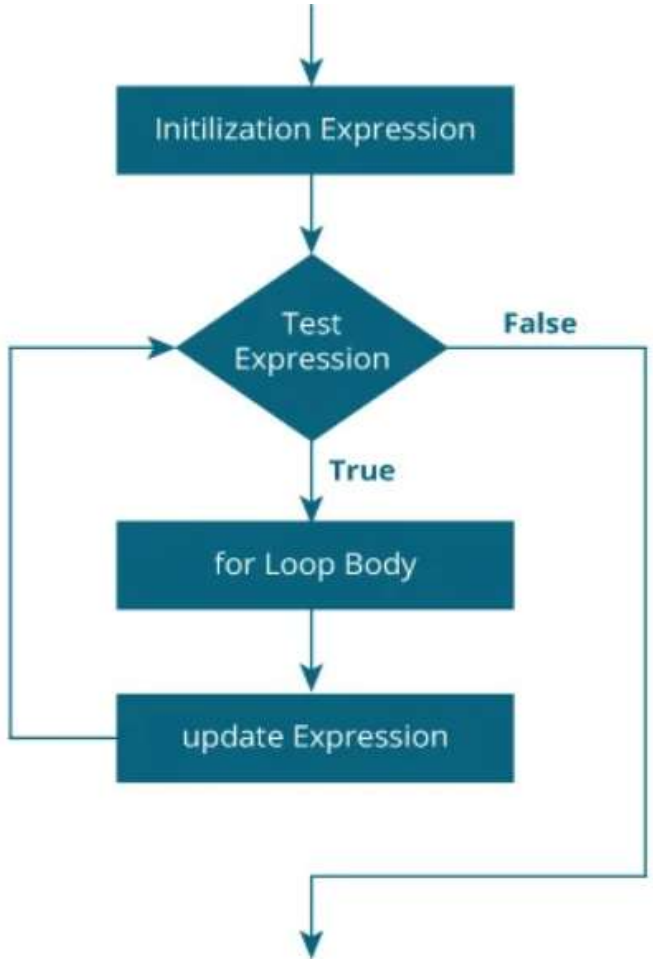
- Write a C program that takes an integer input from the user and reverses its digits using a while loop. The program should repeatedly extract the last digit of the number, build the reversed number, and finally display it.

```c
#include <stdio.h>

int main() {
   int num, reversed = 0, remainder;

   printf("Enter a number: ");
   scanf("%d", &num);

   while (num != 0) {
      remainder = num % 10;            // get last digit
      reversed = reversed * 10 + remainder; // build reversed number
      num = num / 10;                  // remove last digit
   }

   printf("Reversed number = %d\n", reversed);

   return 0;
}
```

for loop

Syntax:

```
for (initialization; test Expression; increment/decrement)
{
   // statements inside the body of loop
}
```



for

```
int main()
{
	int i;
	for (i = 1; i< 10; i++)  // design loop differently, remove each and all ;
	{
		printf("%d\n", i);
	}
	printf ("\n");
}
```

```
void main()
  {
    // declare the local variables
    int i, j, rows, k, m = 0;
    printf (" Enter a number to define the rows: \n");
    scanf ("%d", &rows);
    printf("\n");
    for ( i = rows; i >= 1; i--)
    {
      for ( j = 1; j <= m; j++)
      {
        printf ("  "); // print the space
      }
      for ( k = 1; k <= (2 * i - 1); k++)
      {
        printf ("* "); // print the Star
      }
      m++;
      printf ("\n");
    }
  }
```

Back to control statements: Switch case

```
switch (expression)
{
   case constant1:
     // statements
     break;

   case constant2:
     // statements
     break;
   .
   .
   .
   default:
     // default statements
}
```

- The switch statement allows us to execute one code block among many alternatives.
- You can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.

Switch code

What is the output?

How is it different from if-else?

```
void main()
{
        int day = 0;
        printf("Enter the day in integer\n");
        scanf("%d", & day);
        switch (day)    {
                case 1:
                        printf("Sunday\n");
                        printf(":)\n");
                case 2:
                        printf("Monday\n");
                        printf(":'(\n");
                case 3:
                        printf("Tuesday\n");
                        printf(":(\n");
                case 4:
                        printf("Wednesday\n");
                        printf(":(\n");
                case 5:
                        printf("Thursday\n");
                        printf(":|\n");
                case 6:
                        printf("Friday\n");
                        printf(":D\n");
                case 7:
                        printf("Saturday\n");
                        printf(":D\n");
                default:
                        printf("Invalid input!!!");
        }
}
```

Default can be placed anywhere in the sequence

```
void main() {
        int day = 0;
        printf("Enter the day in integer\n");
        scanf("%d", & day);
        switch (day)        {
                case 1:
                        printf("Sunday\n");
                        printf(":)\n");
                        break;
                case 2:
                        printf("Monday\n");
                        printf(":'(\n");
                        break;
                case 3:
                        printf("Tuesday\n");
                        printf(":(\n");
                        break;
                case 4:
                        printf("Wednesday\n");
                        printf(":(\n");
                        break;
                case 5:
                        printf("Thursday\n");
                        printf(":|\n");
                        break;
                case 6:
                        printf("Friday\n");
                        printf(":D\n");
                        break;
                case 7:
                        printf("Saturday\n");
                        printf(":D\n");
                        break;
                default:
                        printf(“Invalid input!!!");
        }
}
```

The *break* is a keyword in *C.*

```
int main() {

   switch(operation)
   {
      case '+':
         printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
         break;

      case '-':
         printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
         break;

      case '*':
         printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
         break;

      case '/':
         printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
         break;

      // operator doesn't match any case constant +, -, *, /
      default:
         printf("Error! operator is not correct");
   }

   return 0;
}
```

```
char operation;
   double n1, n2;

   printf("Enter an operator (+, -, *, /): ");
   scanf("%c", &operation);
   printf("Enter two operands: ");
   scanf("%lf %lf",&n1, &n2);
```

```
void main()
{
	char day;

	printf("Enter the day in integer\n");
	scanf("%c", & day);
	switch (day)
	{
		case 'A':
			printf("Sunday\n");
			break;
		case 'B':
			printf("Monday\n");
			break;
		case 'C':
			printf("Tuesday\n");
			break;
		case 'D':
			printf("Wednesday\n");
			break;
		case 'E':
			printf("Thursday\n");
			break;
		case 'F':
			printf("Friday\n");
			break;
		case 'a':
			printf("Saturday\n");
			break;
		default:
			printf("Invalid input!!!");
	}
}
```

Can I make it ‘A’ ?

```
void main()
{
	int day = 0;

	printf("Enter the day in integer\n");
	scanf("%d", & day);

	switch (day)
	{
		case day:
			printf("Sunday\n");
			break;
		case day:
			printf("Monday\n");
			break;
		case day:
			printf("Tuesday\n");
			break;
		case day:
			printf("Wednesday\n");
			break;
		case day:
			printf("Thursday\n");
			break;
		case day:
			printf("Friday\n");
			break;
		case day:
			printf("Saturday\n");
			break;
		default:
			printf("Invalid input!!!");
	}
}
```

What about placing a variable here?

```c
int main()
{
	int day = 6;

	switch (day) {
	  case 6:
	    printf("Today is Saturday");
	    break;
	  case 7:
	    printf("Today is Sunday");
	    break;
	  default:
	    printf("Looking forward to the Weekend");
	}
}
```

Is the sequence important?

What is the output?

```
void main()
{

	int day = 2;
	switch () {
	case 1:
	printf("Monday");

	case 2:
	printf("Sunday");

	}
}
```

What is the output?

```
#include <stdio.h>
void main()
{
   int a=10;
   switch(a){
      case 5+5:
         printf("Hello\n");
      default:
         printf("OK\n");
   }
}
```

What is the output?

```
int main() {
    int a = 7, b = 3, c = 2;
    // Complicated expression inside switch
    switch ((a * b) % 5 + (c << 2)) {
        case 0:
            printf("Case 0: Exact match\n");
            break;
        case 4:
            printf("Case 4: (a*b) mod 5 + (c<<1) = 4\n");
            break;
        case 6:
            printf("Case 6: Got 6\n");
            break;
        case 8:
        case 9:
            printf("Case 8 or 9: Multiple matches\n");
            break;
        default:
            printf("Default: No exact match\n");
    }
    return 0;
}
```

break vs. continue (both are keywords)

```
int main()
{
	int i, j;

	for (i = 0; i < 5; i++)
	{
		for (j = 0; j < 5; j++)
		{
			if(j == 3)
				break;

			printf("%d ", i * j);

		}
		printf("\n");
	}
	return 0;
}
```

```
0 0 0
0 1 2
0 2 4
0 3 6
0 4 8
```

```
int main()
{
	int i, j;

	for (i = 0; i < 5; i++)
	{
		for (j = 0; j < 5; j++)
		{
			if(j == 3)
				continue;

			printf("%d ", i * j);

		}
		printf("\n");
	}
	return 0;
}
```

```
0 0 0 0
0 1 2 4
0 2 4 8
0 3 6 12
0 4 8 16
```

Tell me the outputs

```
int i = 0;

while (i < 10) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
  i++;
}
```

break doesn't belong to if.

It only affects loops (for, while, do-while) and switch.

```
int main()
{
   int i = 0;


   while (i < 10) {

      if (i%2==0)
         continue;

      printf ("%d ", i);
      i++;

   }

   return 0;
}
```

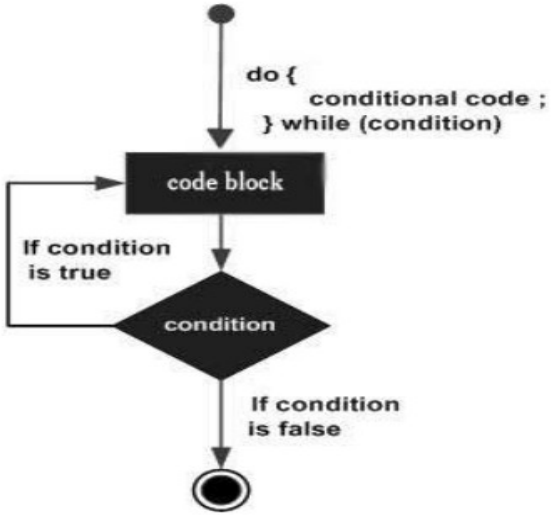
```
int main()
{
   int i = 0;


   while (i < 10) {

      if (i%2==0){
         ++i;
         continue;
      }

      printf ("%d ", i);
      i++;

   }

   return 0;
}
```

Back to loop: do-while

Syntax

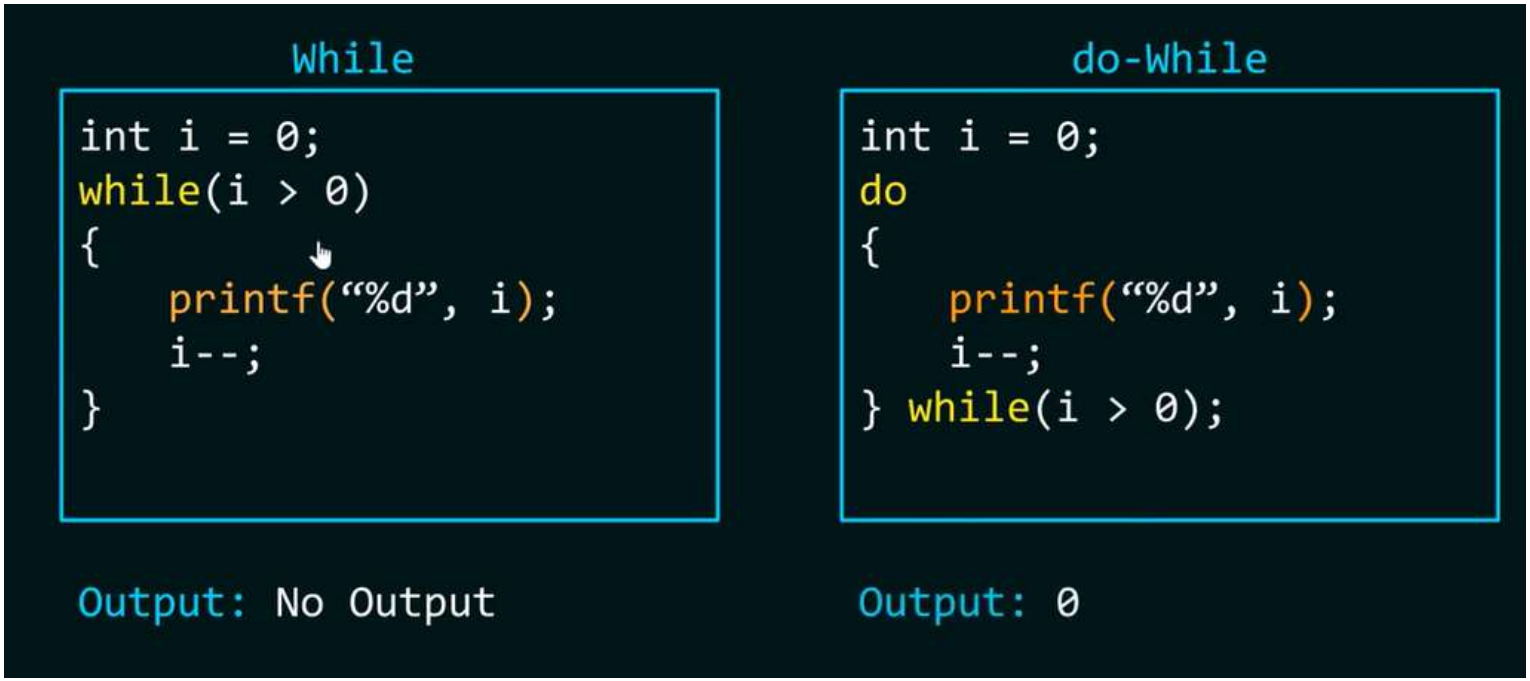
```
do {
        statement(s);
} while( condition );
```

Do not forget




```
#include <stdio.h>

int main ()
{
        int i = 1;
        do{

                printf("hi\n");
                i++;
        }while (i<=5);
}
```

Back to loop: do-while



When should I prefer do-while over while?

Write a program which allows users to enter an integer until a zero value is pressed.

```
int main()
{
   int n;
   printf("Enter an integer: ");
   scanf("%d", &n);
   while (n != 0)
   {
      printf("Enter an integer\n");
      scanf("%d", &n);
   }
   printf("Program ended");
}
```

Same code??

```
int main()
{
   int n;

   do
   {
      printf("Enter an integer\n");
      scanf("%d", &n);
   } while (n != 0);

   printf("Program ended");
}
```

Think about the situation where loop body needs to be executed at least once and then you need to check condition…….

Array

```
void main()
{
    int a[4] = {10, 21, 4 , 0}; // declaring and initializing an array

    a[2] = 18;

    printf ("%d\n", a[0]);
    printf ("%d\n", a[1]);
    printf ("%d\n", a[2]);
    printf ("%d\n", a[3]);
}
```

What if we try to store more Elements than 4?
In GCC, warning with garbage.
Output may change based on Compilers.
C does not give error, so be careful.
Array bounds are not checked.

If we remove the highlighted part, What is the output?

In C, **always stay within the bounds** of the array (`0` to `size-1`). Anything beyond that is undefined behavior.

Tell me the output

```
void main()

{

    float a[4] = {10.3, 21, 4 , 0}; // declaring and initializing an array
    a[2] = 18;

    printf ("%lf\n", a[0]);
    printf ("%lf\n", a[1]);
    printf ("%lf\n", a[2]);
    printf ("%lf\n", a[3]);

}
```

Accessing an Arrays in reverse order using loop

```
void main ()
{
    int n, A[100], i;
    printf("How many numbers to read?");
    scanf("%d", &n);
    for (i = 0; i < n; ++i)
       scanf("%d", &A[i]);
    for (i = n -1; i >= 0; --i)
       printf("%d  ", A[i]);
    printf("\n");
}
```

Why is the loop convenient?

Is the array also changed in the memory?

Output ?

```
int main()
{
   int arr[10], i, temp;

   for (i = 0 ; i<10 ; i++)
      arr[i] = i;

   for(i = 0; i<10/2; i++){
      temp = arr[i];
      arr[i] = arr[10-i-1];
      arr[10-i-1] = temp;
   }

   for (i = 0 ; i<10 ; i++)
      printf("%d ", arr[i]);
   return 0;
}
```

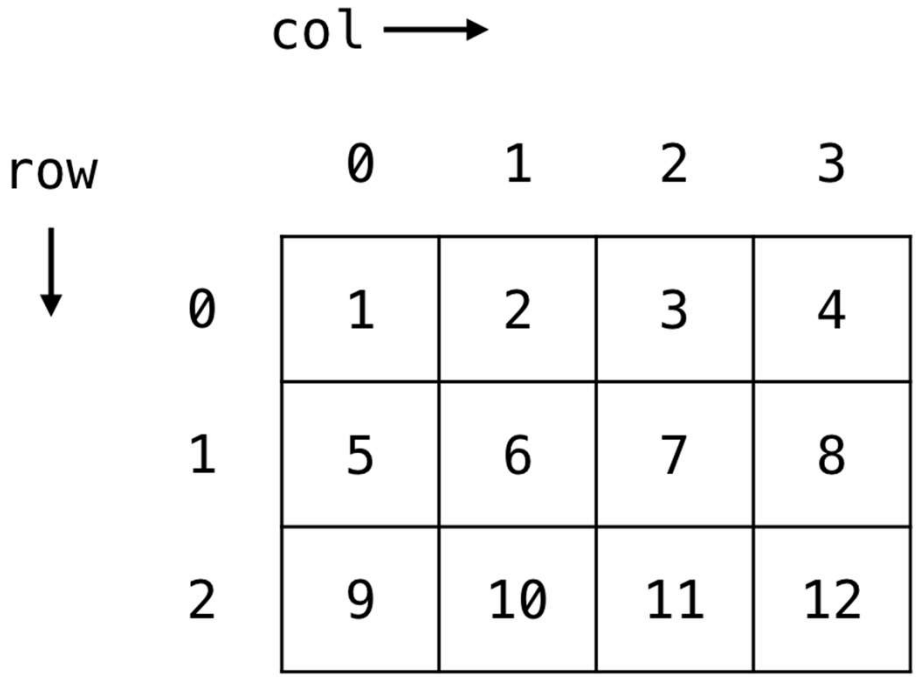
What is the difference with previous reversing?
Building block used is swapping.

Output??

```
void main()
{
  int i, data[10];
  for (i=0; i<10; i++)
        data[i]= i;

  i=0;
  while (i<10)
  {
        printf("Data[%d] = %d\n", i, data[i]);
        i++;
  }
}
```

2D array

```
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```



Methods of initializing a 2D array.

Method 1

```
int arr[2][3] = {
      {1, 2, 3},
      {4, 5, 6}
   };
```

Local arrays with explicit initializer

Uninitialized local (automatic) arrays

Method 2

```
   int arr[2][3] = {1, 2, 3, 4, 5, 6};
```

Method 3

```
   int arr[2][3] = {
   {1},
   {4, 5}
}
```

→ What will happen to the missing elements?

```
int main()
{
   int a [3][3] = {1, 2, 3,
                   4, 5, 6,
                   7, 8, 9}; /// Can I write it in one line.
   int i, j;
  for(i=0 ; i< 3; i++){
      for (j=0; j<3; j++){
         a[i][j] = a[i][j] * 2;
      }
   }
 for(i=0 ; i< 3; i++){
      for (j=0; j<3; j++){
         printf ("%d ", a[i][j]);
      }
      printf("\n");
   }
}
```

In memory view?

Sum of 2 matrices?

Multiplication of two matrices?

Sum of matrices

```
int main() {
  int r, c, a[100][100], b[100][100], sum[100][100], i, j;
  printf("Enter the number of rows (between 1 and 100): ");
  scanf("%d", &r);
  printf("Enter the number of columns (between 1 and 100): ");
  scanf("%d", &c);
  printf("\nEnter elements of 1st matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);
    }
   printf("Enter elements of 2nd matrix:\n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("Enter element b%d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]);
    }
   // adding two matrices
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
    }
  // printing the result
  printf("\nSum of two matrices: \n");
  for (i = 0; i < r; ++i)
    for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
      if (j == c - 1) {
        printf("\n\n");
      }
    }
 }
```

Quick exercises

1. Search an element in the array.

2. Transpose a matrix.

Character array

```
int main()
{
	char a[5];
	int i;
	a[0] = 'H';
	a[1] = 'E';
	a[2] = 'L';
	a[3] = 'L';
	a[4] = 'O';

	for (i = 0; i < 5; ++i)
	{
		printf("%c", a[i]);
	}
	return 0;
}
```

No string data type is directly available

A **string** in C is simply:
👉 **An array of characters terminated by a special character `'\0'` (null character).**

That `'\0'` at the end is what tells the compiler and standard library functions (like `printf`, `strlen`, etc.) where the string ends.

special null character, denoted by '\0'

```
int main()
{
        char a[5];
        int i;
        a[0] = 'H';
        a[1] = 'E';
        a[2] = 'L';
        a[3] = 'L';
        a[4] = 'O';
        a[5] = '\0'; //This should be explicitly inserted to treat it like a string
        for (i = 0; i < 5; ++i)
        {
                printf("%c", a[i]);
        }
        printf("\n%c",a[i]);
        printf("\n%s", a);
        return 0;
}
```

```
int main()
{
	char a[6]="hello";   // make it 6 for safe side.
	char b[] = "Welcome IITJ students"; // This is also possible

	printf("%s\n", a);
	printf("%s\n", b);
	return 0;
}
```

Output?

```
int main()
{
	char b[] = "Welcome IITJ Students";
	int count = 0,i = 0;

	while (b[i] != '\0')
	{
		if(b[i] == 'S' || b[i] == 's')
			count++;
		i++;
	}
	printf("count: %d\n", count);
	return 0;
}
```

String library <string.h>

```
int main()
{
	char b[] = "Welcome IITJ Students";
	char a[50], c[50];
	int count = 0,i = 0;

	count = strlen(b);
	printf("The length is : %d\n", count);
	for (i = 0; i < count; ++i)
	{
		a[i] = b[i];
	}
	a[i] = '\0';
	strcpy(c, b); // (des, src);

	if(strcmp(a, c) == 0)
		printf("Both same\n");

	return 0;
}
```

Can we compare strings like this?

```
int main()
{

	char s1[] = "This is IIT J.";
	char s2[] = "This is IIT J.";

	if (s1 == s2)
		printf("Equal\n");
	else
		printf("Not Equal");
}
```

American Standard Code for Information Interchange (ASCII)

```
int main()
{
	// A = 65
	// a = 97
	// space = 32
	// 0 = 48
	char a[]="Aa 0";
	for (int i = 0; i < 4; ++i)
	{
		printf("%d\n", a[i]);
	}
}
```

String library <string.h>

```
int main()
{

	char s1[] = "This is IIT J.";
	char s2[] = "This ia IIT J.";

	if(strcmp(s1, s2) == 0)
		printf("Equal\n");
	else if(strcmp(s1, s2) < 0)
		printf("s1 < s2\n");
	else if(strcmp(s1, s2) > 0)
		printf("s1 > s2\n");
}
```

```
#include<string.h>
int main()
{

	char src [11] = "1Abcde7890";
	char dest1[50], dest2[5], dest3[50], dest4[50], dest5;

	strcpy(dest1, src);
	printf("dest1: %s\n", dest1);

	strcpy(dest2, src); // working but will avoid for other compilers
	printf("dest2: %s\n", dest2);

	 strncpy(dest3, src, 5);
	dest3[5] = '\0'; /// might not need but it is safer to ensure with \0
	printf("dest3: %s\n", dest3);

	strncpy(dest4, src, 40); // will print up to null
	printf("dest4: %s\n", dest4);

}
```

3D Array

Multidimensional arrays

```c
#include<stdio.h>

int main()
{
            int test [2] [3] [3]={
                                    {{1,2,3},
                                     {4,5,6},
                                     {7,8,9}},

                                    {{1,2,3},
                                     {4,5,6},
                                     {7,8,9}}
                                };
            int i, j, k;

           for(i = 0; i<2; i++)
           {
                for(j=0; j<3; j++)
                {
                    for(k=0; k<3; k++)
                    {
                        printf ("%d ", test[i][j][k]);
                    }
                    printf("\n");
                }
                printf("\n\n");
           }
}
```

A few quick problems on array

Write a program in C to find the sum of all elements of the array.

Write a program in C to count the total number of duplicate elements in an array.

Write a program in C to print all unique elements in an array.

Write a program in C to count the frequency of each element of an array.

Write a program in C to find the maximum and minimum elements in an array.

Write a program in C to find the second largest element in an array.

Write a program in C to find the second smallest element in an array.

Write a program in C to find the sum of the right diagonals of a matrix.

Sorting an array.

Practice problems

1. Take a string reverse it.
2. Take two strings and concatenate them.
3. Count the number of words in a line.
4. Find substring in a string.

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

int main() {
    char str[100];
    int i, len;

    printf("Enter a string (no spaces): ");
    //scanf("%s", str);
    //scanf("%[^\n]s", str);   // reads input until space or newline
    fgets(str, sizeof(str), stdin);

    len = strlen(str);

    printf("Reversed string: ");
    for (i = len - 1; i >= 0; i--) {
        printf("%c", str[i]);
    }
    printf("\n");

    return 0;
}
```

```c
#include <stdio.h>

int main() {
    char str1[100], str2[50];
    int i, j;

    printf("Enter first string: ");
    //scanf("%s", str1);
    fgets(str1, sizeof(str1), stdin);
    printf("Enter second string: ");
    //scanf("%s", str2);
    fgets(str2, sizeof(str2), stdin);
    // Find the end of str1
    for (i = 0; str1[i] != '\0'; i++);

    // Append str2 to str1
    for (j = 0; str2[j] != '\0'; j++) {
        str1[i] = str2[j];
        i++;
    }

    // Add null terminator at the end
    str1[i] = '\0';

    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

```c
#include <stdio.h>
#include <string.h> // for strcat(), strcspn()

int main() {
    char str1[100], str2[50];

    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);

    // Remove newline character from both strings (if present)
    str1[strcspn(str1, "\n")] = '\0';
    str2[strcspn(str2, "\n")] = '\0';

    // Concatenate using strcat
    strcat(str1, str2);

    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

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

int main() {
    char str[200];
    int i, words = 0;

    printf("Enter a sentence: ");
    fgets(str, sizeof(str), stdin);  // read full line with spaces

    // Count words
    for (i = 0; str[i] != '\0'; i++) {
        // if current char is space/newline/tab AND next is not space or end, a new word starts
        if ((str[i] == ' ' || str[i] == '\n' || str[i] == '\t') &&
            (str[i + 1] != ' ' && str[i + 1] != '\0' && str[i + 1] != '\n')) {
            words++;
        }
    }

    // If string is not empty, words = spaces + 1
    if (strlen(str) > 1)
        words++;

    printf("Total words = %d\n", words);

    return 0;
}
```

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[100], sub[50];
    int i, j, found;
    printf("Enter the main string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // remove newline

    printf("Enter the substring: ");
    fgets(sub, sizeof(sub), stdin);
    sub[strcspn(sub, "\n")] = '\0';  // remove newline
    int len1 = strlen(str);
    int len2 = strlen(sub);
    found = 0;
    for (i = 0; i <= len1; i++) {
        for (j = 0; j < len2; j++) {
            if (str[i + j] != sub[j]) {
                break;
            }
        }
        if (j == len2) {  // substring fully matched
            printf("Substring found at index %d\n", i);
            found = 1;
            //break;
        }
    }
    if (!found) {
        printf("Substring not found.\n");
    }
    return 0;
}
```

Function

A function is **a group of statements that together perform a task.**

- **Library Function** e.g. scanf, printf.
- **User defined function** (You need to make for various purpose).

Reuse of codes

How can you create your own function?

- Declarations
- Definition
- Calling the function

A general way of creating your own function

```
#include<stdio.h>

int show(); //Declaration

void main ()
{
        printf("I am from main\n");
        show(); // Calling the function
        printf("I am back to main\n");
}

int show() // Definition
{
        printf("Hi! all\n");
        return 0;
}
```

A function **declaration** tells the compiler about a function's name, return type, and parameters.

The result of function body will be placed from the place of call.

A function **definition** provides the actual body of the function.

Parts of a Function

Return Type:

The return type is the data type of the value that the function returns to the caller. If a function doesn't return any value, its return type is void.

Function Name:

This is the actual name of the function, used to identify and call it in the program.

Parameters / Arguments:

Parameters are placeholders (variables) defined in the function declaration/definition to accept input.

Arguments are the actual values passed to those parameters when the function is called.

Function Body:

The function body is the block of statements enclosed in { } that define what the function does — i.e., the actual logic or task it performs.

A few ways of creating a function

1. No Argument Passed and No Return Value
2. No Arguments Passed but Returns a Value
3. Argument Passed but No Return Value
4. Argument Passed and Returns a Value

```
void sum ();
int main()
{
   printf("From main\n");

   sum();

   printf("Back in main\n");
   return 0;
}
void sum ()
{
   int a, b, c;

   scanf("%d %d", &a, &b);
   c = a + b;
   printf("The sum is %d\n", c);
}
```

```
void sum (int, int);
int main()
{
   int a, b, c;
   printf("From main\n");
   scanf("%d %d", &a, &b);
   sum(a, b);
   printf("Back in main\n");
   return 0;
}

void sum (int x , int y)
{
   int c;
   c = x + y;
   printf("The sum is %d\n", c);
}
```

Arguments

Parameters

```
int sum (int, int);

int main()
{
   int a, b, c;
   printf("From main\n");
   scanf("%d %d", &a, &b);
   c = sum(a, b);
   printf("Back in main\n");
   printf("The sum is %d", c);
   return 0;
}
int sum (int x , int y)
{
   int c;
   c = x + y;
   return c;
}
```

Are these 2 variables same ?

Local vs. Global variable

```
float sum (int, float);
int main()
{
   int a;
   float b, c;
   printf("From main\n");
   scanf("%d %f", &a, &b);
   c = sum(a, b);
   printf("Back in main\n");
   printf("The sum is %f", c);
   return 0;
}
float sum (int x , float y)
{
      return x + y;
}
```

- Can I call a function multiple times?
- Can I use loop to call a function?
- Parameter list can have more than 2 parameters.

Output?

```
void test (void);
int main()
{
   int a =  5;
   test();
}
void test ()
{
   printf("%d", a);
}
```

```
#include <stdio.h>
void test (void);
int a;
int main()
{
   int a =  5;
   printf("%d\n", a);
   test ();

}
void test ()
{
      printf("%d\n", a);
}
```

```
void test (void);
int a = 6;
int main()
{
   int a =  5;
   printf("%d\n", a);
   test ();

}
void test ()
{
   printf("%d\n", a);
}
```

Output?

```
void test (void);
void test1 (void);
int test2 ();
int main()
{
   printf("I am main\n");
   test();
   printf("Back in main\n");
}
void test ()
{
   test1();
   printf("I am test\n");
}
void test1 ()
{
   test2();
   printf("I am test1\n");
}
int test2 ()
{
   printf("I am test2\n");
   return 0;
}
```

```
void test (void);
void test1 (void);
int test2 ();
int main()
{
   printf("I am main\n");
   test();
   printf("Back in main\n");
}
void test ()
{
   test1();
   test2();
   printf("I am test\n");
}
void test1 ()
{
    printf("I am test1\n");
}
int test2 ()
{
   printf("I am test2\n");
   return 0;
}
```

```
void test (void);
void test1 (void);
int test2 ();
int main()
{
   printf("I am main\n");
   test();
   printf("Back in main\n");
}
void test ()
{
   test1();
   printf("I am test\n");
}
void test1 ()
{
   test2();
   printf("I am test1\n");
}
int test2 ()
{
   main();
   printf("I am test2\n");
   return 0;
}
```

Output?

```
void test (void);
int main()
{
   printf("I am main\n");
   test();
   printf("Back in main\n");
}
void test ()
{
   printf("I am test\n");
   test();
}
```

Do not forget!!! It is a recursion.

```
void test (void);
int main()
{
   printf("I am main\n");
   test();
   main();
   printf("Back in main\n");
}
void test ()
{
   printf("I am test\n");
}
```

Quick exercise using a function:
Min-Max
Fact
Sum up to a range

Output?

```c
int checkPrimeNumber(int n);

int main() {

  int n, flag;

  printf("Enter a positive integer: ");
  scanf("%d",&n);
  flag = checkPrimeNumber(n);
  if(flag == 1)
    printf("%d is not a prime number",n);
  else
    printf("%d is a prime number",n);

  return 0;
}

int checkPrimeNumber(int n) {

  if (n == 0 || n == 1)
    return 1;

  int i;

  for(i=2; i <= n/2; ++i) {
    if(n%i == 0)
      return 1;
  }
  return 0;
}
```

Scopes of the variables

```
int main() {
    int n1; // n1 is a local variable to main()
    printf("%d", n2);
}
void func() {
   int n2;  // n2 is a local variable to func()
   printf("%d", n1);
}
```

Local variable

```
int main ()
{
        for (int i = 0; i < 5; ++i)
        {
        }
        printf("%d\n", i);
}
```

```
int main ()
{
        {
                int a = 5;
        }
        printf("%d\n", a);
}
```

```
void display();

int n = 5;  // global variable

int main()
{
    ++n;
    display();
    printf("n = %d", n);
    return 0;
}

void display()
{
    ++n;
    printf("n = %d", n);
}
```

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

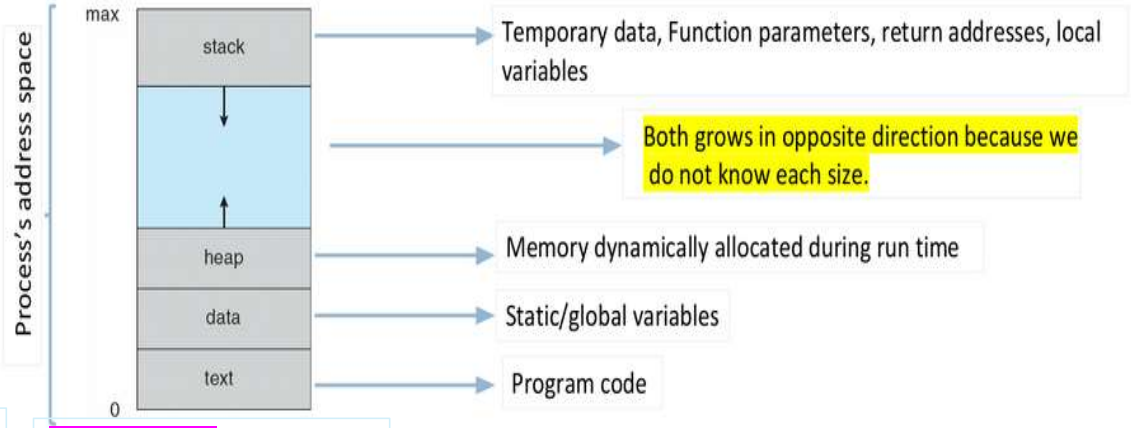
1.automatic
2.external
3.static
4.register

Static storage class

Points to be remember about static variables:

- Initialization of static variables will be done one time only.
- Preserve their value till end of the program
- Store their value in data segment
- Initialize the value to zero

Process (a program in execution) in memory




```
int test(void);
int main()
{
   printf("%d ", test());
   printf("%d ", test());
   return 0;
}
int test()
{
   int var = 5;
   var--;
   return var;
}
```

```
int test(void);
int main()
{
   printf("%d ", test());
   printf("%d ", test());
   return 0;
}
int test()
{
   static int var = 5;
   var--;
   return var;
}
```

```
int test(void);
int main()
{
   printf("%d ", test());
   printf("%d ", test());
   var--;
   printf("%d ", var);
   return 0;
}
int test()
{
   static int var = 5;
   var--;
   return var;
}
```

```
int test(void);
static int var = 5;
int main() {
   printf("%d ", test());
   printf("%d ", test());
   var--;
   printf("%d ", var);
   return 0;
}
int test()
{
   var--;
   return var;
}
```

Optional in some sense

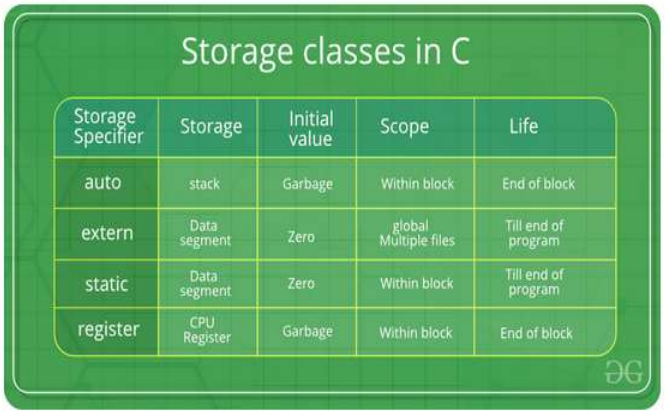
Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Static local vs. Static global

Extern storage class

- "extern" keyword is used to extend the visibility of variables/functions().
- When extern is used with a variable, it's only declared not defined.




```
#include<stdio.h>
int main()
{
        extern int var;
        var = 9;
        printf("%d ", var);
}
```

```
#include<stdio.h>
int main()
{
        extern int var;
        printf("Hi");
}
```

```
#include<stdio.h>
#include"palash.h"
int main()
{
        extern int var;
        printf("%d\n", var);
        return 0;
}
```

```
#include<stdio.h>
#include"palash.h"
extern int sum (int, int);
int main()
{
        int c = 0;
        c = sum (5, 6);
        printf("%d ", c);
        return 0;
}
```

Do you see any differences?

- Standard library header
- User defined header

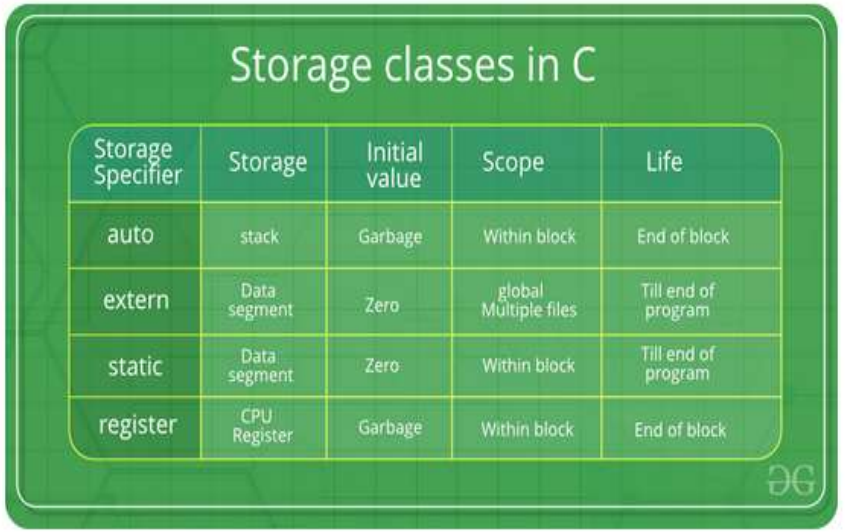
The difference between the two types is in the location where the preprocessor searches for the file to be included in the code.

palash.h

```
int var = 20;
int sum (int a, int b)
{
        return a+b;
}
```

Register Storage Class

```
int main()
{
   register int a = 5, c;
   int b = 6;
   printf("The value is %d %d %d\n", a, b,c);
   printf("The address is %p\n", &a); // It is register
   printf("The address is %p\n", &b); // It  is in RAM
   return 0;
}
```



Output??

Points to remember

- You can get benefit for frequently used variables.
- Stored value is limited by the size of CPU register.

No separate slide for auto
because you have unknowingly used it.

int a is equivalent to auto int a (try it)

Recursion

Recursion is technique in which a function calls itself directly or indirectly.

For example

```
int func ()
{
        func();
}
```

How to write a recursive program?

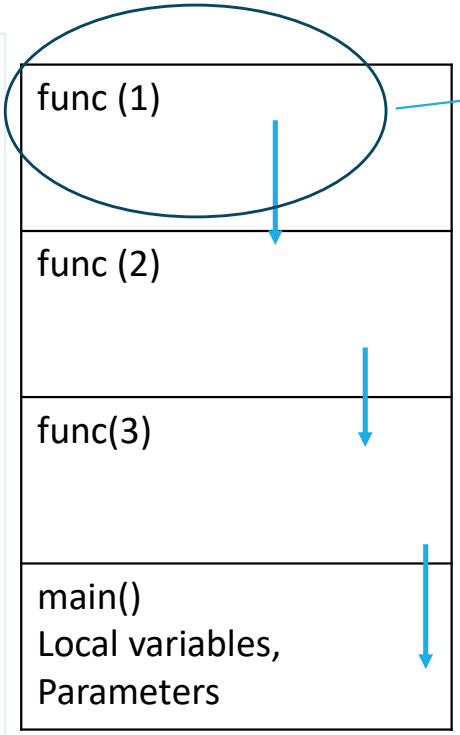
Divide the problem into smaller sub-problems.

Specify the base condition to stop the recursion.

— it reduces the problem size in each call.

A Demonstration of Recursion

```
int func (int);
int main ()
{
        int n = 3;
        printf("%d\n", func(n));
        return 0;
}

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



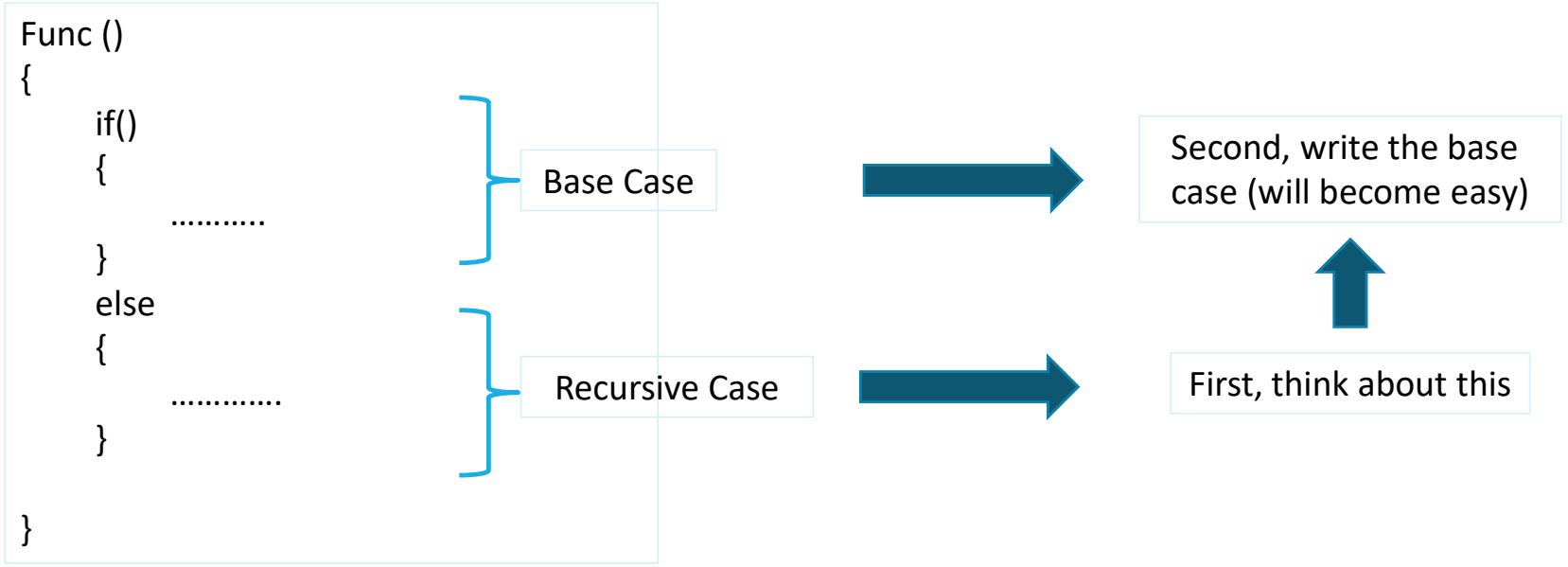
Stack Section of a Process

Base Case

Call Stack Visualization

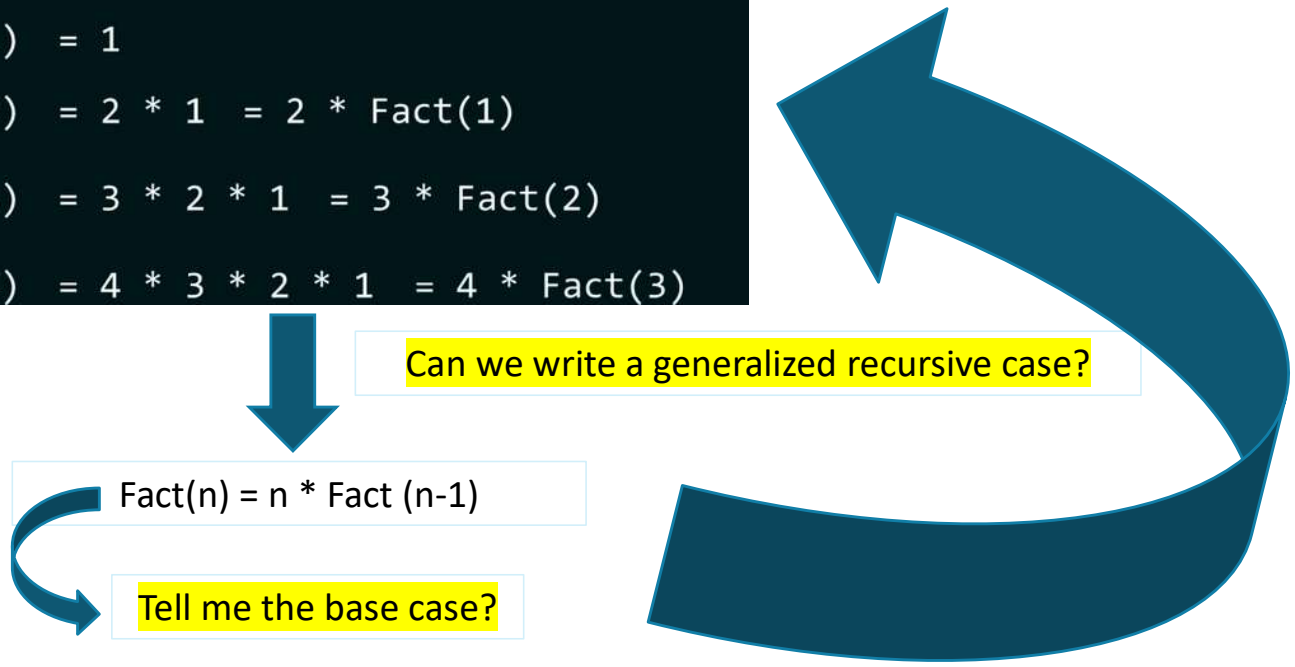
Stack Level	Function Call	Status
3	`func(3)`	Waiting on `func(2)`
2	`func(2)`	Waiting on `func(1)`
1	`func(1)`	Returns `1`
Return	`func(2)` returns `1 + 1 = 2`	
Return	`func(3)` returns `1 + 2 = 3`	

A basic structure of writing a recursive procedure



1. Divide the problem into smaller problems

```
Calculate Fact(4)

Fact(1)  = 1
Fact(2)  = 2 * 1  = 2 * Fact(1)
Fact(3)  = 3 * 2 * 1  = 3 * Fact(2)
Fact(4)  = 4 * 3 * 2 * 1  = 4 * Fact(3)
```



Replacing the code in the basic structure

```
Fact (n)
{
        if(n==1)
        {
                return 1;
        }
        else
        {
        return n * Fact (n-1);
        }

}
```

Base Case

Recursive Case

Stops the recursion, so no recursive call should be here.

So the full code of recursive factorial is…

```
#include <stdio.h>
int fact(int n)
{
   if(n == 1)
      return 1;
   else
      return n*fact(n-1);
}
int main()
{
  int n;
  printf("Enter the number: ");
  scanf("%d", &n);
  printf("Factorial is %d", fact(n));
}
```

Returning From	What it returns	Result
`fact(1)`	`1`	`1`
`fact(2)`	`2 * fact(1)` = `2 * 1`	`2`
`fact(3)`	`3 * fact(2)` = `3 * 2`	`6`
`fact(4)`	`4 * fact(3)` = `4 * 6`	`24`

Solve sum of natural numbers using recursion

```c
#include <stdio.h>

int main() {
    int n, sum = 0;
    printf("Enter n: ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        sum += i;
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

```c
#include <stdio.h>

int sum(int n) {
    if (n == 0)
        return 0;                 // Base case
    else
        return n + sum(n - 1);    // Recursive call
}

int main() {
    int n;
    printf("Enter n: ");
    scanf("%d", &n);

    printf("Sum = %d\n", sum(n));
    return 0;
}
```

Iterative vs. Recursive, which is more efficient?

A few important types of recursion

- Direct Recursion
- Indirect Recursion
- Tail Recursion
- Non Tail/ Head Recursion

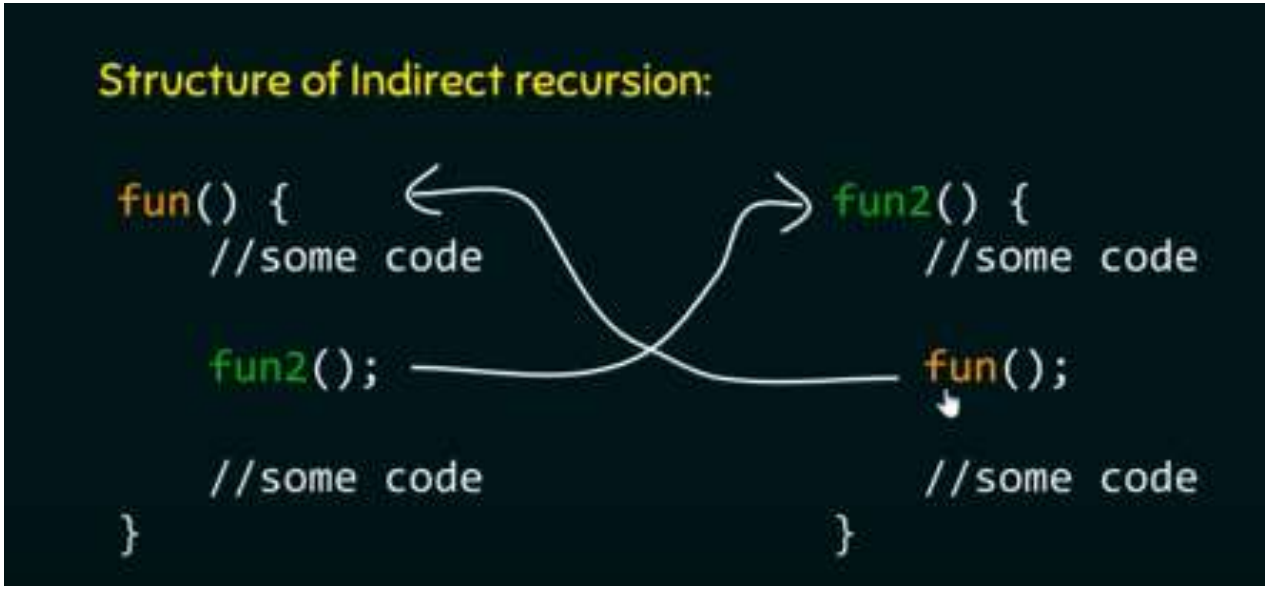
Direct Recursion

A function is called direct recursive if it calls itself.

```
int func ()
{
        some code
        func();
        some code
}
```

Indirect recursion

A function (lets say fun) is called indirect recursive if it calls another function (lets say fun2) and then fun 2 calls fun directly or indirectly.



Introduction to Pointers

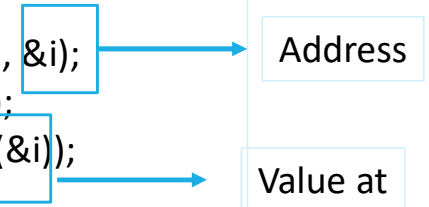
- We can work with memory addresses too. We can use variables called **pointers**.
- A **pointer** is a variable that contains the address of a variable.
- Pointers provide a powerful and flexible method for manipulating data in your programs; but they are difficult to master.

– Close relationship with arrays and strings

Introduction to Pointers

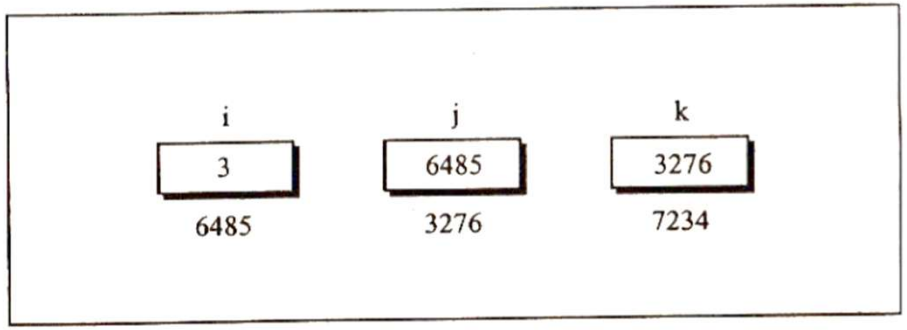
- Pointers allow you to reference a large data structure in a compact way.
- Pointers facilitate sharing data between different parts of a program.
  - Call-by-Reference
- **Dynamic memory allocation:** Pointers make it possible to reserve new memory during program execution.

```c
#include<stdio.h>

void main()
{

	int i = 3;
	printf("Address of i = %p\n", &i);
	printf("Value of i = %d\n", i);
	printf("Value of i = %d\n",*(&i));
}
```



```
#include<stdio.h>

void main()
{

	int i = 3;
	int *j;
	j = &i;
	printf("Address of i = %p\n", &i);
	printf("Address of i = %p\n", j);
	printf("Address of j = %p\n", &j);

	printf("Value of j = %p\n", j);
	printf("Value of i = %d\n", i);
	printf("Value of i = %d\n", *(&i));
	printf("Value of i = %d\n", *j);
}
```

```
#include<stdio.h>
void main()
{
	int i = 3;
	int *j, **k;
	j = &i;
	k = &j;
	printf("Address of i = %p\n", &i);
	printf("Address of i = %p\n", j);
	printf("Address of i = %p\n", *k);
	printf("Address of j = %p\n", &j);
	printf("Address of j = %p\n", k);
	printf("Address of k = %p\n", &k);

	printf("Value of j = %p\n", j);
	printf("Value of k = %p\n", k);
	printf("Value of i = %d\n", i);
	printf("Value of i = %d\n", *(&i));
	printf("Value of i = %d\n", *j);
	printf("Value of i = %d\n", **k);
}
```



```c
#include<stdio.h>

void main()
{
   char c, *cc;
   int i, *ii;
   float a, *aa;
   c = 'A';
   i = 54;
   a = 3.14;
   cc = &c;
   ii = &i;
   aa = &a;
      printf("Address of c = %p\n", cc);
      printf("Address of i = %p\n", ii);
      printf("Address of a = %p\n", aa);
      printf("Value of c = %c\n", *cc);
      printf("Value of i = %d\n", *ii);
      printf("Value of a = %f\n", *aa);
      printf("Address of c = %p\n", cc + 1);
      printf("Address of i = %p\n", ii + 1);
      printf("Address of a = %p\n", aa + 1);
}
```

Pointer arithmetic

```
#include<stdio.h>

void main()
{

	int i = 54;
	float a = 3.14;
	char *ii, *aa;

	ii = (char *)&i;
	aa = (char *)&a;

	printf("Address of ii = %p\n", ii);
	printf("Address of aa = %p\n", aa);

	printf("Value at the address contained in ii = %d\n", *ii);
	printf("Value at the address contained in aa = %d\n", *aa);
}
```

54 = 00000000 00000000 00000000 00110110

3.14 = 01000000 01001000 11110101 11000011 (**IEEE 754 32-bit format**)

Endian-ness determines the **byte order** of data in memory: **little endian** vs **big endian**

```
#include<stdio.h>
void swapv(int,  int);
void main()
{

        int a = 10;
        int b = 20;

        swapv(a, b);
        printf("a=%d\n",a);
        printf("b=%d\n",b);
}
void swapv(int x, int y)
{
        int t;
        t = x;
        x = y;
        y = t;
        printf("x=%d\n",x);
        printf("y=%d\n",y);
}
```

Call by value

```
#include<stdio.h>
void swapv(int *,  int *);
void main()
{

        int a = 10;
        int b = 20;

        swapv(&a, &b);
        printf("a=%d\n",a);
        printf("b=%d\n",b);
}

void swapv(int *x, int *y)
{
        int t;
        t = *x;
        *x = *y;
        *y = t;
}
```


Call by Reference

```
#include<stdio.h>
int calculate (int, float *, float *);
void main ()
{
	int radius;
	float area, perimeter;
	printf("Enter the radius of a circle\n");
	scanf("%d", &radius);
	calculate(radius, &area, &perimeter);
	printf("Area=%f\n", area);
	printf("Perimeter=%f\n", perimeter);
}

int calculate(int r, float *a, float *p)
{
	*a = 3.14*r*r;
	*p = 2*3.14*r;
}
```

Is not the function returning two values in some sense?

```
#include<stdio.h>

int * func ();

void main ()
{
        int *p;
        p = func();
        printf("The address is %p\n", p);
        printf("The value is %d\n", *p);
}

int * func()
{
        int i = 20;
        return (&i);
}
```


Won’t work as it is out of scope….
static will increase the scope

```
#include<stdio.h>

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

	for (i = 0; i < 5; ++i)
	{
		display (marks[i]); /// call by value
	}
}

void display(int m)
{
	printf("%d\n", m);
}
```

```
#include<stdio.h>

void display (int *);
void main()
{
	int i;
	int marks[] = {1,2,3,4,5};

	for (i = 0; i < 5; ++i)
	{
		display (&marks[i]); /// call by reference
	}
}

void display(int *m)
{
	printf("%d\n", *m);
}
```

```
void main (){
	int i=3, *x;
	float j = 1.5, *y;
	char k = 'c', *z;
	printf("The value of i=%d\n", i);
	printf("The value of j=%f\n", j);
	printf("The value of k=%c\n", k);
	x = &i;
	y = &j;
	z = &k;
	printf("%lu %lu %lu\n", sizeof(i), sizeof(j), sizeof(k));
	printf("The address of i=%p\n", x);
	printf("The address of j=%p\n", y);
	printf("The address of k=%p\n", z);
	x++;
	y++;
	z++;
	printf("The address of i=%p\n", x);
	printf("The address of j=%p\n", y);
	printf("The address of k=%p\n", z);

}
```

Pointer increments

```
#include<stdio.h>

void main ()
{
        int num[] = {1,2,3,4,5}, i = 0;
        while (i < 5)
        {
                printf("%p\n", &num[i]);
                i++;
        }

}
```

Accessing each contiguous mem. Locs. of an array

DECIMAL	HEX	BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

```
#include<stdio.h>
void main ()
{
	int num[] = {1,2,3,4,5}, i = 0, *j;
	j = num; //&num[0]
	while (i < 5)
	{
		printf("%p\n", j);
		printf("%d\n", *j);
		j++;
		i++;
	}
}
```

Accessing the whole array through a single pointer

Array also uses implicit pointers (I did not tell before :D)

Passing an entire array to a function

```
#include<stdio.h>
void display (int *, int);
int main()
{
	int num [] = {1, 2, 3, 4, 5};
	display (&num[0], 5);  /// can be num also
	return 0;
}
void display (int *x, int n)
{
	int i = 0;
	while(i<n)
	{
		printf("Address of each elements %p\n", x);
		printf("Elements = %d\n", *x);
		printf("----------------\n");
		i++;
		x++;
	}
}
```

```
#include<stdio.h>
int main()
{
	int num [] = {1, 2, 3, 4, 5};
	int i = 0;

	while(i < 5)
	{
		printf("Address = %p\n", &num[i]);
		printf("element = %d\n", num[i]);
		printf("%d\n", *(num+i));
		printf("%d\n", *(i+num));
		printf("%d\n", i[num]);
		i++;
		printf("--------------------\n");
	}
}
```

```
#include<stdio.h>
int main()
{
	int s[5][2] = {
			{1, 2},
			{3, 4},
			{5, 6},
			{7, 8},
			{9, 0},
		};

	for (int i = 0; i < 5; ++i)
	{
		printf("%p\n", s[i]);
	}
}
```

```
#include <stdio.h>
int main() {
   int s[5][2];
   printf("s      = %p\n", s);
   printf("s + 1   = %p\n", s + 1);
   printf("s[0]    = %p\n", s[0]);
   printf("s[0] + 1= %p\n", s[0] + 1);
   printf("&s[0]   = %p\n", &s[0]);
   printf("&s[0]+1 = %p\n", &s[0] + 1);
}
```

Expression	Points to	Address change
`s`	row 0 (`s[0][0]`)	base
`s + 1`	row 1 (`s[1][0]`)	+8 bytes
`s + 2`	row 2 (`s[2][0]`)	+16 bytes

Expression	Points to	Address change
`s[i]`	first element of row `i`	base
`s[i] + 1`	second element of row `i`	+4 bytes

Expression	Points to	Address change
`&s[0]`	address of row 0	base
`&s[0] + 1`	address of row 1	+8 bytes

```
#include<stdio.h>
void display (int *, int);
int main()
{
	int s[5][2] = {
			{1, 2},
			{3, 40},
			{5, 6},
			{7, 8},
			{9, 0},
		}, i, j;
	for (i = 0; i < 5; ++i)
	{
		for (j = 0; j < 2; ++j)
		{
			printf("%d ", *(*(s+i)+j));
		}
		printf("\n");

	}
}
```

Explanation: *(*(s+1)+1)

Expression	Value
`s`	address of `s[0]`
`s + 1`	address of `s[1]`
`*(s + 1)`	address of `s[1][0]`
`*(s + 1) + 1`	address of `s[1][1]`
`*(*(s + 1) + 1)`	value `40`

Structures

Structures are collection of heterogeneous data.

Each variable in the structure is known as a **member** of the structure.

You can create a structure by using the struct keyword.

```
struct MyStructure {   // Structure declaration
        int myNum;          // Member (int variable)
        char myLetter;      // Member (char variable)
}; // End the structure with a semicolon
```

```
#include<stdio.h>
struct laptop //Global decleration
	{
		int id;
		char model[20];
		int price;
	};
static int i = 1;
void display (struct laptop);
void main()
{
	struct laptop dell, hp, hp1;

	printf("Enter the details of 1st laptop\n");
	printf("Enter id\n");
	scanf("%d", &dell.id);
	printf("Enter model\n");
	scanf("%s", dell.model);
	printf("Enter price\n");
	scanf("%d", &dell.price);

	printf("Enter the details of 2nd laptop\n");
	printf("Enter id\n");
	scanf("%d", &hp.id);
	printf("Enter model\n");
	scanf("%s", hp.model);
	printf("Enter price\n");
	scanf("%d", &hp.price);
```

```
 hp1 = hp;	//// copy like this is possible
	display(dell);
	display(hp);
	display(hp1);
	printf("Size of hp1 %lu\n", sizeof(hp1));
}
void display (struct laptop temp)
{
	printf("You have entered %d\n", temp.id);
	printf("You have entered %s\n", temp.model);
	printf("You have entered %d\n", temp.price);
	printf("---------------End of the details of laptop %d----------------\n", i);
	i++;
}
```

```
#include<stdio.h>
struct laptop //Global decleration
        {
                int id;
                char model[20];
                int price;
        };
void main() {
        struct laptop companies[5];
        int i = 0;
        for (i = 1; i <= 2; ++i) {
                printf("Enter the details of laptop %d\n", i);
                printf("Enter id\n");
                scanf("%d", &companies[i].id);
                printf("Enter model\n");
                scanf("%s", companies[i].model);
                printf("Enter price\n");
                scanf("%d", &companies[i].price);
        }
   printf("The details are as follows\n");
   for (i = 1; i <= 2; ++i)     {
                printf("The details of laptop %d\n", i);
                printf("Id = %d\n", companies[i].id);
                printf("Model = %s\n", companies[i].model);
                printf("Price = %d\n", companies[i].price);
                printf("--------------------------------\n");
        }
}
```

Array of structure variables

Idea of a Void Pointer

A void pointer is a pointer that has no associated data type with it.

A void pointer can hold address of any type.

Type casting is required to access the stored values.

One use case is dynamic memory allocation.

```
int main()
{
        int n = 10;
        float f = 2.45;
        void *ptr  = &n;
        printf("%d\n", *(int *)ptr); // type casting is must.
        ptr = &f;
        printf("%f\n", *(float *)ptr);
        return 0;
}
```

Introduction to dynamic memory

Memory allocation during compile time is called static memory allocation.

The allocated memory is fixed and can not be increased or decreased during runtime.

```
int main()
{
        int arr[5] = {1, 2, 3, 4, 5};   //mem. allocation is compiled time and fixed
}
```

Either wastage of memory or out of bound may need to abnormal behavior (no bound checks for C).

Solution is dynamic memory allocation.

The process of allocating memory at the time of execution is called dynamic memory allocation.

DM uses the heap section of a process.

Built-in support for DM

malloc()

calloc()

realloc()

free()

malloc()

malloc() is a library function declared in the header file <stdlib.h>

It is used to dynamically allocate a single large block of contiguous memory according to the size specified.

Syntax:

(void *) malloc(size_t size) → You should typecast the return type

Allocates the memory in heap based on the size specified

On success returns a pointer pointing to the first byte (if byte addressable) of the allocated memory, else returns NULL.

size_t is defined in <stdlib.h> as an unsigned int.

It returns void pointer because it only reserves the memory of specified size, without caring about the data to be stored there.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
	int i, n;
	printf("Enter the number of integers:\n");
	scanf("%d", &n);
	int *ptr = (int *) malloc (n *sizeof(int));
	if(ptr == NULL) {
		printf("Memory not available\n");
		exit(1);
	}

	for (i = 0; i < n; ++i)
		scanf("%d", ptr+i); /// ptr[i], i[ptr] will also work
	printf("You have entered\n");
	for (i = 0; i < n; ++i)
		printf("%d ", *(ptr+i));
	printf("\nYou have entered\n");
}
```

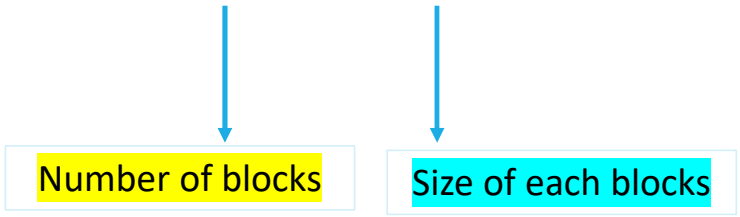
Do I need a & here?

calloc()

calloc() function is used to dynamically allocate multiple blocks of memory.

Syntax:

void * calloc (size_t n, size_t size)



Example

int *ptr = (int *) calloc(10, sizeof(int));

An equivalent malloc call:

int *ptr = (int *) malloc(10*sizeof(int));

Memory allocated by calloc is initialized to zero (garbage in malloc)

Convert the previous code using calloc

Who is faster?

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

int main() {
    int *m = malloc(5 * sizeof(int));
    int *c = calloc(5, sizeof(int));

    for (int i = 0; i < 5; i++)
        printf("malloc[%d] = %d\tcalloc[%d] = %d\n", i, m[i], i, c[i]);

    free(m);
    free(c);
    return 0;
}
```

realloc()

realloc() function is used to change the size of the memory block without losing the old data.

Syntax:

void * realloc(previous pointer, size_t newSize)

On failure, realloc returns NULL.

Moves the contents of the old block to a new block and the old data remains there.

However, you may lose the data when the new size is smaller than the old size.

```
#include <stdio.h>
#include <stdlib.h>
void main(){
        int i;
        int *ptr = (int *) malloc (2 *sizeof(int));
        if(ptr == NULL) {
                printf("Memory not available\n");
                exit(1);
        }
        printf("Enter the two numbers\n");
        for (i = 0; i < 2; ++i)
                scanf("%d", ptr+i); /// ptr[i], i[ptr] will also work
        ptr = (int *)realloc(ptr, 4 *sizeof(int));
        if(ptr == NULL) {
                printf("Memory not available\n");
                exit(1);
        }
        printf("Enter two more numbers\n");
        for (i = 2; i < 4; ++i)
                scanf("%d", ptr+i); /// ptr[i], i[ptr] will also work
        printf("You have entered\n");
        for (i = 0; i < 4; ++i)
                printf("%d ", *(ptr+i));
        printf("\nYou have entered\n");
}
```

free()

free () function is used to release the dynamically allocated memory in heap.

Syntax

void free (ptr)

The memory allocated in heap will not be released automatically after using the memory. The space remains there and can't be used (of course till the program is alive in main memory).

It is a good practice to free up the used part.

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

int * input()
{
	int *ptr, i;
	ptr = (int *) malloc (5*sizeof(int));
	printf("Enter 5 numbers:\n");
	for (i = 0; i < 5; ++i)
		scanf("%d", ptr+i);
	return ptr;
}
void main()
{
	int i, sum = 0;
	int *ptr = input();
	for (i = 0; i < 5; ++i)
		sum += *(ptr+i);
	printf("Sum is: %d\n", sum);
	free(ptr); // releasing the memory at the end
	for (i = 0; i < 5; ++i)
		printf("%d\n", ptr[i]);
	ptr = NULL; // we do not want dangling pointer, so it is a good practice to assign NULL
}
```

File handling

1.Creating a new file – **fopen() with attributes as “a” or “a+” or “w” or “w+”**
2.Opening an existing file – **fopen()**
3.Reading from file – **fscanf() or fgets()**
4.Writing to a file – **fprintf() or fputs()**
5.Moving to a specific location in a file- **fseek(), rewind()**
6.Closing a file – **fclose()**

A few basic operations of file handling

File Pointer a very useful tool: A file pointer is a reference to a particular position in the opened file.

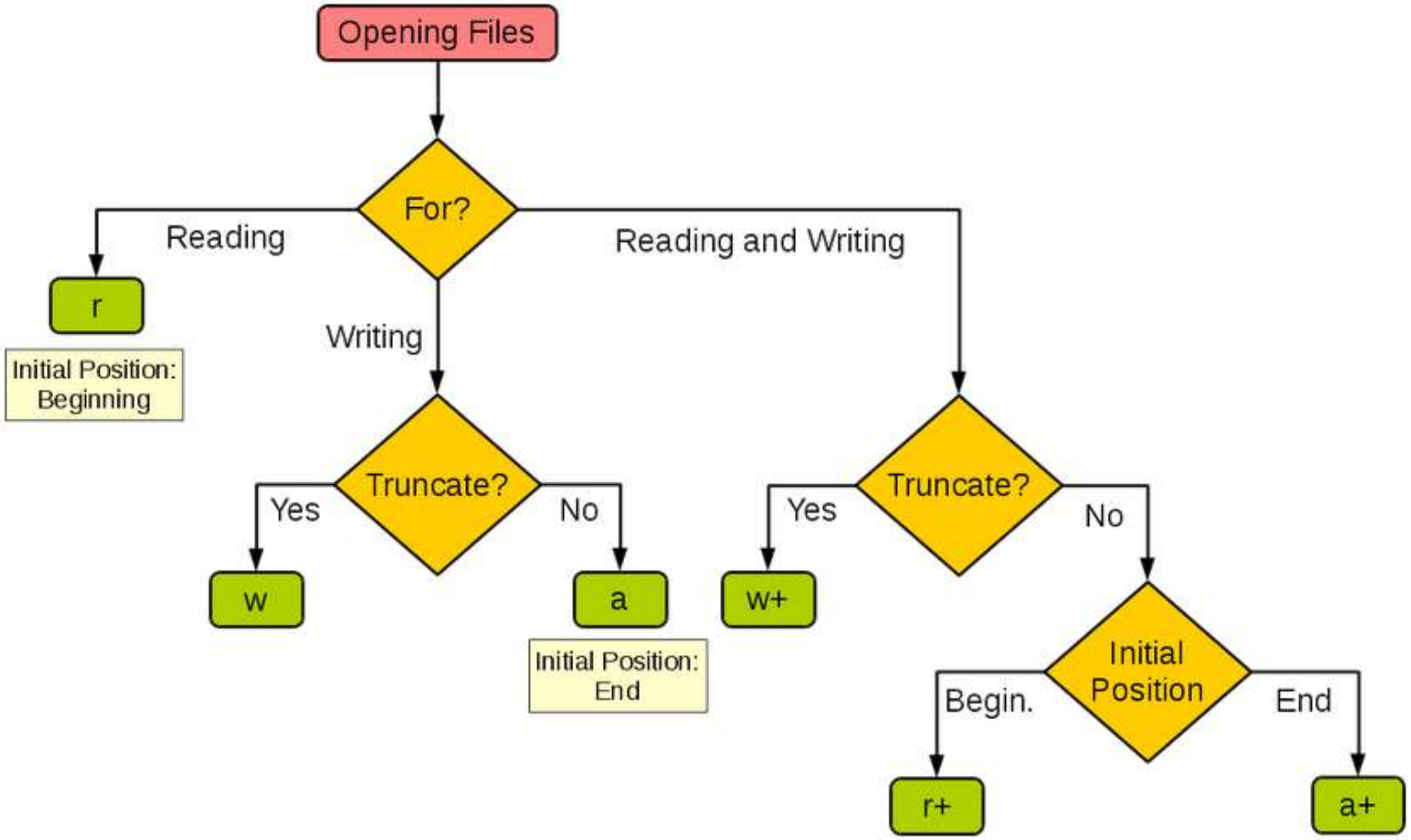
Syntax: FILE * *pointer_name*;

Modes

File opening modes or access modes specify the allowed operations on the file to be opened. A few of them are as follows.

r	Opens a file in read mode -> Only reading possible. Can not create the file if it does not exist.
w	Opens or create a file in write mode -> Only writing is possible. Create the file if it does not exist; otherwise, erase the old content of the file and open a blank file.
a	Opens a file in append mode -> Only writing is possible. Create a file; if it does not exist, otherwise open the file and write from the end of the file. (Do not erase the old content).
r+ , w+	Both r+ and w+ can read and write to a file. However, r+ doesn't delete the content of the file and doesn't create a new file if such file doesn't exist, whereas w+ deletes the content of the file and creates it if it doesn't exist.
a+	Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

Basic Modes


Opening Files
For?
Reading
Writing
Reading and Writing
r
Initial Position: Beginning
Truncate?
Yes
No
w
a
Initial Position: End
Truncate?
Yes
No
w+
Initial Position
Begin.
End
r+
a+

```
#include<stdio.h>

int main()
{

	FILE *fp;
	fp = fopen("demo.txt", "r");
	if(fp == NULL)
		printf("Unable to create the file\n");

	else
		printf("File opened successfully\n");

	fclose(fp);
}
```

```c
#include<stdio.h>
#include<string.h>
int main()
{
	FILE *fp;
	char data[20] = "HI! Students.";
	int len = strlen(data), i;

	fp = fopen("demo.txt", "w");
	if (fp == NULL)
		printf("Unable to open the file\n");

	else{
		for (i = 0; i < len; ++i)
		{
			printf("Writing the char %c\n", data[i]);
			fputc(data[i], fp);
		}
		printf("Data written to the file successfully\n");
	}
	fclose(fp);
	return 0;
}
```

```
#include<stdio.h>
#include<string.h>
int main()
{
	FILE *fp;
	char input[20];

	fp = fopen("demo.txt","a");
	printf("Enter a string to write to the file\n");
	//gets(input);
	fgets(input, 20, stdin); /// Recommended use.
	if (fp == NULL)
		printf("Unable to open the file\n");
	else{
		fputs(input, fp);
		printf("Data is successfully entered\n");
	}
	fclose(fp);
}
```

```
#include<stdio.h>
#include<string.h>
int main()
{
	FILE *fp;
	char name[20];
	int age;

	fp = fopen("demo.txt","a");
	printf("Enter your name\n");
	fgets(name, 20, stdin);
	printf("Enter your age\n");
	scanf("%d", &age);
	if (fp == NULL)
		printf("Unable to open the file\n");
	else{
		fprintf(fp,"%s%d\n", name,age); /// file pointer, format specifier, list of variables
		printf("Data is successfully entered\n");
	}
	fclose(fp);
}
```

```
#include<stdio.h>
#include<string.h>
int main()
{
	FILE *fp;
	char temp_ch;
	fp = fopen("demo.txt","r");

	if (fp == NULL)
		printf("Unable to open the file\n");
	else{
		while(!feof(fp))
		{
			temp_ch = fgetc(fp);
			printf("%c", temp_ch);
		}
	}
	fclose(fp);
}
```

```c
#include <stdio.h>

int main() {
   FILE *fp;
   int temp_ch; // must be int to store EOF

   fp = fopen("demo.txt", "r");
   if (fp == NULL) {
      printf("Unable to open the file\n");
      return 1;
   }

   while ((temp_ch = fgetc(fp)) != EOF) { // read and check EOF
in one step
      printf("%c", temp_ch);
   }

   fclose(fp);
   return 0;
}
```


Solution

Quick Practice

Put Some text in a file and count the number of vowels.

Read the characters and print the ASCIIs.

Check if there are any digit present in the file