Session 04 C

Thursday, 18 September 2025 11:17 AM

Initialization : Exit Controlled Loop:

```
do {

    ______
    ______
    ______
    ______

    Updation;
} while (Condi  );
```

```
int x = 1;      x = 50;
do {
    printf("%d", x);
    x = x+1;
} while( x <= 10);
```

```
int addRange(int minRange, int maxRange)
{
    int sum = 0;
    int x = minRange;
    while (x <= maxRange)
    {
        if (x % 2 == 0)
            sum = sum + x;
        x++;
    }
    return sum;
}
```

minRange $\boxed{5}$
maxRange $\boxed{25}$
x $\boxed{5}$

```
main ( ) {

    addRange ( 5, 25);
}
```

n = 57$\boxed{3}$, sum = 0, prod = 1;

```
        rem  = n%10;
[3]     sum = sum + rem;
[3]     prod = prod * rem;
        n = n/10;
                7
        rem = n%10;
[10]    sum = sum + rem;
[21]    prod = prod * rem;
[5]     n = n/10;
        [5]
        rem = n%10;
[15]    sum = sum + rem;
[105]   prod = prod * rem;
        n = n/10;
```

return 105 - 15

n = $\boxed{573}$ →

3+7+5
sum = 15
n = 15;
n = sum = 6

```
int sum;
(n>10)
{
    sum = 0;
    while ( n != 0 )
    {
        sum = sum + (n%10);
        n = n/10;
    }
    n = sum;
}
return n;
```

1. Memory Layout
2. Storage Classes
3. Session 2 Question
4. Session 4 Question

```
int subtractProductAndSum(int n) {
    int sum = 0;
    int prod = 1;
    while(n!=0){
        sum = sum + (n%10);
        prod = prod *(n%10);
        n=n/10;
    }
    return prod - sum;
}
```

num = 573
$\boxed{573}$
sum $\boxed{0}$
prod $\boxed{1}$

Given an integer num, repeatedly add all its digits until the result has only one digit, and return it.

Example 1:

```
Input: num = 38
Output: 2
Explanation: The process is
38 --> 3 + 8 --> 11
11 --> 1 + 1 --> 2
Since 2 has only one digit, return it.
```

rev = 245

```
x = 542
int rev=0
→ rem = (x%10)
  [illegible]
  x = x/10;
  [54]  rev = 2
  rem:   (x%10);
```

rev ~~02~~ 245
rem ~~24~~ 5
x ~~54~~ 5 0

```
x = 542
int rev = 0;
→ rem = x%10;
→ rev = (rev * 10) + rem;
      = 0 + 2
  x = x/10;
      = 20 + 4
        24*10 + 5
      = 245
```

_ __   9 9

16 Bit , 32 bit

8 [0 0 0 0 0 0 0 0

8 [1 1 1 1 1 1 1 1] → 255

x = 285

rev = ○

0 < rev <= 999

INT_MIN   INT_MAX

Given
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
0 1 2 3 4 5 6 7 8 9 ...

n = 6   5

```
int fib ( int n)
{
        int n0 = 0;
        int n1 = 1;
        int nextTerm;
```

n0	n1	nt
0	1	1
0	1	2

n0 n1 nt

nextterm = n0+n1;

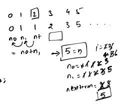
```
if(n<=1)
  return n;
while (i=2 ; i<n ; i++)
  {
  nextterm = n0+n1;
     n0 = n1;
     n1 = nextterm;
  }
return nextterm;
  }
```

0 1 1 2

0 1 2 3 4 5

```
int fib(int n){
    int n0 = 0;
    int n1 = 1;
    int i =2;
    int nextterm ;
    if (n<=1)
    return n;
    while(i<=n){
        nextterm = n0+n1;
        n0 =n1;
        n1 = nextterm;
    }
    return nextterm;
}
```

i++;



In **C programming**, *storage classes* define **the scope, lifetime, visibility, and default initial value** of a variable or function. They tell the compiler *where* the variable will be stored, *how long* it will exist, and *who* can access it.
There are **four main storage classes in C**:

1. Automatic (auto)

- **Keyword:** auto (but usually omitted, since it's the default for local variables).
- **Scope:** Local to the block/function in which it is defined.
- **Lifetime:** Created when the block is entered, destroyed when the block is exited.
- **Default value:** Garbage (undefined).
- **Storage:** Memory (stack).
- **Example:**

```
void func() {
   auto int x = 10;  // same as just "int x = 10;"
   printf("%d", x);
}
```

2. Register

- **Keyword:** register
- **Scope:** Local to the block/function.
- **Lifetime:** Same as auto (till function/block execution).
- **Default value:** Garbage.
- **Storage:** CPU **register** (if available), otherwise RAM.
- **Special Note:** Address (&x) **cannot be taken** because it may not be stored in memory.
- **Example:**

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

3. Static

- **Keyword:** static
- **Scope:** Local to the block if defined inside a function, but retains its value between function calls. If defined globally, its scope is limited to the file.
- **Lifetime:** Entire program execution.
- **Default value:** 0 (for integers, NULL for pointers, etc.).
- **Storage:** Memory (Data segment).
- **Example:**

```
void func() {
   static int count = 0;  // initialized only once
   count++;
   printf("%d\n", count);
}

int main() {
   func(); // 1
   func(); // 2
   func(); // 3
}
```

4. Extern

- **Keyword:** extern
- **Scope:** Global (visible across multiple files).
- **Lifetime:** Entire program execution.
- **Default value:** 0.
- **Storage:** Memory (Data segment).
- **Usage:** Used to declare a global variable in another file or scope.
- **Example:**

```
// file1.c
int count = 10;   // global variable

// file2.c
extern int count; // declaration, no memory allocated
printf("%d", count);
```

🔑 Quick Comparison

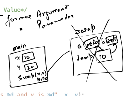
Storage Class	Keyword	Scope	Lifetime	Default Value	Storage Location
Automatic	auto	Local (block)	Till block ends	Garbage	Stack
Register	register	Local (block)	Till block ends	Garbage	CPU Register/RAM
Static	static	Local/Global	Entire program	0	Data segment
Extern	extern	Global (file)	Entire program	0	Data segment

👉Do you want me to also make a **memory layout diagram (stack, heap, data, code segments)** with these storage classes shown visually?

```
/*Swap Two Variables Call by Value*/
#include <stdio.h>
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
int main()
{
    int x = 10;
    int y = 20;
    swap(x, y);
    printf("The value of x is %d and y is %d", x, y);
    return 0;
}
```

formal Argument
" Parameter

actual



main
x 10
y 20
swap(x,y)
10,20
swap
a 10 20
b 20 10
temp 10