Stack Mechanism Discussion with Time Complexity

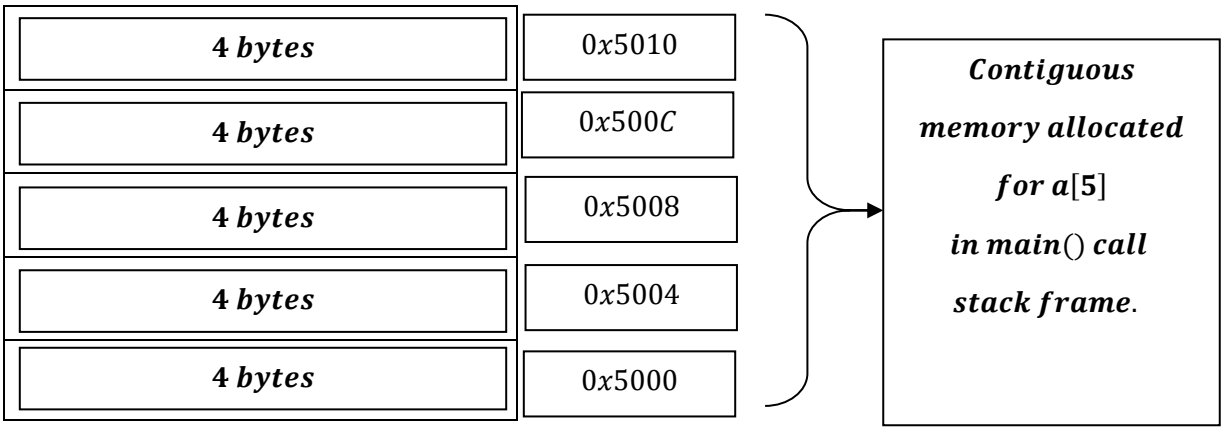
1. *Creation of Stack*

```
int main()
{

int size;
    cout << "Enter the size of the stack" << endl;
    cin >> size;
    int stack[size];
    cout << "Stack is Created" << endl;
.
.
}
```

***User enter size , say*: 5**

***Stack*[*size* = 5] ; *this will allocate contiguous memory in the stack frame when main function is called* [*main function call stack frame*].**



During Static allocation of memory say a[5], the memory is allocated at compile time, memory is reserved before program runs, Cannot change size later[Physical size]though logically we can change. Such during deletion in program but that doesnot change the physical size of array which always remain 5.

What happens here ?

→ ***Here memory allocation occurs during runtime.***

→ ***That is when main() runs, during that time only size of the array is known .***

→ ***But then also it is static Why?***

- → ***It uses an array or array based stack = Static Stack***
- → ***Size cannot grow and shrink later.***
- → ***No new or malloc()***
- → ***No heap memory as heap memory based allocation is dynamic.***

Then when Compile Time – Static ?

```
#include <iostream>
using namespace std;

int size =5;

int main(){

    int arr[size];

    return 0 ;
}
```

→ *Here `size` is declared globally .*

→ *Hence compiler knows before hand the size of the array before execution of main func. during compilation.*

→ *Size cannot grow or shrink later.*

→ *Memory allocated? → When main() runs (runtime stack allocation).*

→ *Compile – time fixed – size array allocated on stack memory.*

Similarly,

```cpp
#include <iostream>
using namespace std;

#define size 5


int main(){


    int arr[size];

    return 0 ;
}
```

#define size 5 is a preprocessor macro.

Before compilation even begins, the preprocessor deals with #define , no memory calculation;

→ The size is known during compilation [Compilation of the program and generating executable file.]

→ Memory layout is fixed at compile time.

What does it mean:

→ During Compilation of the program , Compiler knows the size: 5.

→ Here compiler already sees arr[size] as arr[5].

Because, even though arr inside main func.
the compiler reads and analyzes the entire function
before the program ever runs. The compiler doesnot wait for main function to execute.

→ Now compiler able to know that arr needs 5 integers to be allocated.

→ Calculates required stack space 5×4 bytes $= 20$ bytes.

→ Fixes the stack frame size for main() func.

→ Calculating the required stack space and fixing the size and writing the instruction [sub rsp 20 [Subtract Register Stack Pointer 20 bytes]] to the executable file happens at compile time of the program.

[The above instruction is to reserve fixed 20 bytes memory in main stack frame i.e. for the array .]

→ Hence here arr[size] as arr[5] is known as compile − time constant.

→ The array is allocated on the stack memory when main()runs.

How?

During run time it executes the instruction written to the executable file i.e. $\left[\text{sub rsp 20}\left[\begin{array}{c}\text{Subtract}\\ \text{Register Stack Pointer}\\ \text{20 bytes}\end{array}\right]\right]$ ***and allocates the memory.***

So the classification is:

- ***Size determination → during Compilation time – time or Preprocessing time.***
- ***Allocation moment → Runtime (when function starts)***
- ***Memory type → Stack memory***
- ***Resizable → No***

[More Clarification at Page 8 – 10 of this PDF]

When it is compile time constant?

Ans: ***when it is written:***

#define

const

constexpr

i.e. defined as a constant .

$\#define\ size\ 5$

$const\ int\ size = 5;$

$constexpr\ int\ size = 5;$

The below expressions are all compile time contant :

```cpp
#include <iostream>
using namespace std;

#define size 5


int main(){


    int arr[size];

    return 0 ;
}
```

```cpp
#include <iostream>
using namespace std;

const int size = 5;

int main()
{
    int arr[size];
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
    const int size = 5;
    int arr[size];
}
```

```cpp
#include <iostream>
using namespace std;

constexpr int size = 5;

int main()
{
    int arr[size];
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
    constexpr int size = 5;
    int arr[size];
}
```

Also, the rule of compile time constant is applied for every user defined function.

When we say: "The array size is a compile – time constant" we mean: "The compiler knows the exact size of the array during compilation, before the program runs."

When we say ``Not Compile Time Constant?``

- $int\ size\ =\ 5;$
- $const\ int\ size\ =\ some\_runtime\_value;$

→ *It does NOT matter whether it is inside or outside the function.*

→ *If the value depends on runtime data or a non − constant variable:*
→ *It is NOT a compile − time constant.*

Eg: Not a Compile time constant.

```
int x = 5;

const int size = x;
```

This is more understable with 2 examples:

Eg : −1:

```
#include <iostream>
using namespace std;

int size = 5;

int main()
{
    int arr[size];
    return 0;
}
```

The compiler may treat it as :

→ ***The size is known during compilation*** [***Compilation of the program and generating executable file.***]

→ ***Memory layout is fixed at compile time.***

What does it mean:

→ ***During Compilation of the program , Compiler knows the size*: 5.**
→ ***Here compiler already sees arr[size] as arr[5].***
Because, even though arr inside main func.
the compiler reads and analyzes the entire function before the program ever runs. The compiler doesnot wait for main function to execute.
→ ***Now compiler able to know that arr needs 5 integers to be allocated.***
→ ***Calculates required stack space*** 5×4 ***bytes*** $= 20$ ***bytes.***
→ ***Fixes the stack frame size for main() func.***
→ ***Calculating the required stack space and fixing the size and writing the instruction*** [***sub rsp 20*** [***Subtract Register Stack Pointer 20 bytes***]] ***to the executable file happens at compile time of the program.***
[***The above instruction is to reserve fixed 20 bytes memory in main stack frame i.e. for the array .***]

→ ***Hence here arr[size] as arr[5] is known as compile − time constant.***

<u>It is not guaranteed by compiler as, it can treat also</u>:

→ *Size known as* 5 *during compilation time.*

→ *Then after analyzing the full code it may generate instruction* : *execute size* = 5 *and load it to register, thereafter*

, multiply by sizeof(int) and then allocate stack frame .

to run the instruction at run − time.

→ *Therefore it doesn't fix the stack frame size for main()func.*

during compile time.

→ *That is during Compile time its unknown that what to multiply with size of int.*

→ *During run − time it takes the instruction and then multiply* $5 \times sizeof(int) i.e. 5 \times 4$ *bytes , fixes the size and then allocates memory for main func. in the stack.*

This is more evident when we write code as shown in example 2:

```
#include <iostream>
using namespace std;

int size = 5;

int main()
{
    size=10;
    int arr[size];
    return 0;
}
```

→ *Size known as* 5 *during compilation time.*

→ *Then after analyzing the full code it may generate instruction* : *execute size* = 5 *and load it to register, thereafter*

, multiply by sizeof(int) and then allocate stack frame .

to run the instruction at run − time.

→ Therefore it doesn't fix the stack frame size for main()func. during compile time.

→ That is during Compile time its unknown that what to multiply with size of int. [*sizeof(int)is known to the compiler, but which size will multiply with sizeof(int)is unknown.*]

The compiler:

- ***Reads the entire function.***
- ***Sees*** *size* = **10**;
- ***Sees*** *int arr*[*size*];
- ***Knows*** *size* ***is a modifiable variable.***
- ***Therefore, it cannot treat it as a constant expression.***
- ***So it decides***: ***stack space must be calculated at runtime.***

→ During run − time of main ()func moves the value **10** *to the register , where value* **5** *is stored and modifies or replace or overwrite it with* **10** *, and then it is loaded to the register for further operation , now multiply* **10** × *sizeof(int) i. e.* **10 × 4** *bytes fixes the size of the main stack frame , and then allocates memory for main func. in the stack.*

It is a static data structure(because its capacity is fixed once created), but the memory allocation itself is dynamic stack allocation as it happens during run time. True static memory allocation only happens when the exact byte count is baked (*generated instruction*)into the executable blueprint at compile time.

"True static memory allocation" only happens when the compiler knows the exact number of bytes before the program ever runs.

- **Think of `int arr[5];` or `#define size 5`.**
- **The compiler does the math immediately during compilation (5 integers × 4 bytes = 20 bytes).**
- **It permanently stamps the instruction `sub rsp, 20` directly into the `.exe` file.**
- **There is no calculation happening at runtime; the CPU just blindly shifts the stack pointer by that hardcoded 20 bytes.**

What is contiguous memory allocation ?

Contiguous means All elements are stored in consecutive memory blocks with no gaps between them.

Time Complexity

```
int size;
    cout << "Enter the size of the stack" << endl;
    cin >> size;
    int stack[size];
    cout << "Stack is Created" << endl;
```

1. $Declaration\ of\ int\ size \rightarrow O(1)$

2. $Printing\ ``Enter\ the\ size\ of\ the\ stack`` \rightarrow O(1)$

3. $Taking\ input\ `size`\ from\ the\ user. \rightarrow O(1)$

4. $Allocation\ of\ contiguous\ memory\ in\ the\ main\ function\ stack\ according\ to\ size\ input\ by\ user\ i.e.\ int\ stack[size]\ takes\ constant\ `c`\ time\ . \rightarrow O(1).$

$Total\ Time\ Complexity : O(1) + O(1) + O(1) + O(1) = O(1).$