Module 1

(Part 1)

Name, Scope, and Binding

- A name is a mnemonic character string used to represent something
- Names in most languages are identifiers(alphanumeric tokens)
- Names allow us to refer to variables, constants, operations, types, and so on using symbolic identifiers rather than low-level concepts like addresses.
- A binding is an association between two things, such as a name and the thing it names
- The scope of a binding is the part of the program (textually)in which the binding is active

Binding

Binding time is the time at which a binding is created or, more generally, the time at which any implementation decision is made.

There are many different times at which decisions may be bound:

Language design time

language semantics are chosen when the language is designed.

E.g. the control flow constructs, the set of fundamental (primitive) types,

Language implementation time:

precision (number of bits) of the fundamental

types, the coupling of I/O to the operating system's notion of files,

the organization and maximum sizes of stack and heap,

and the handling of run-time exceptions such as arithmetic overflow.

Binding

Program writing time:

Programmers of course choose algorithms, data structures, and names.

Compile time:

Compilers choose the mapping of high-level constructs to machine code

Link time:

a program is usually not complete until the various modules are joined together by a linker.

When a name in one module refers to an object in another module, the binding between the two is not finalized until link time.

Binding

Load time:

Load time refers to the point at which the operating system loads the program into memory so that it can run.

In primitive operating systems, the choice of machine addresses for objects within the program was not finalized until load time.

Run time:

Bindings of values to variables occur at run time, as do a host of other decisions that vary from language to language.

The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively

Binding Time

- Binding is an association of elements and its property(data types)

int a=10; here var a is binded with int data type

Binding time:

Execution Phase(Runtime/late binding/Dynamic binding)

Translation Phase(Compile time/early binding/static binding)

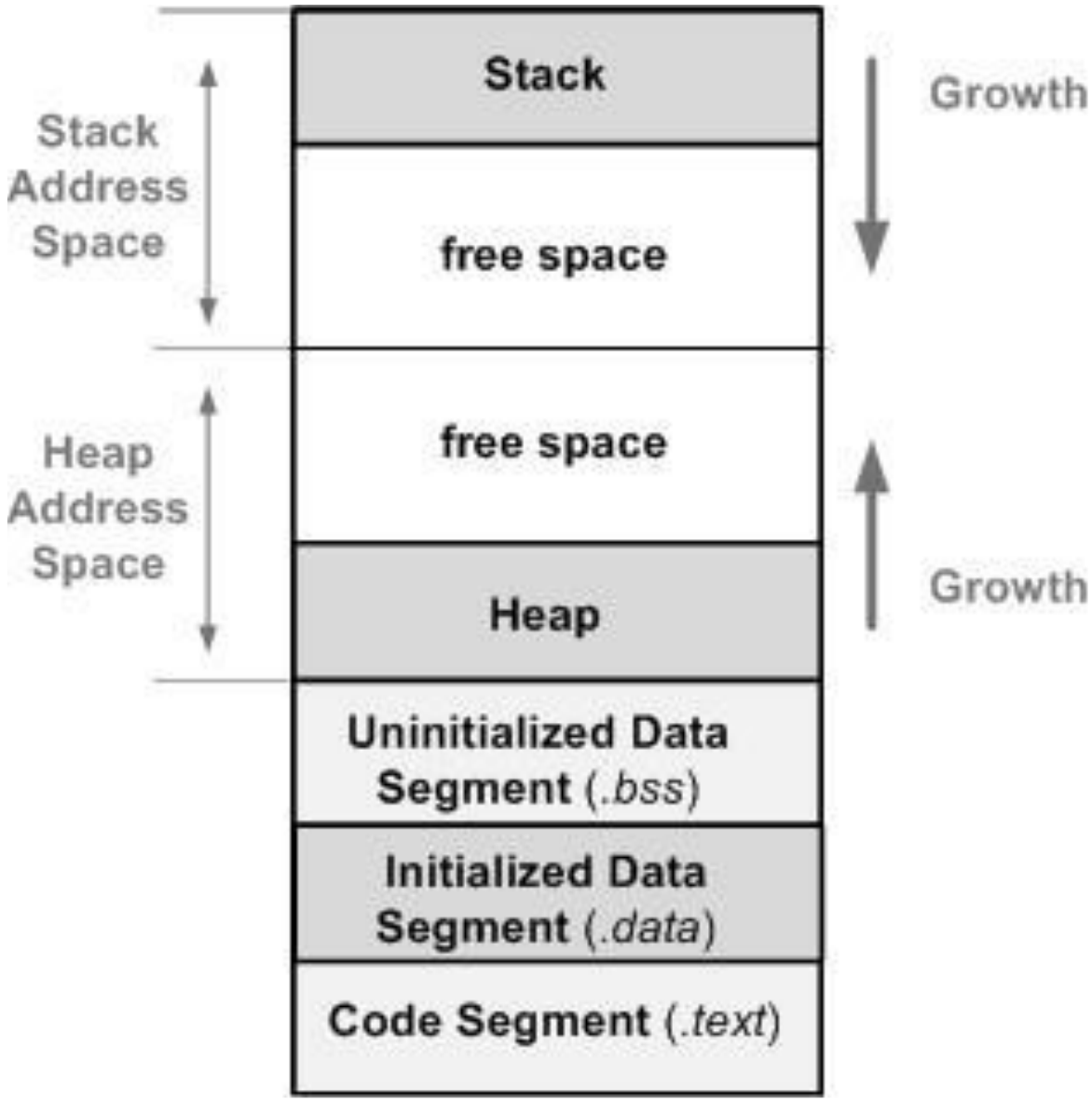
Binding Time

```
#include<stdio.h>
#include<conio.h>
int main()
{ int a=10,b=20;            //compile time
a=sum(&a, &b); // binding at execution time/late binding
}
int sum(int *p, int *q)
{ int *k;
*k=*p+*q;
return *k;
}
```

Lifetime and Storage Management

- **Binding's lifetime** -The period of time between the creation and the destruction of a name-to object binding is called the binding's lifetime.
- **Object's lifetime** -the time between the creation and destruction of an object is the object's lifetime.
- These lifetimes need not necessarily coincide.
- If object outlives binding it's garbage
- If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is active is its scope

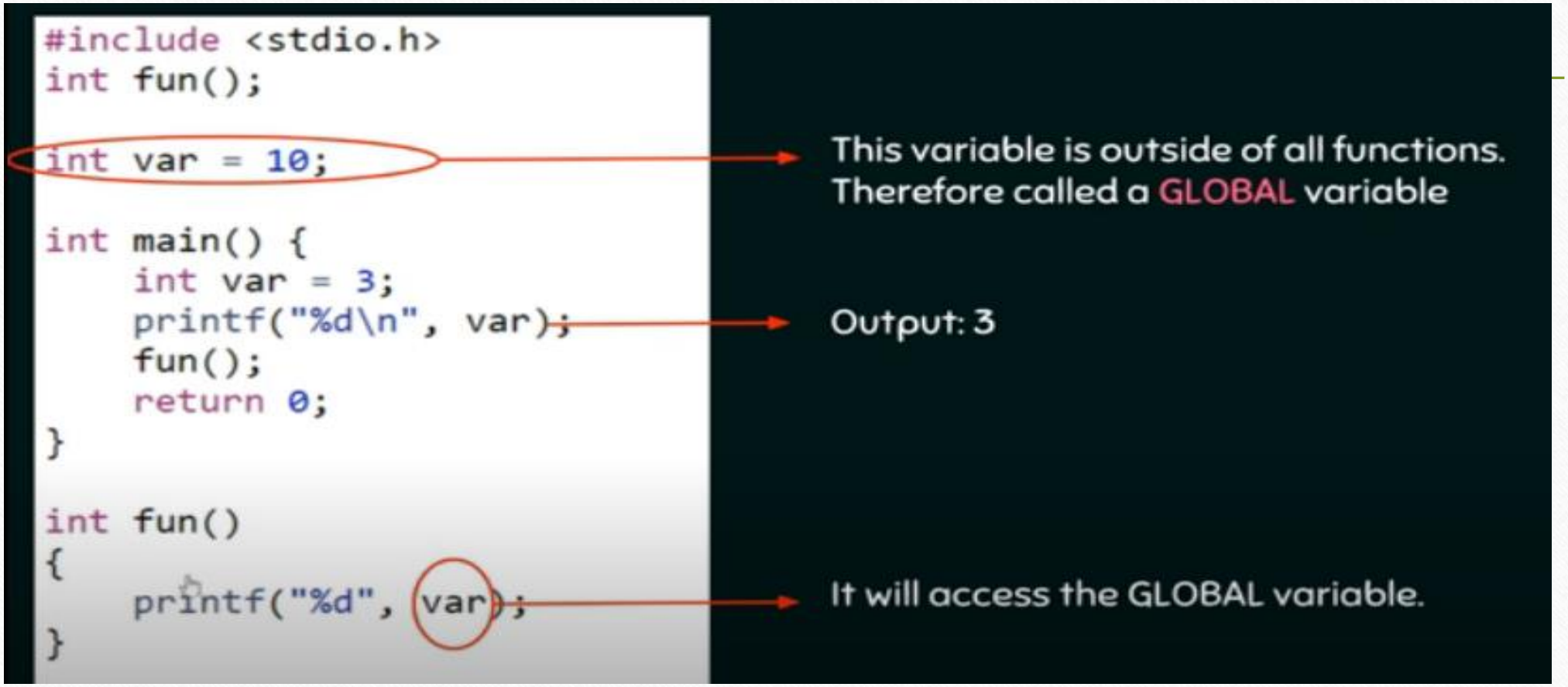
Memory Layout

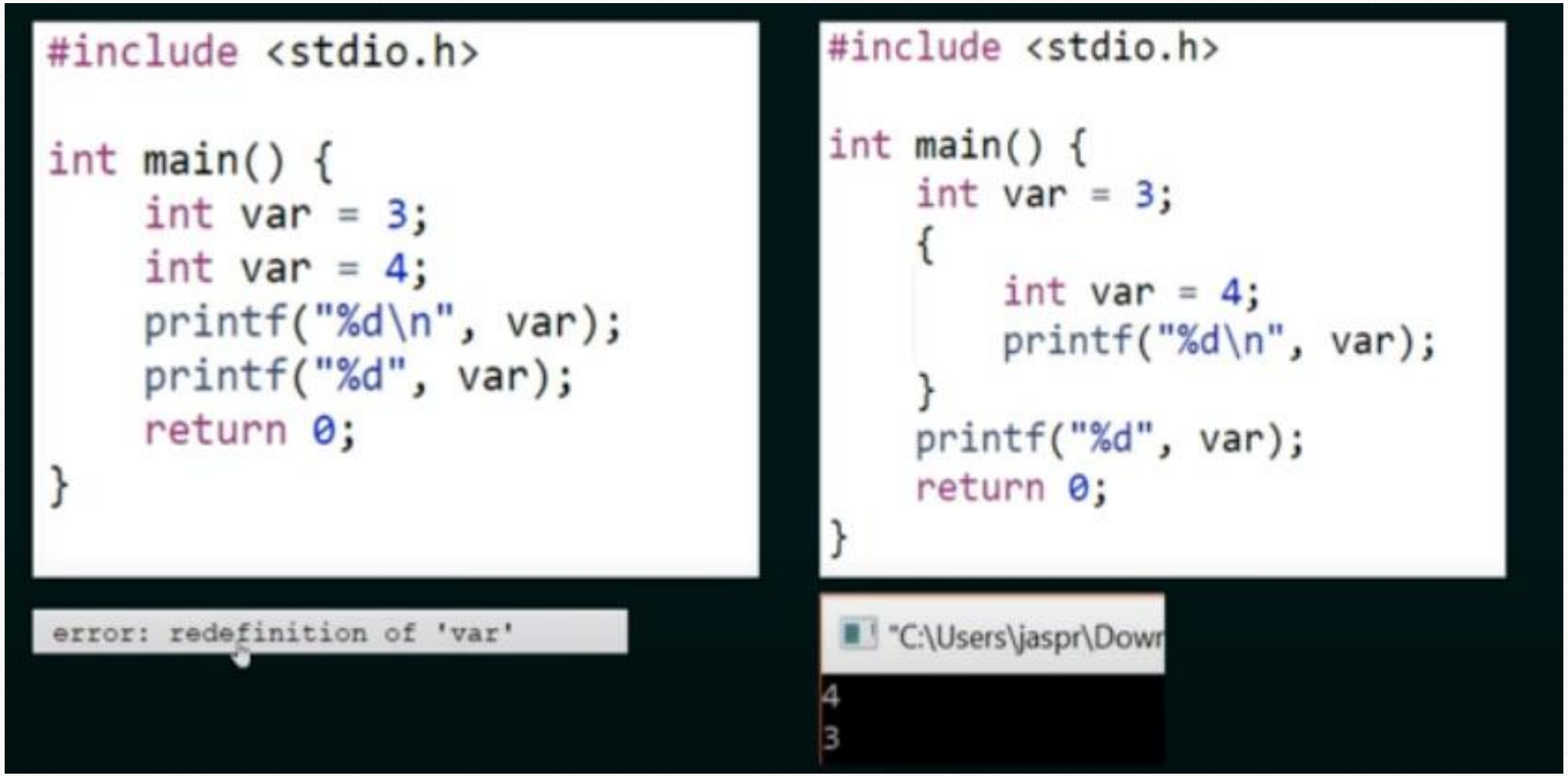


Static and Global Variable

```c
#include <stdio.h>
int fun();

int var = 10;

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

int fun()
{
    printf("%d", var);
}
```

Static and Global Variable




#include <stdio.h>
int main() {
int var = 3;
int var = 4;
printf("%d\n", var);
printf("%d", var);
return 0;
}
error: redefinition of 'var'
#include <stdio.h>
int main() {
int var = 3;
{
int var = 4;
printf("%d\n", var);
}
printf("%d", var);
return 0;
}
"C:\Users\jaspr\Down
4
3

Text Segment

- A text segment , also known as a code segment or simply as text contains executable instructions.
- As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- The text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Initialized Data Segment

- Initialized data segment, or the Data Segment.
- A data segment contains the global variables and static variables that are initialized by the programmer.
- Data segment is not read-only, since the values of the variables can be altered at run time.
- This segment can be further classified into initialized read-only area and initialized read-write area.
- E.g. global string defined by char s[ ] = “hello world”

Uninitialized Data Segment

- Uninitialized data segment, often called the “bss” segment (Block starting symbol)
- Contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
- For instance a variable declared static int i; would be contained in the BSS segment.

Stack

- For recursion, static allocation of local variables won't work

Contents of a stack frame

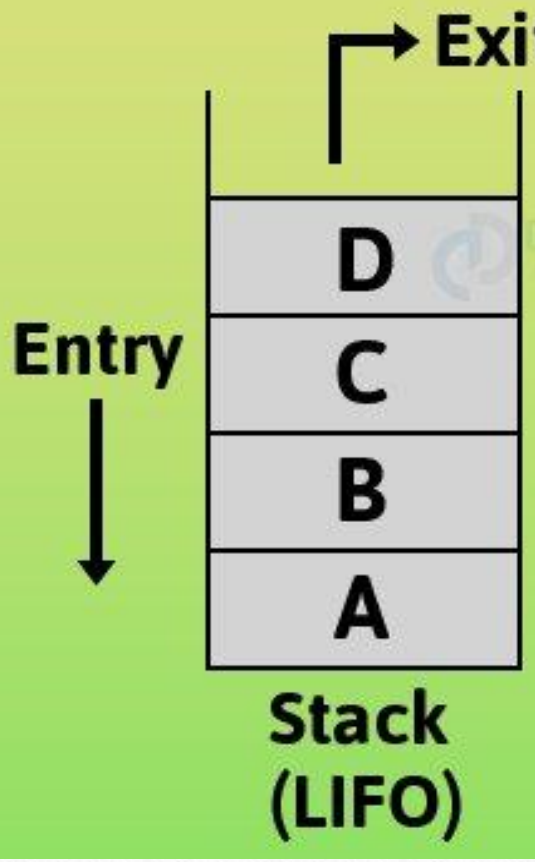
- arguments and return values,
- local variables,
- Temporaries-intermediate values produced in complex calculations
- Bookkeeping information-subroutine's return address, a reference to the stack frame of the caller, additional saved registers, debugging information

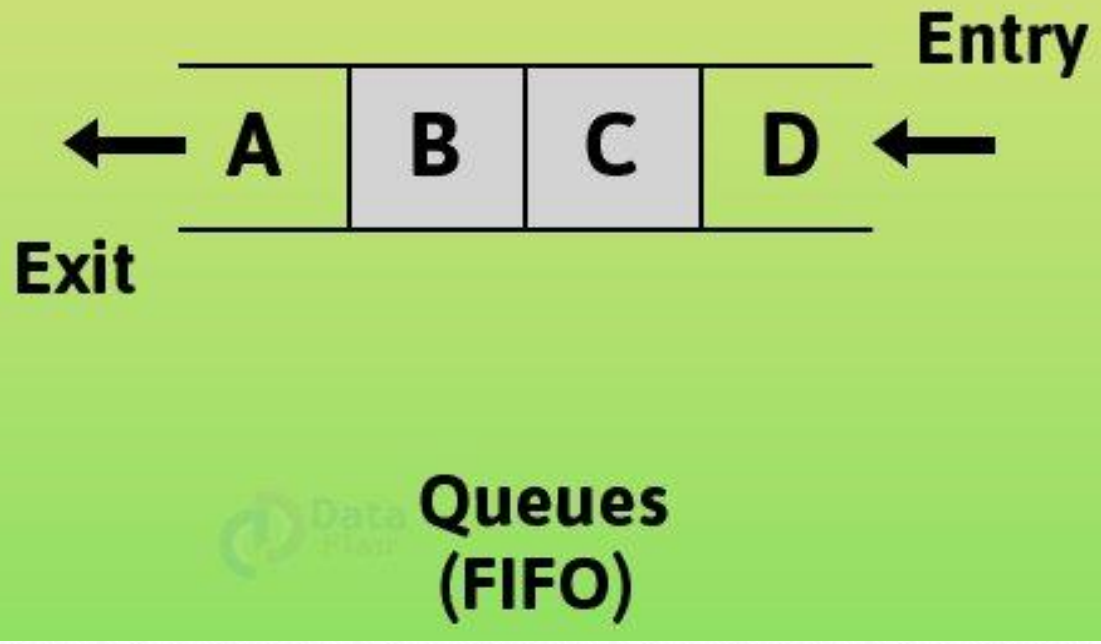
- The stack area, a LIFO structure, is typically located in the higher parts of memory.
- It grows in the opposite direction
- When the stack pointer meets the heap pointer, free memory is exhausted.

Stack

- A “stack pointer” register keep track of the top of the stack;
- It is incremented each time a value is “pushed” onto the stack.
- The set of values pushed for one function call is termed a “stack frame”
- The newly called function then allocates room on the stack for its automatic and temporary variables, return address, other information of caller
- Each time a recursive function calls itself, a new stack frame is used

Stacks and Queues





```
#define LIMIT 100
void display()
{
int stack[LIMIT], top, i;
if(top == -1)
{
printf("Stack underflow\n");          // Stack is empty
}
else if(top > 0)
{
printf("The elements of the stack are:\n");
for(i = top; i >= 0; i--)             // top to bottom traversal
{
printf("%d\n",stack[i]);
}
}
}
```

Heap

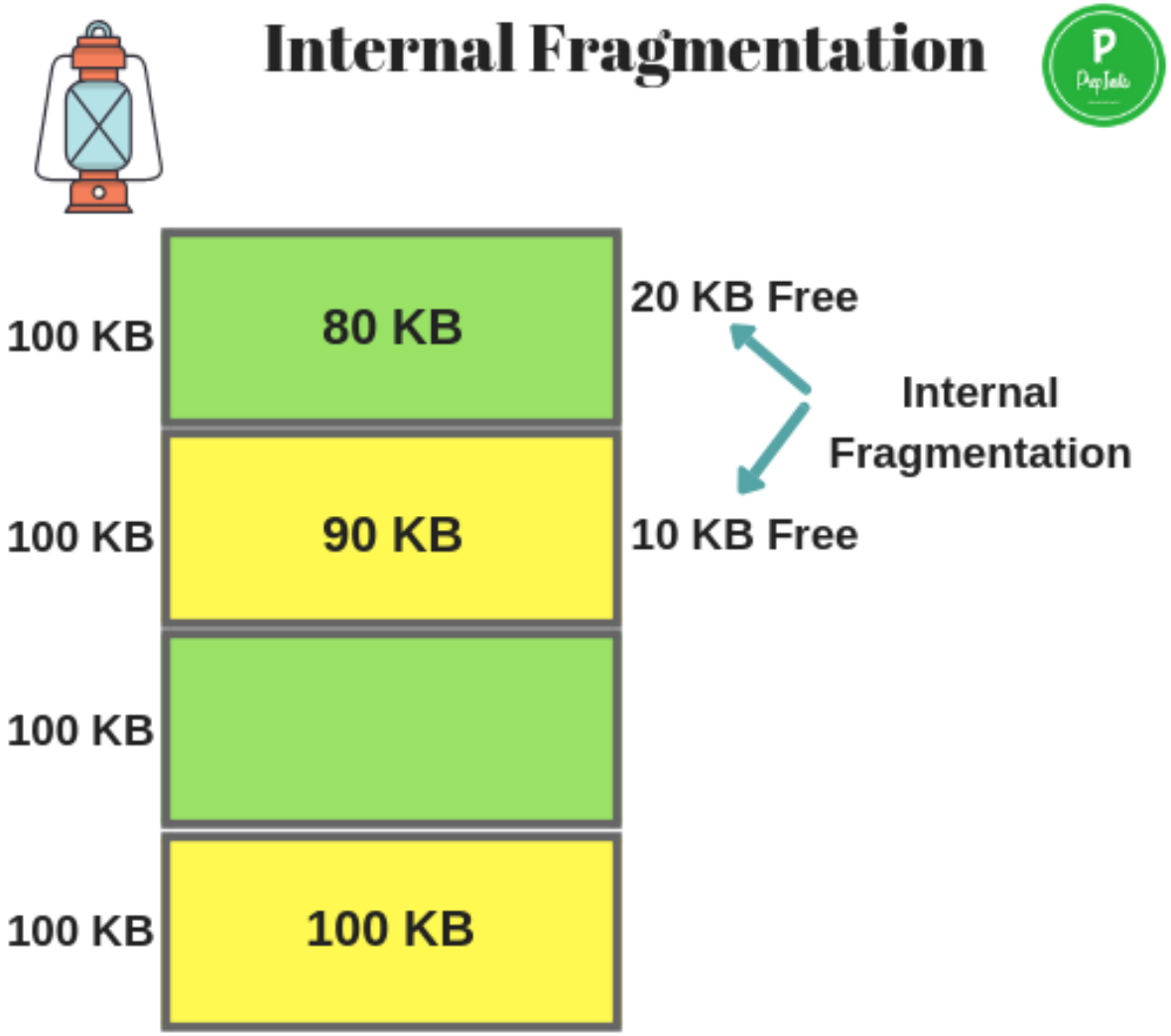
- Heap is the segment where dynamic memory allocation usually takes place.
- The heap area begins at the end of the BSS (Block Starting Symbol)segment and grows to larger addresses from there.
- Heaps are used for storing dynamic data like linked data structures, strings, list, set whose size may change.
- There are many strategies to manage space in a heap.

  Virtual Memory

  Paging and Segmentation

  Leads to Internal and external fragmentation

Internal Fragmentation



Static memory allocation

Memory allocated during compile time is called static memory.

The memory allocated is fixed and cannot be increased or decreased during run time.

Example:

```
int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
}
```

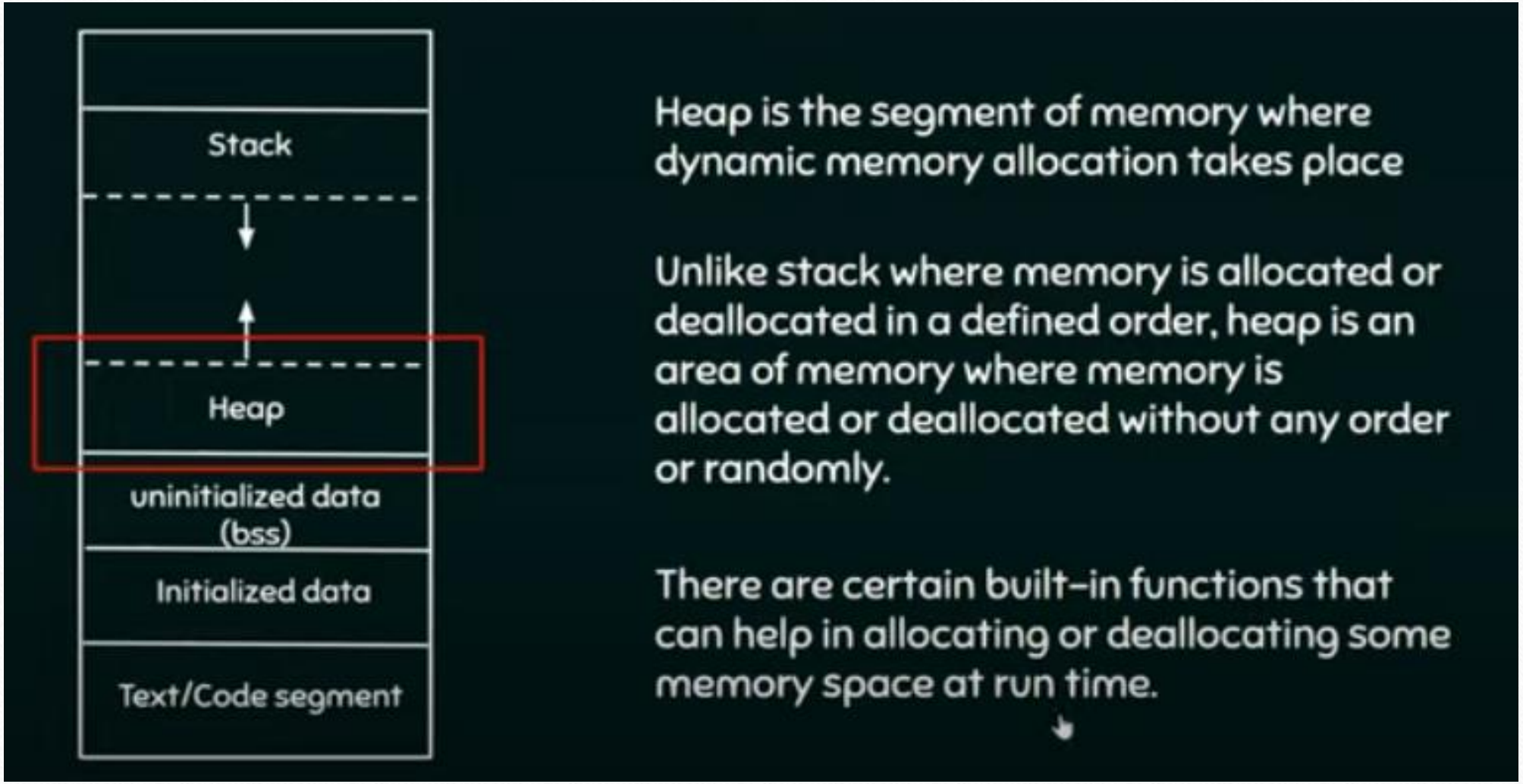
Memory is allocated at compile time and is fixed.

Problems faced in static memory allocation

If you are allocating memory for an array during compile time then you have to fix the size at the time of declaration. Size is fixed and user cannot increase or decrease the size of the array at run time.

If the values stored by the user in the array at run time is less than the size specified then there will be wastage of memory.

If the values stored by the user in the array at run time is more than the size specified then the program may crash or misbehave.


Stack
Heap
uninitialized data
(bss)
Initialized data
Text/Code segment
Heap is the segment of memory where dynamic memory allocation takes place
Unlike stack where memory is allocated or deallocated in a defined order, heap is an area of memory where memory is allocated or deallocated without any order or randomly.
There are certain built-in functions that can help in allocating or deallocating some memory space at run time.

Scope Rules

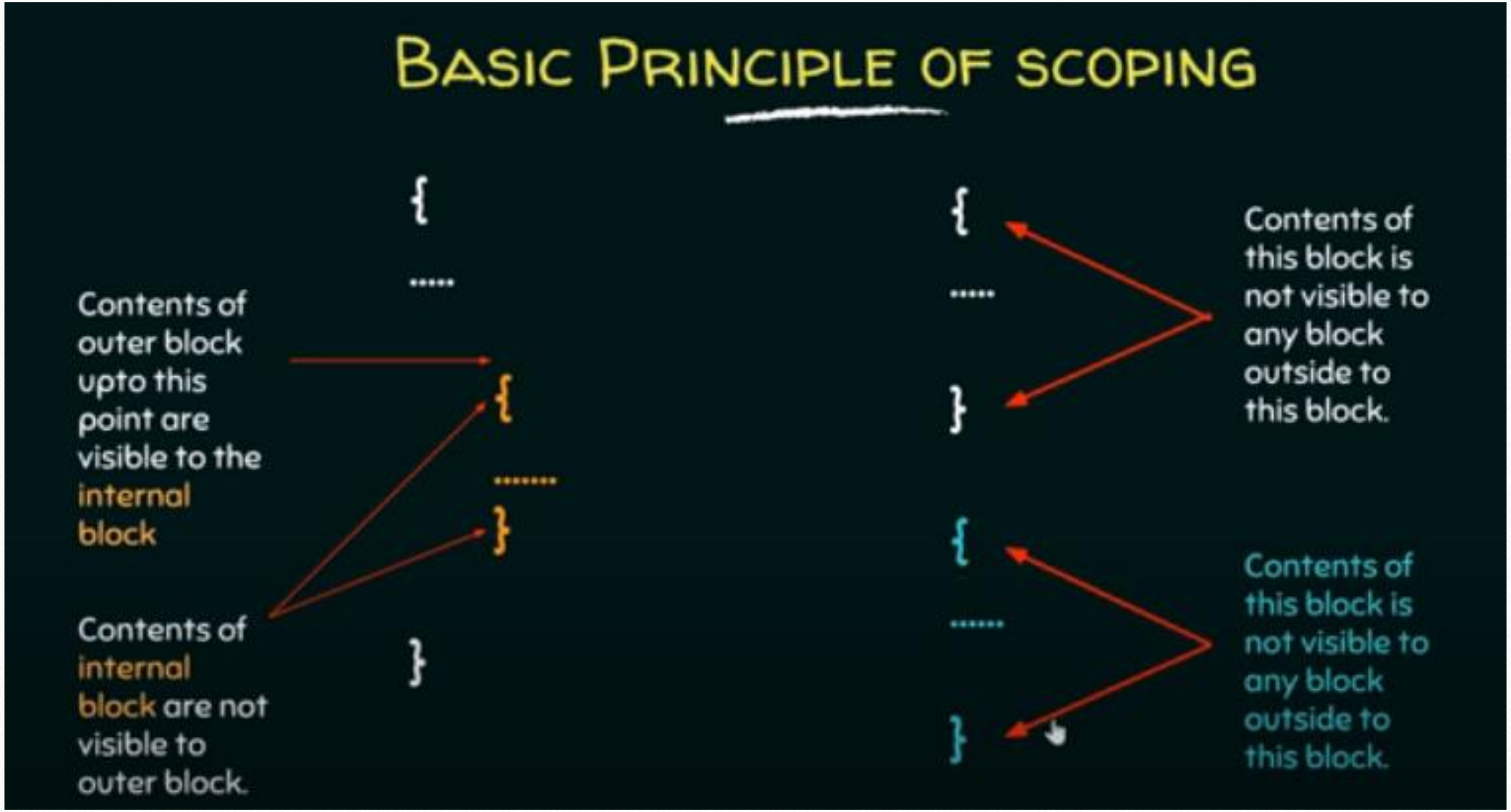
- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted
- In static scoping free variables are resolved from **global variables**
- In Dynamic scoping free variables are resolved from **previous function calls.**

Defining Scope

Scope = Lifetime

The area under which a variable is applicable or alive.

Strict Definition: a block or a region where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed.


Basic Principle of scoping
{
.....
Contents of outer block upto this point are visible to the internal block
{
.....
}
Contents of internal block are not visible to outer block.
}
{
.....
}
Contents of this block is not visible to any block outside to this block.
{
.....
}
Contents of this block is not visible to any block outside to this block.

Scope Rules

```
#include <stdio.h>
int a=50, b=60;
swap();
int main()
{
   int a=10, b=20;
   swap();
   printf("\nA & b from main %d %d",a,b);
   return 0;
}
```

```
swap()
{
   int t;
   t=a;
   a=b;
   b=t;
   printf("\nA & b from swap %d %d",a,b);
}
```

Output:

Scoping

- **In static Scoping**

In swap function

a's value = global variable value

In main function

a's value = locally declared value

- **In Dynamic Scoping**

In swap function

- a & b values are resolved from previous function call
- main() is calling the swap() function, therefore values defined in main would be considered
- a's value = values defined in main()

In main function

- a's value = locally declared value

Thank You