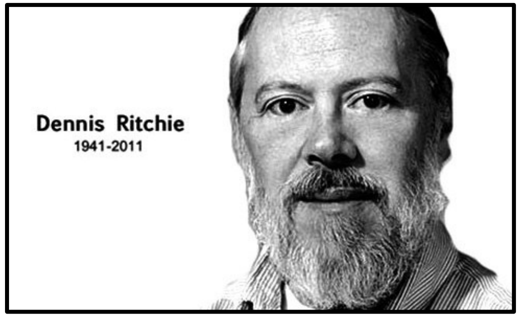
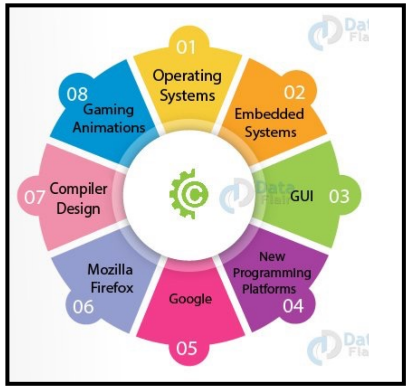
C Primer

CS2106 Introduction to Operating Systems

What is C?

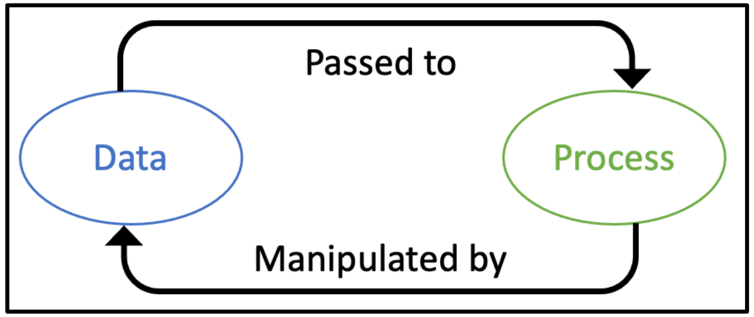
- A ***programming language*** created by ***Dennis Ritchie*** in the late 1960s and early 1970s.





Pic from: https://data-flair.training/blogs/applications-of-c/

- C is
  - ***General-purpose***: used for building ***variety of applications***
  - ***Procedural***: consists of ***procedures*** to perform tasks



- C ***program***
  - Collection of ***C source code*** (with `.c` extension)
  - One or more ***header files*** (with `.h` extension)

C is ***General-Purpose*** Language

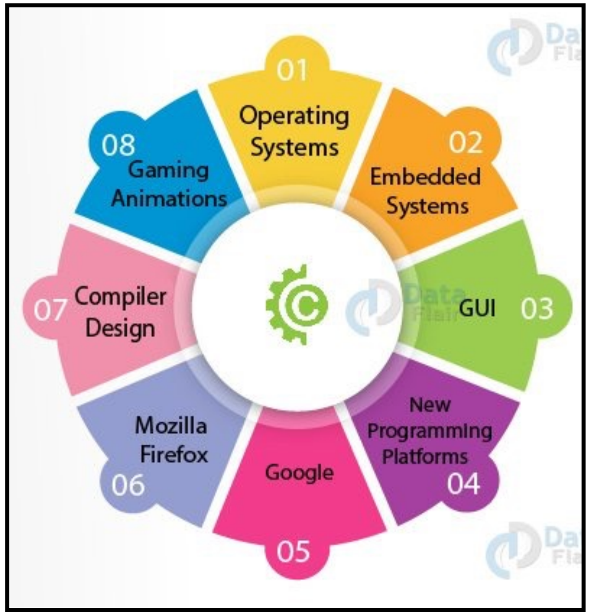
- Wide variety of ***real-world applications***

- ***Operating Systems: Linux and Windows OS*** are programmed in C

- ***Embedded Systems:*** scripting ***drivers*** or program ***microcontroller*** for embedded systems

- ***Compilers:*** compilers were designed using C such as Bloodshed Dev-C, Clang C, MINGW, and Apple C.

- … (https://data-flair.training/blogs/applications-of-c/)



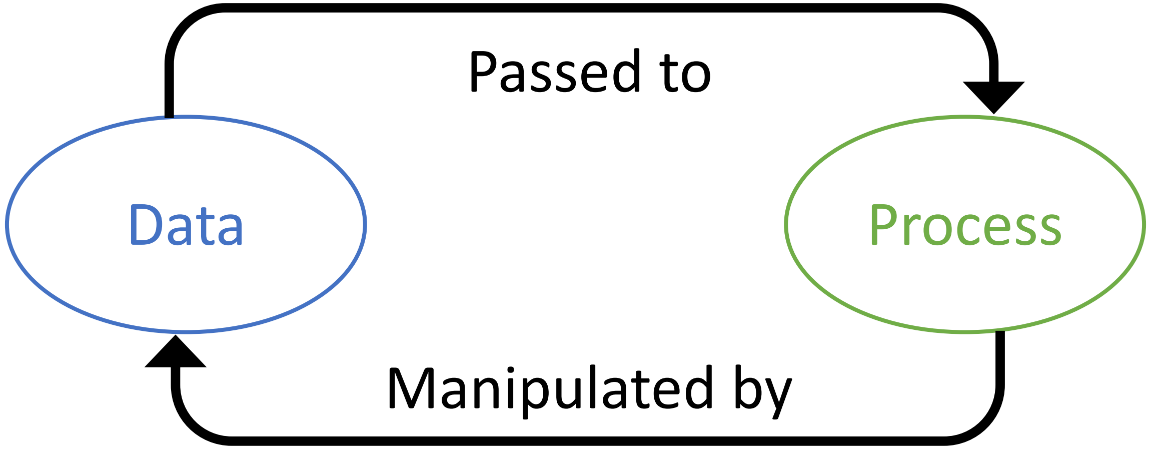
Pic from: https://data-flair.training/blogs/applications-of-c/

C Follows ***Procedural Programming Model***

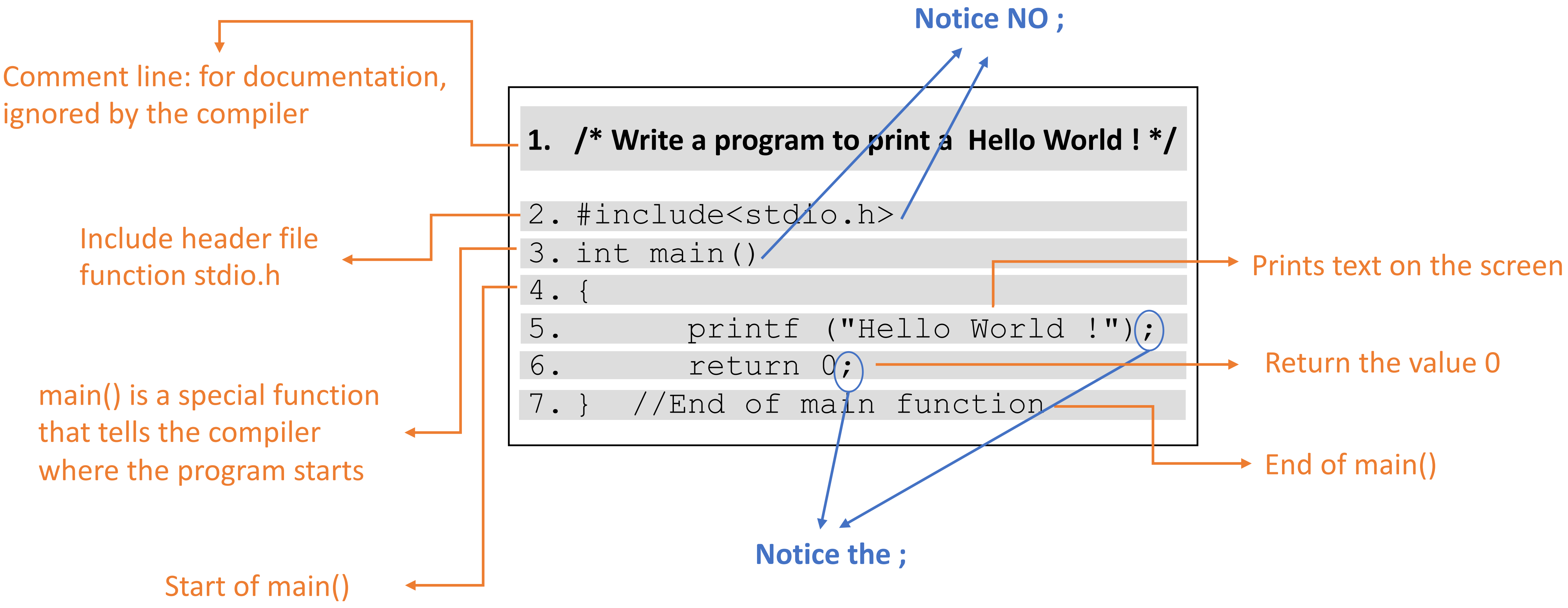
- Specifies a ***well-defined procedure*** to complete a task

- Consist of
  - ***Data:*** directly accessed by the process
  - ***Process:*** are functions or procedures that manipulate the data

- Programmer responsibility to:
  - Introduce ***meaningful organization***
  - ***Separate process and data*** into logical groups



A Simple ***“Hello World !”*** C program



```c
/* Write a program to print a  Hello World ! */

#include<stdio.h>
int main()
{
        printf ("Hello World !");
        return 0;
}  //End of main function
```

Comment line: for documentation, ignored by the compiler

Include header file function stdio.h

main() is a special function that tells the compiler where the program starts

Start of main()

Notice NO ;

Prints text on the screen

Return the value 0

End of main()

Notice the ;

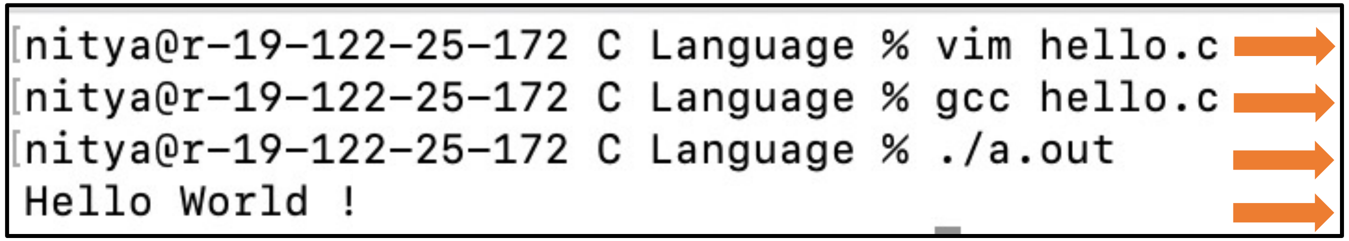
Compile and Execute a C Program

```
/* Write a program to print a  Hello World ! */

#include<stdio.h>
int main()
{
        printf ("Hello World !");
        return 0;
}   //End of main function
```

Steps

1. Write program in any editor (vim, …etc)
2. Save it with extension .c (hello.c)
3. Compile using gcc on command prompt
4. Run using ./a.out

Command prompt



```
[nitya@r-19-122-25-172 C Language % vim hello.c
[nitya@r-19-122-25-172 C Language % gcc hello.c
[nitya@r-19-122-25-172 C Language % ./a.out
Hello World !
```

Write program in vim editor

Compile using gcc, generate a.out file

Run using ./a.out

Output

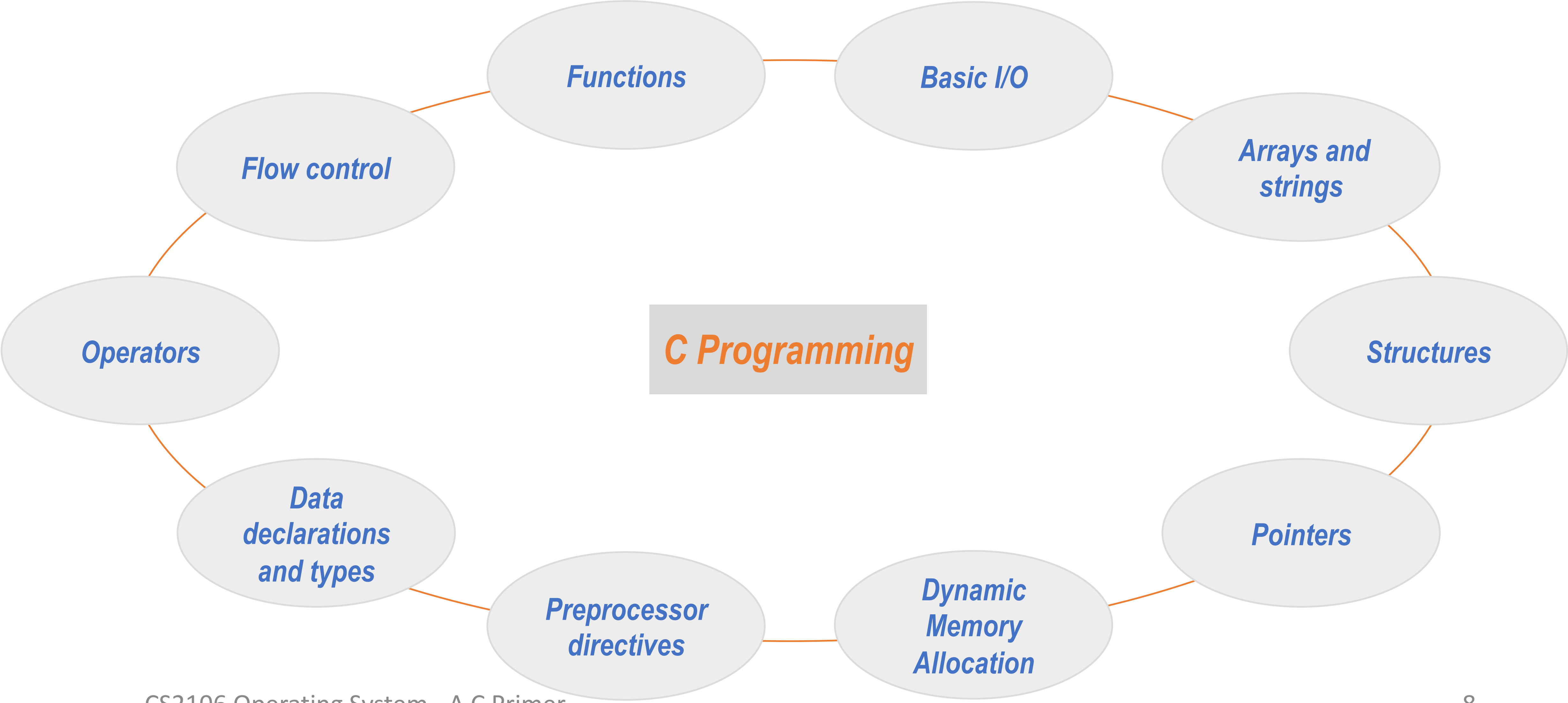
C and Java: ***Program Comparison***

C	Java

`#include <stdio.h>`
` int main( ) {`
`        printf( "Hello World!" );`
`        return 0;`
`}`
	`public class HelloWorld {`
	`public static void main( String args[] ) {`
	`System.out.println( "Hello World!" );`
	`}`
	`}`

C is classless (***no concept*** of class)	Has `class` that ***encapsulates data and methods (or function)*** into a single unit
Procedural programming language	***Object-Oriented*** programming language
Source file: `hello.c` (no restriction)	***Source file:*** `HelloWorld.java` (same name as class with main())
Compile: `gcc hello.c`	***Compile:*** `javac HelloWorld.java`
Execution: `./a.out`	***Execution:*** `java HelloWorld`

Overview



C Programming

Functions

Basic I/O

Arrays and strings

Structures

Pointers

Dynamic Memory Allocation

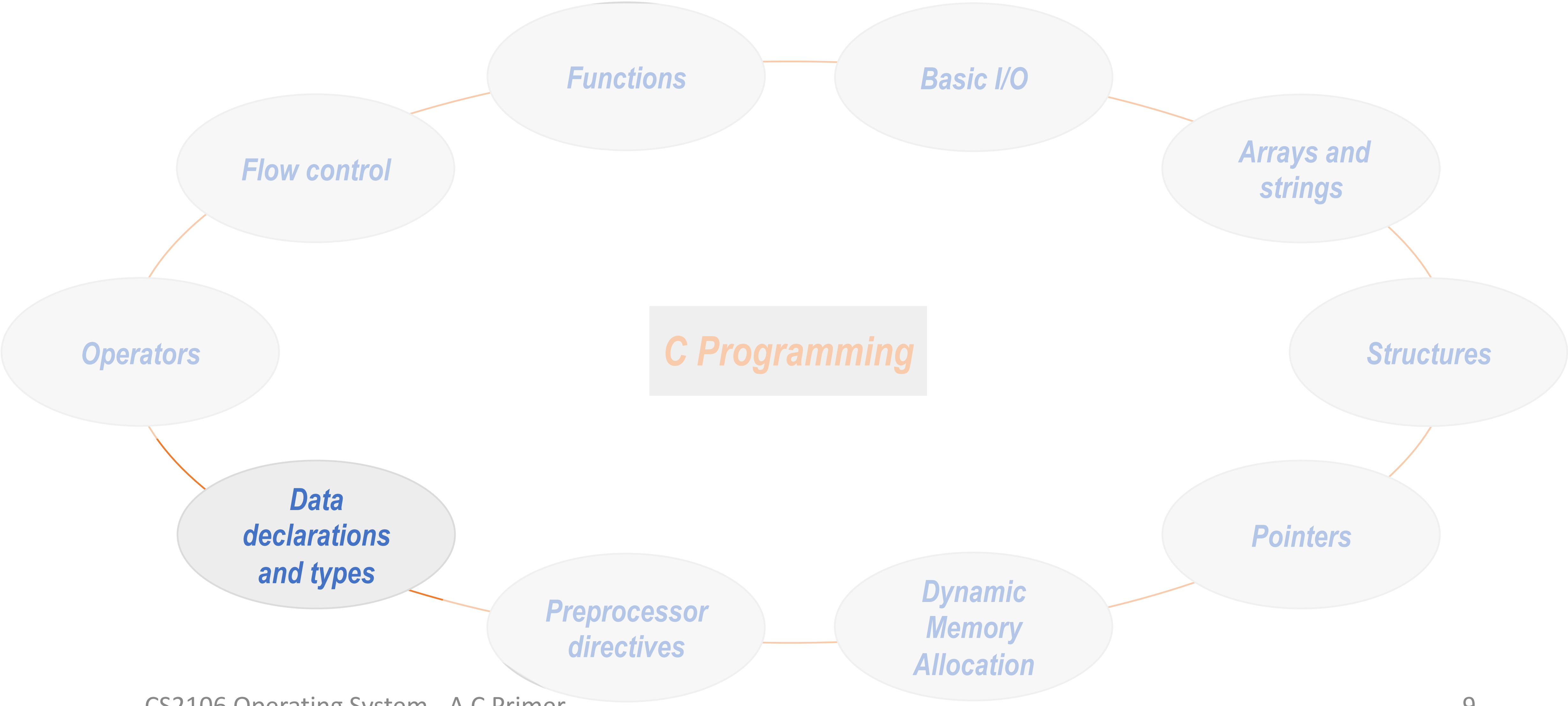
Preprocessor directives

Data declarations and types

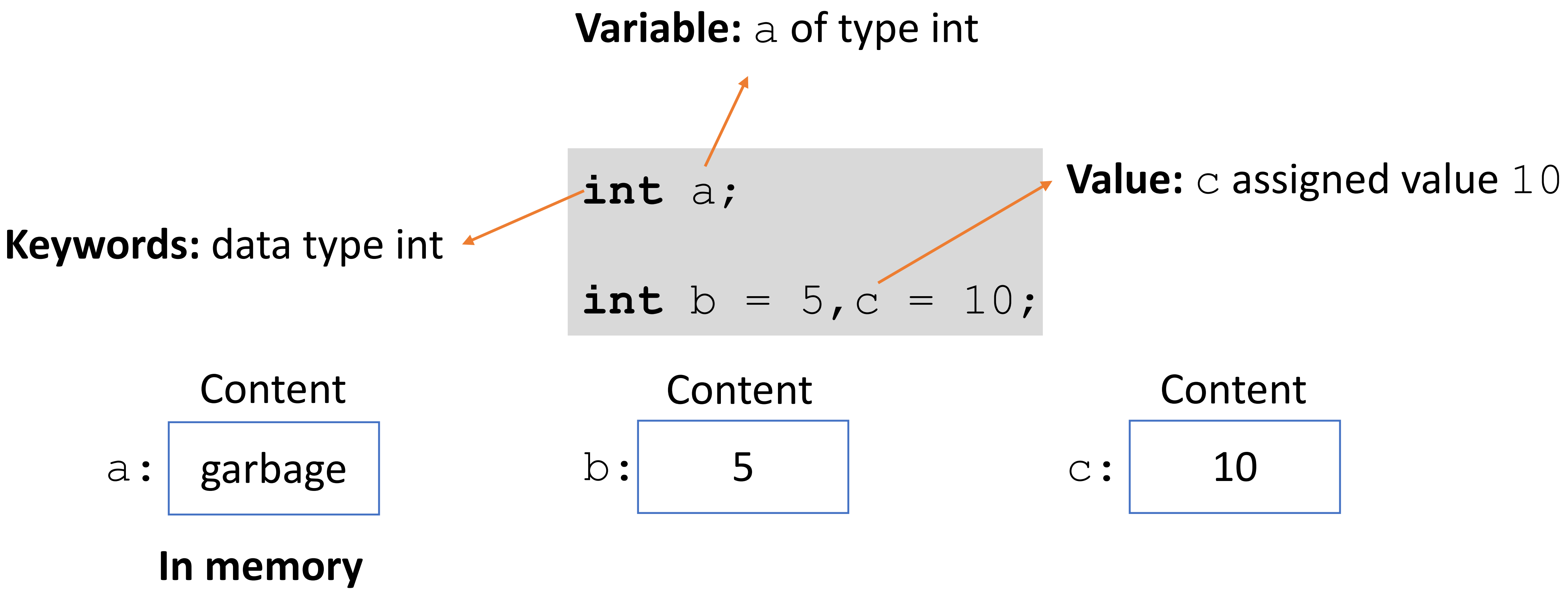
Operators

Flow control

Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Data *Declaration*



Variable: `a` of type int

```
int a;

int b = 5,c = 10;
```

Keywords: data type int

Value: `c` assigned value `10`

	Content
`a:`	garbage
`b:`	5
`c:`	10

In memory

Primary Data Types

Integers

- `int` or `signed int` => positive and negative integer values (2 bytes)
- `unsigned int` => only non-negative integer values

Floating points

- `float` => for real numbers (4 byte)
- `double` => same as float but with longer precision

Character

- `char` or `signed char` => character constant, stored as ASCII (1 byte)
- ASCII => character encoding scheme (e.g., ‘A’ is stored as 65)

Void

- `void` => specify an empty set of values; used as return for functions

Data ***Declaration*** vs Data ***Definition***

- ***Declaration of a variable***
  - Informs compiler about ***name and type*** of the variable, ***initial values*** if any

- ***Definition of a variable***
  - Compiler ***allocates memory*** for the variable

- In C, data declaration and definition ***take place at the same time***

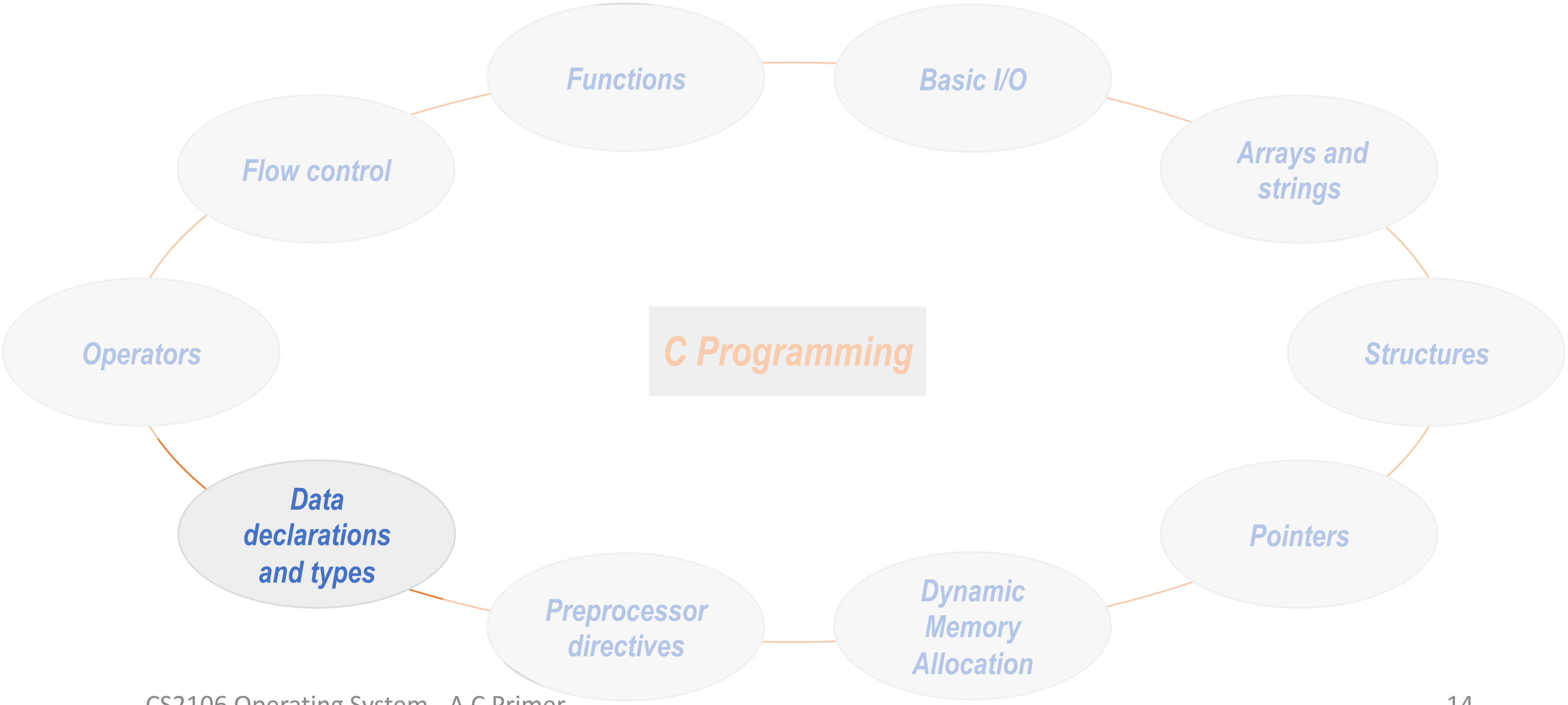
- To ***only declare and not define*** a variable: `extern int a;`
  - Declare a variable `a` of type `int`, no memory allocated
  - Need to define the variable somewhere else

C and Java: ***Data Type Comparison***

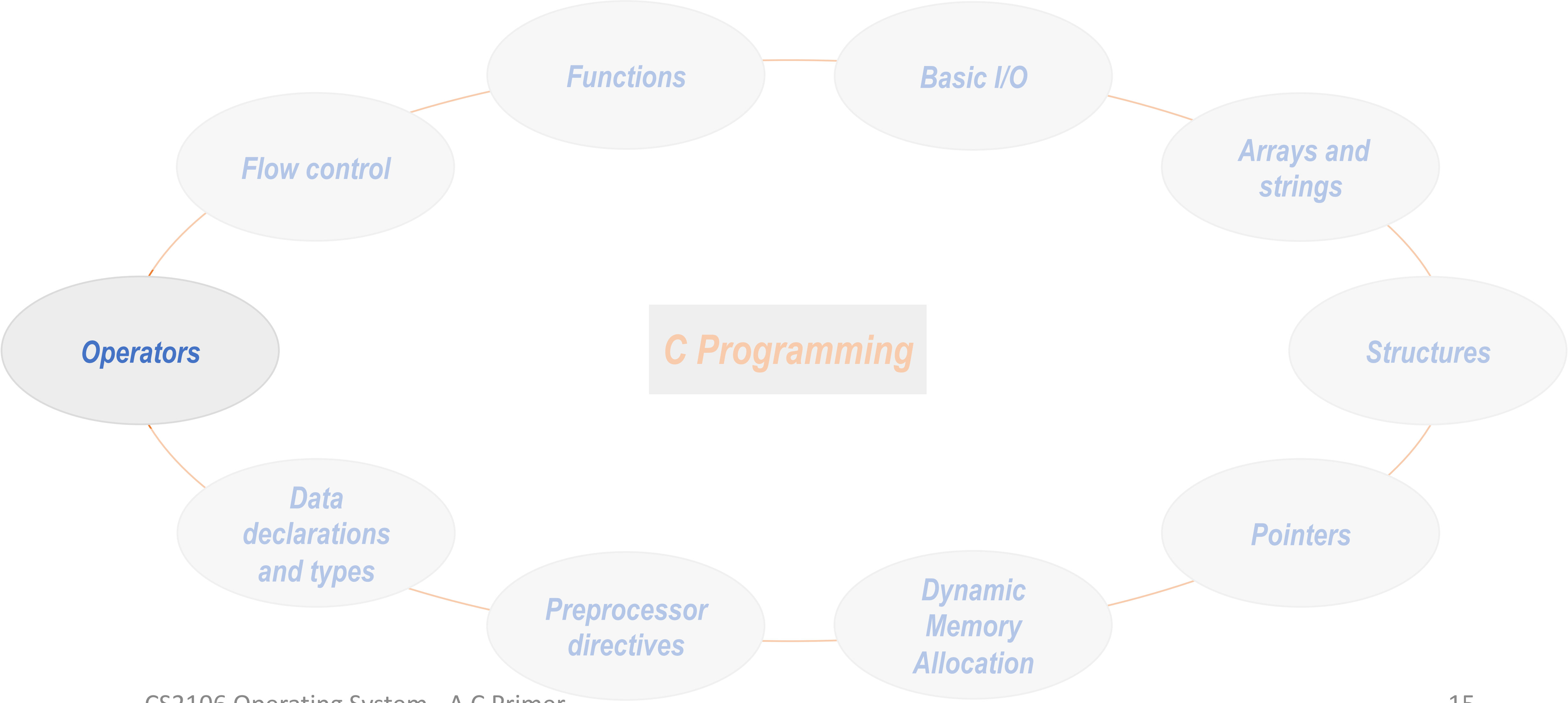
- C ***does not have object version*** of primary data type
  - Java has ***Integer class*** :
    - Wrapper class for ***`int`*** ***data types***
    - Contains ***function*** to deal with `int` values (e.g., convert `int` to `float`)
  - C ***does no***t have such a version

- `char` data type in C is ***ASCII encoding*** whereas in Java is ***Unicode encoding***
  - ASCII represents max of 128 characters (***1 byte***)
  - Unicode represent max of 65,536 characters (***2 bytes***)

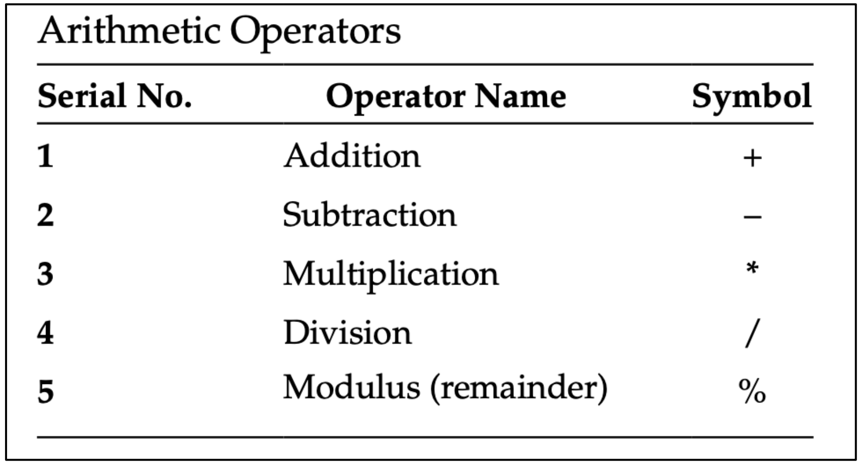
Overview


C Programming
Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control

Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Operators

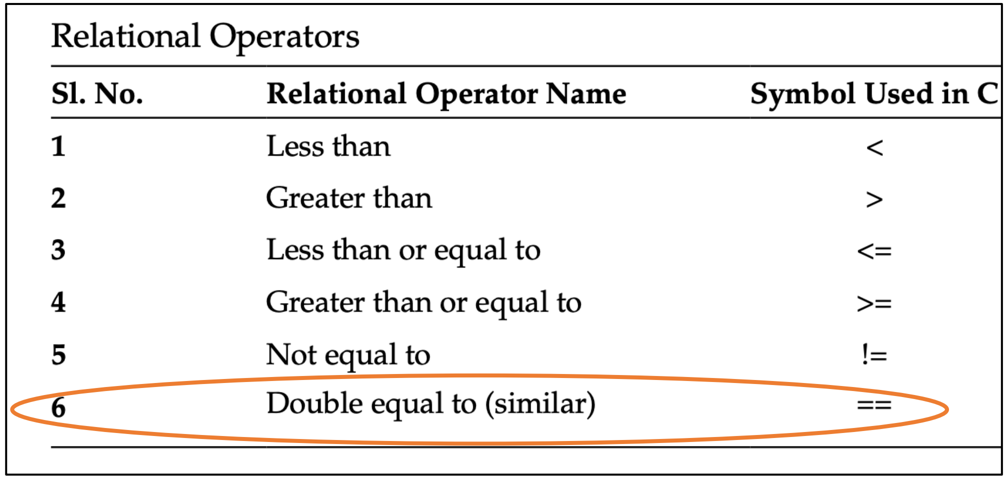


Arithmetic Operators

Serial No.	Operator Name	Symbol
1	Addition	+
2	Subtraction	–
3	Multiplication	*
4	Division	/
5	Modulus (remainder)	%

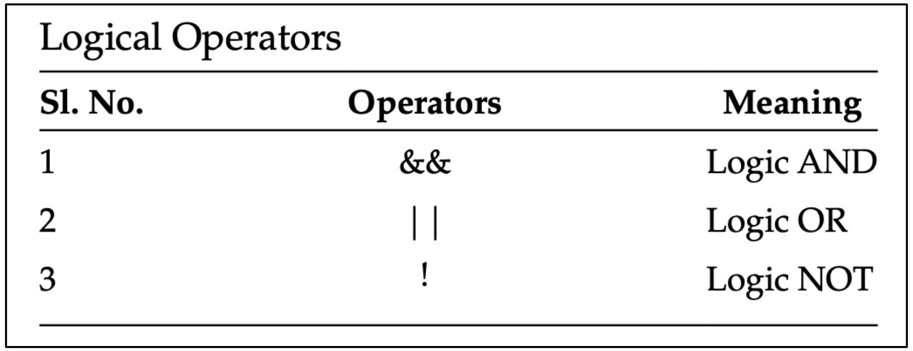
a==b

a=b



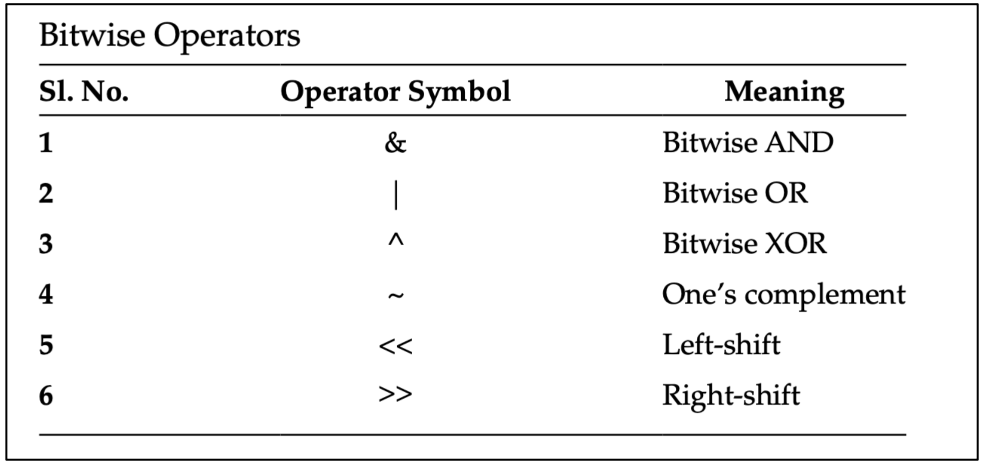
Relational Operators

Sl. No.	Relational Operator Name	Symbol Used in C
1	Less than	<
2	Greater than	>
3	Less than or equal to	<=
4	Greater than or equal to	>=
5	Not equal to	!=
6	Double equal to (similar)	==



Logical Operators

Sl. No.	Operators	Meaning
1	&&	Logic AND
2	| |	Logic OR
3	!	Logic NOT



Bitwise Operators

Sl. No.	Operator Symbol	Meaning
1	&	Bitwise AND
2	|	Bitwise OR
3	^	Bitwise XOR
4	~	One's complement
5	<<	Left-shift
6	>>	Right-shift

Increment/Decrement Operator

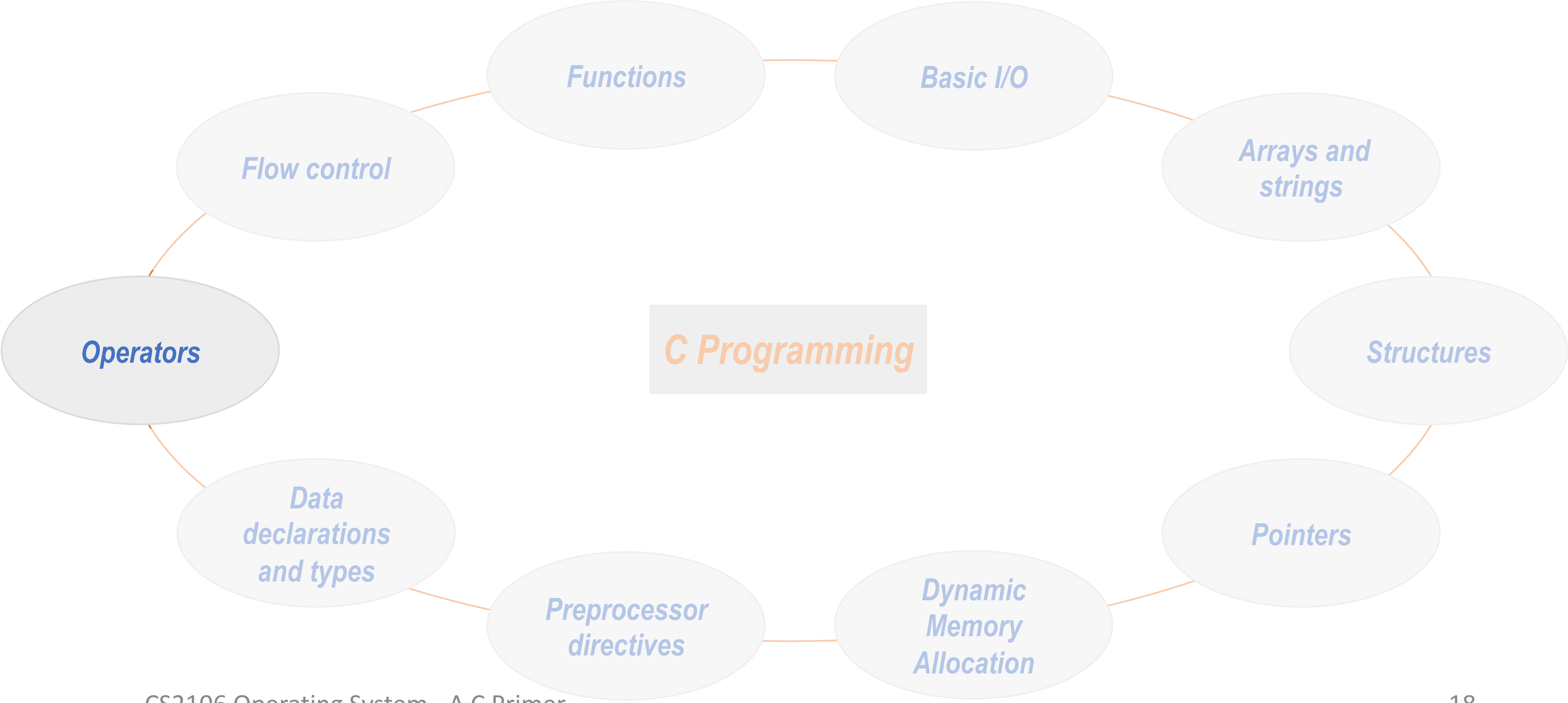
- ***Increment (++)***
  - will increment the value of a variable by 1
  - `x++` is same as `x=x+1`

- ***Decrement (– –)***
  - will decrement the value of a variable by 1
  - `x--` is same as `x=x-1`

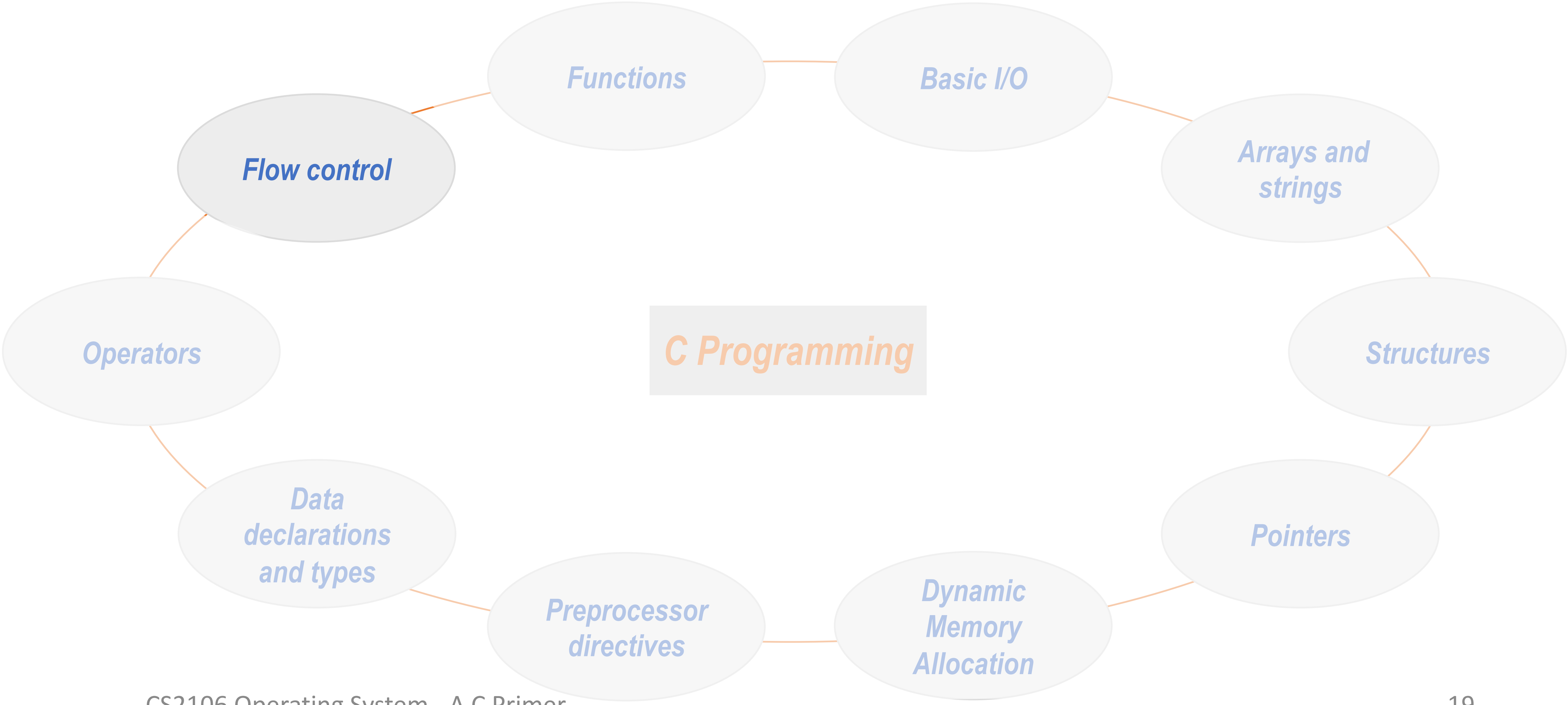
Example:

```
    x=5, y=5
    x++        x=x+1 = 6
    y--        y= y-1 = 4
```

Overview

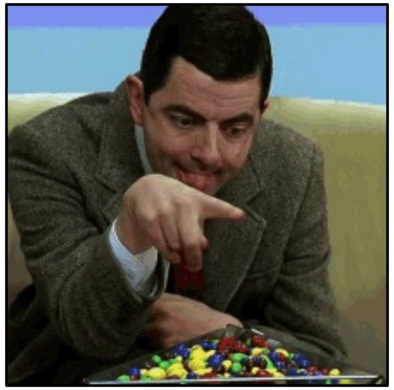

Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Flow Control Statements

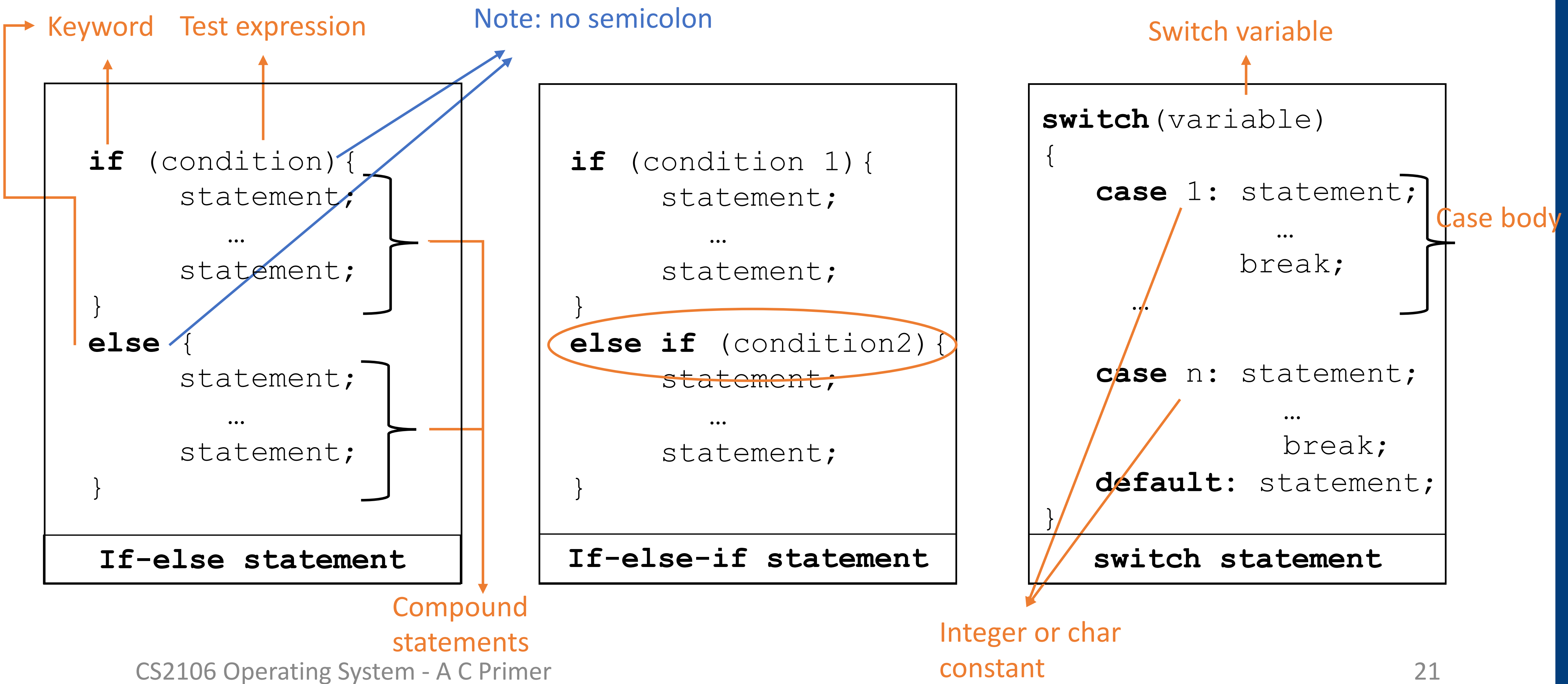
- Indicate the ***order*** in which the various instructions are executed



- Two categories:
  - ***Selection flow control:*** makes decisions about which statement is to be executed next.
  - ***Looping flow control:*** to execute group/single statements repeatedly until a condition is satisfied.

- Similar to Java

Flow Control: ***Selection***



Keyword

Test expression

Note: no semicolon

```
if (condition){
      statement;
         …
      statement;
}
else {
      statement;
         …
      statement;
}
```

If-else statement

Compound statements

```
if (condition 1){
      statement;
         …
      statement;
}
else if (condition2){
      statement;
         …
      statement;
}
```

If-else-if statement

Switch variable

```
switch(variable)
{
   case 1: statement;
             …
           break;
   …

   case n: statement;
             …
             break;
   default: statement;
}
```

switch statement

Case body

Integer or char constant

Flow Control: ***Selection Example***

```
int a=8, b=10;
if (a<b){
     printf(“a<b”);
}
else {
      printf(“a>b”);
}
```

```
int a=12,b=10,c=15;
if (a<b){
     printf(“a<b”);
}
else if(a<c) {
      printf(“a<c”);
}
```

```
n=2;
switch(n):
{
  case 1: printf(“n is 1”);
          break;

  case 2: printf(“n is 2”);
          break;

 default: printf(“neither 1
or 2”);
}
```

Flow Control: ***Loops***

```
Initialize

while(condition){
      statement 1;
         …
      statement n;
}
```

Test condition first and repeat statements until condition is false

while statement

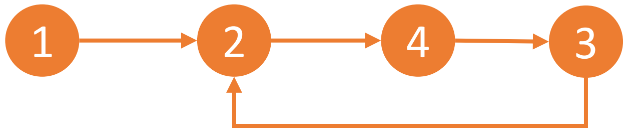
```
Initialize

do {
      statement 1;
         …
      statement n;
} while(condition);
```

Execute the statement then test condition, if true, repeat statements

do-while statement

```
for(initialization;           1
condition;  2
increment/decrement)  3
{
        statement 1;
         …              4
        statement n;
}
```

Order of execution:



for statement

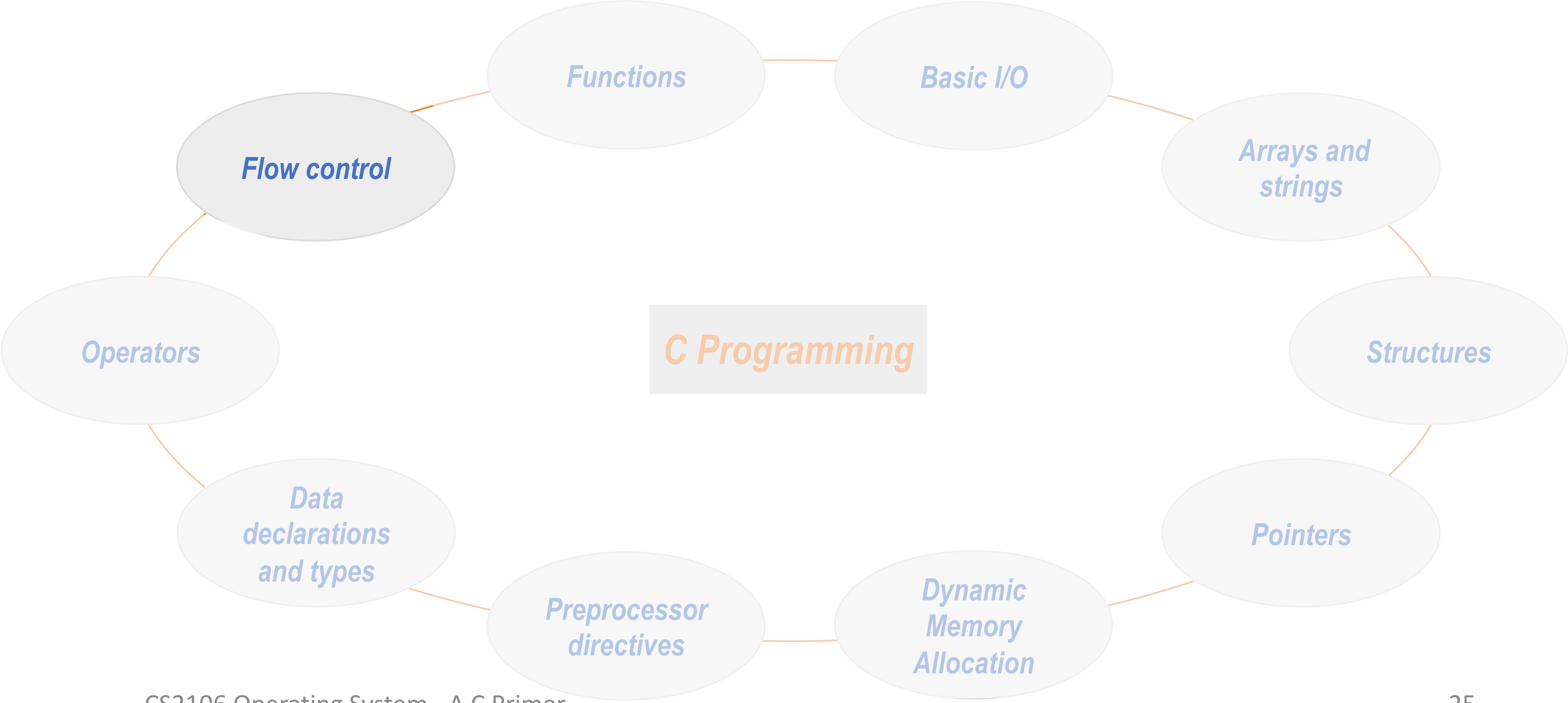
Flow Control: ***Loops Example***

```
int i=0;

while(i<10){
        printf(“Loop”);
        i++;
}
```

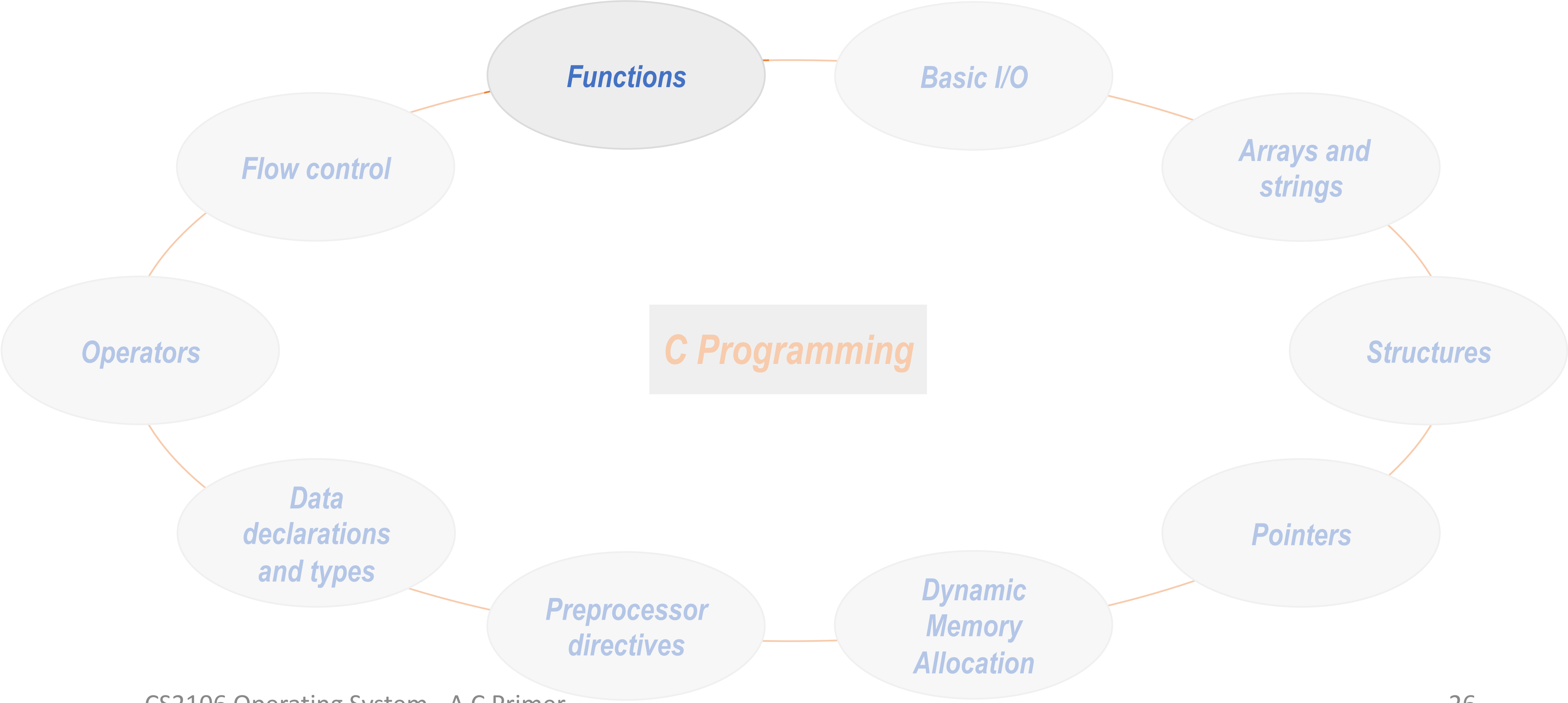
```
int i=10;

do{
        printf(“Loop”);
        i++;
}while(i<10);
```

```
int i;
for(i=0;i<10;i++){
        printf(“Loop”);
}
```

Overview


Flow control
Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
C Programming

Overview


Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control
C Programming

Functions

- Group of statements that ***performs a specific task***

- ***Advantages:***
  - Helps to write a ***modular program***
  - Program and function debugging are ***easier***
  - ***Reduction in size*** of the code

- ***Two types:***
  - ***User-defined*** functions
  - ***Library*** or ***pre-defined*** functions

User-Defined Functions

- Function which are ***written by the users*** for a specific a task

Function declaration provide information such as function name and parameter list

Function definition when the memory is ***allocated***

- ***Function prototypes:***

```
ReturnType FunctionName(Parameter list);
```

- ***Function definition:***

```
ReturnType FunctionName(Parameter list){
      statement 1;
          …
      statement n;
      return var;
}
```

Formal parameters

- ***Function call:***

```
ReturnType var = FunctionName(Parameter list);
```

Actual parameters

Function *Example*

Find ***factorial*** of a number

```
int factorial(int n);
```

Function prototype

```
int main(){
    int n=10, fact;
    fact = factorial(10);
}
```

Function call

```
int factorial(int n){

        int result =1, i;
        for(i=2;i<=n;i++)
                result = result * i;
        return result;
}
```

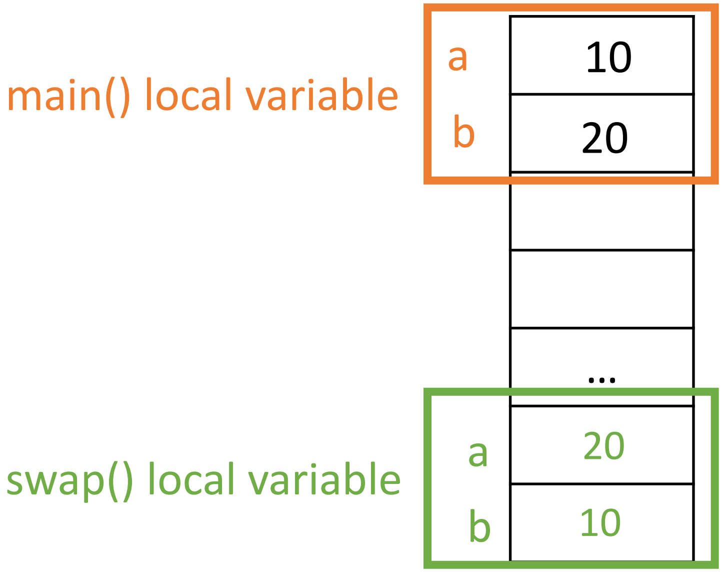
Function definition

Parameter Passing: ***Pass by Value***

- Function ***makes a copy*** of the parameters and works on that copy.
- ***Actual*** parameter values are ***unaffected***

```
void swap(int a, int b);
int main(){
   int a=10, b=20;
   swap(a,b);
}
void swap(int a, int b){

      int t;
      t=a;
      a=b;
      b=t;
}
```

When swap() function is called, only local variable a, b swaps



C and Java: ***Function Comparison***

- For ***both C and Java***
  - Functions performs a specific task (same purpose)
- In C, functions ***are not tied*** to a class or structure
- In Java,
  - ***Always a part of a class***, cannot exist like in C
  - Call a function ***using the object*** of the class

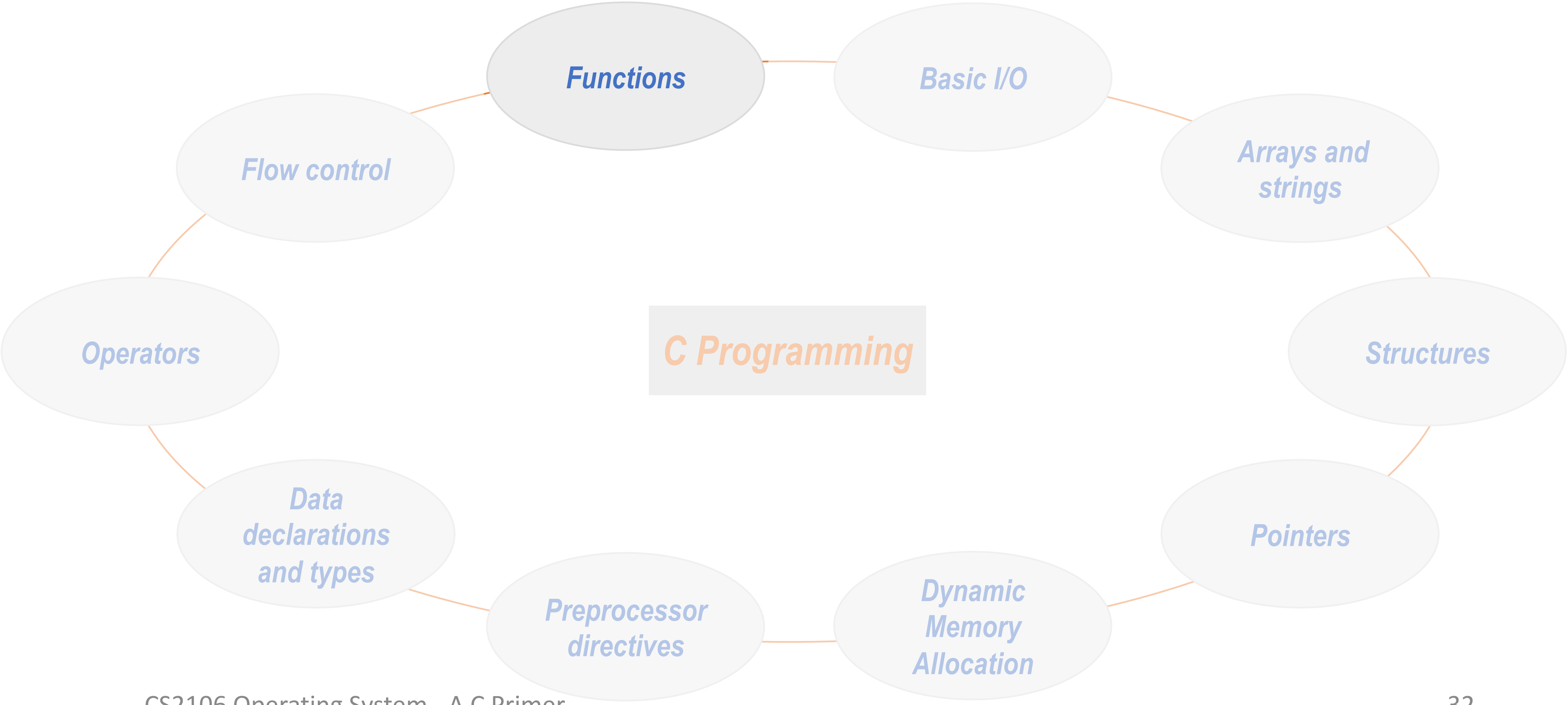
C

```
void display(){
   printf("Test Method");
}
int main()  {
    printf("Hello World!");
    display();
 }
```

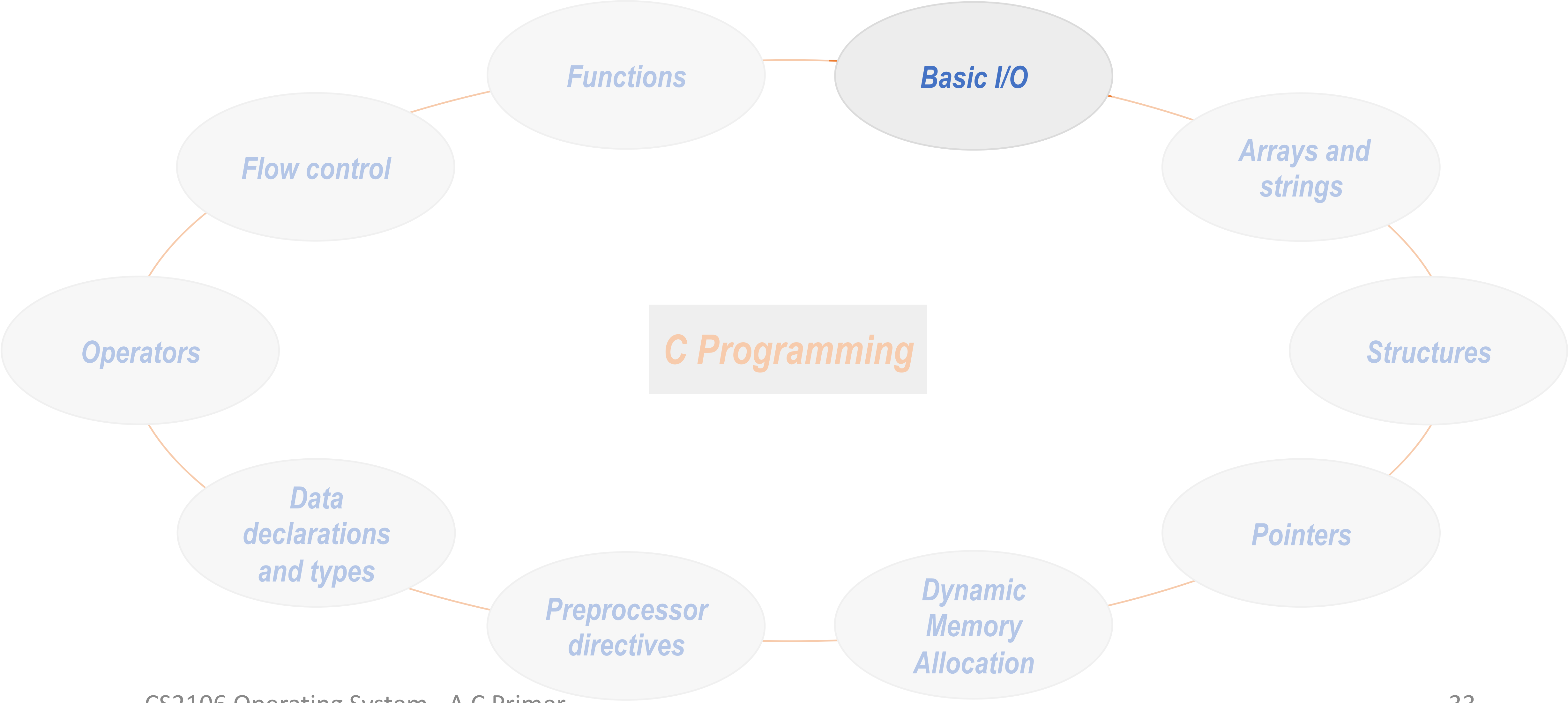
JAVA

```
class  Test {
    public void display(){
        System.out.println("Test Method");
    }
}
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World!");
        Test obj = new Test();
        obj.display();
    }
}
```

Overview


Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control
C Programming

Overview


Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control
C Programming

Pre-Defined Functions: Basic ***Input/Output***

- ***Do not write*** the function definition, only ***use them***

- Number of ***in-build library I/O functions*** available in C
  - ***Console I/O*** => takes input from the keyboard and prints output on monitor
  - ***Disk or File I/O*** => read and write from/to files on disk

- ***Console I/O***
  - ***Header files***: `#include<stdio.h>,#include<conio.h>` (for some cases)
  - Include at the beginning of the program
  - ***Common output function :*** `printf`
  - ***Common input function :*** `scanf`

Basic ***Output*** Function: `printf`

- `printf` => prints output on to the screen or monitor

- Syntax: `printf(“Format strings”, list of variables);`

- `Format strings`
  - Specify format for the `variables` printed (if any)
  - E.g.,

```
int a = 5;
float b = 5.6;
printf(“a=%i b=%f”,a,b);
```

Basic *Output* Function: `printf`

- Common ***format specifiers***:

`%i or %d`	integer
`%u`	unsigned int
`%f`	float
`%lf`	double
`%c`	character

- Format ***Modifier :***

```
int a=15;
float b=3.141122;
printf(“a=%8i”,a);
printf(“b=%1.3f”,b);
```

Minimum width of the field for output

Output: a=......15

Output: b = 3.141

Before . ➔ minimum width of the field for output

After . ➔ number of digits to be displayed after the .

Basic *Input* Function: `scanf`

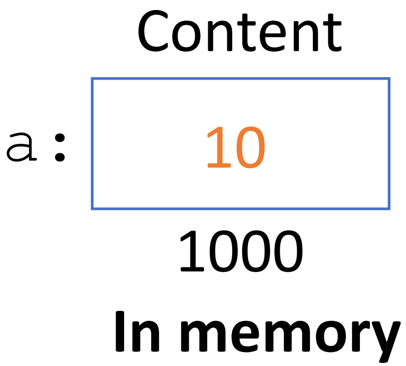
- `scanf` => read input from a keyboard

- Syntax: `scanf("Format String", &VariableName);`
  - `Format String` – specify type of data inputted (same as `printf`)
  - `&VariableName` – variable names preceded with the address of the operator (&).

- Example:

```
int a;
scanf("%i",&a);
```

Store value inputted in address 1000



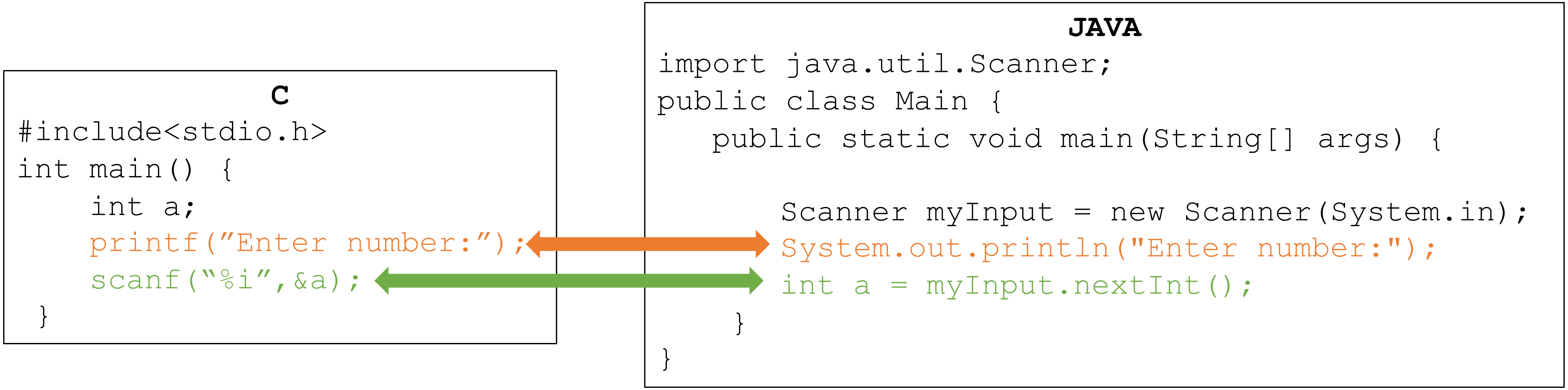
Content

a: 10

1000

In memory

C and Java: ***I/O function*** Comparison



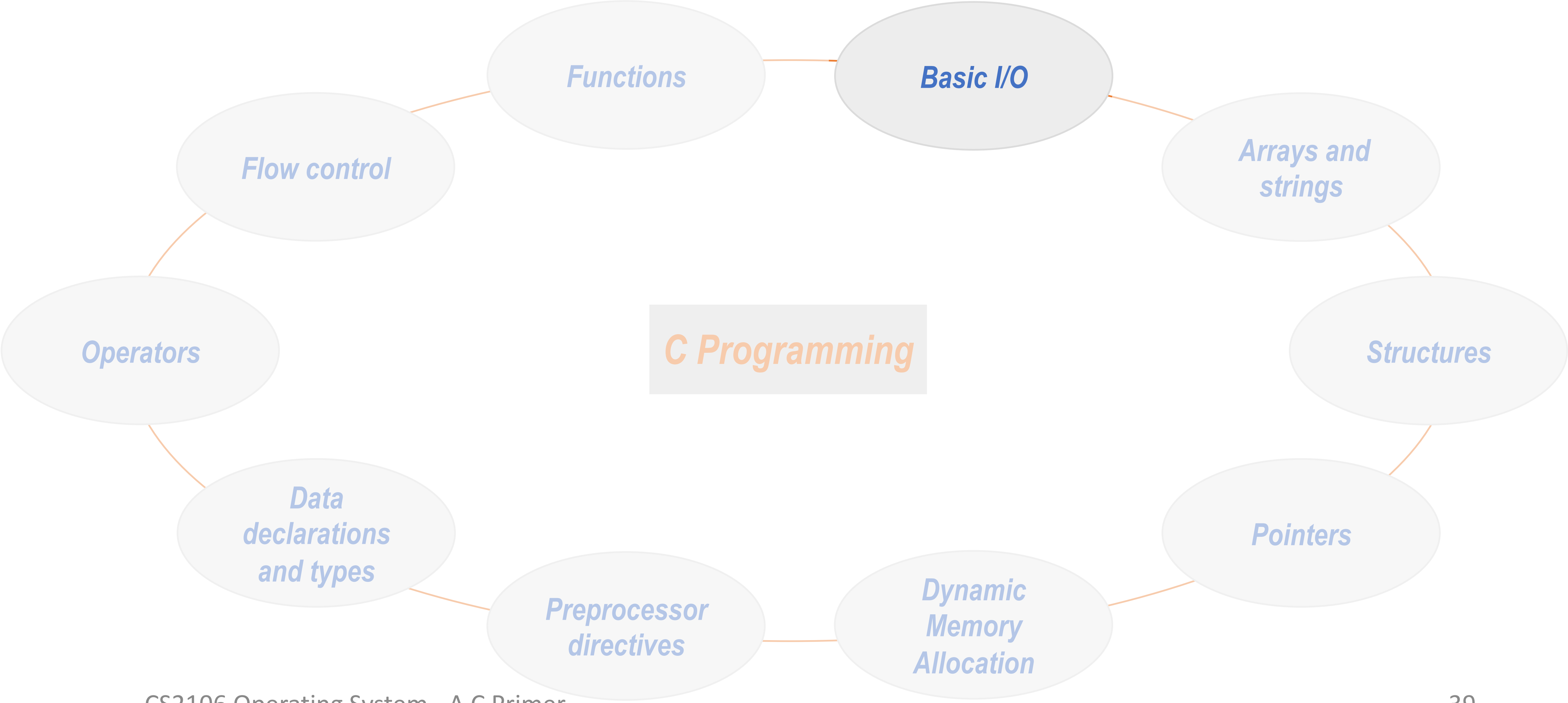
C

```
#include<stdio.h>
int main() {
    int a;
    printf(”Enter number:”);
    scanf(“%i”,&a);
 }
```

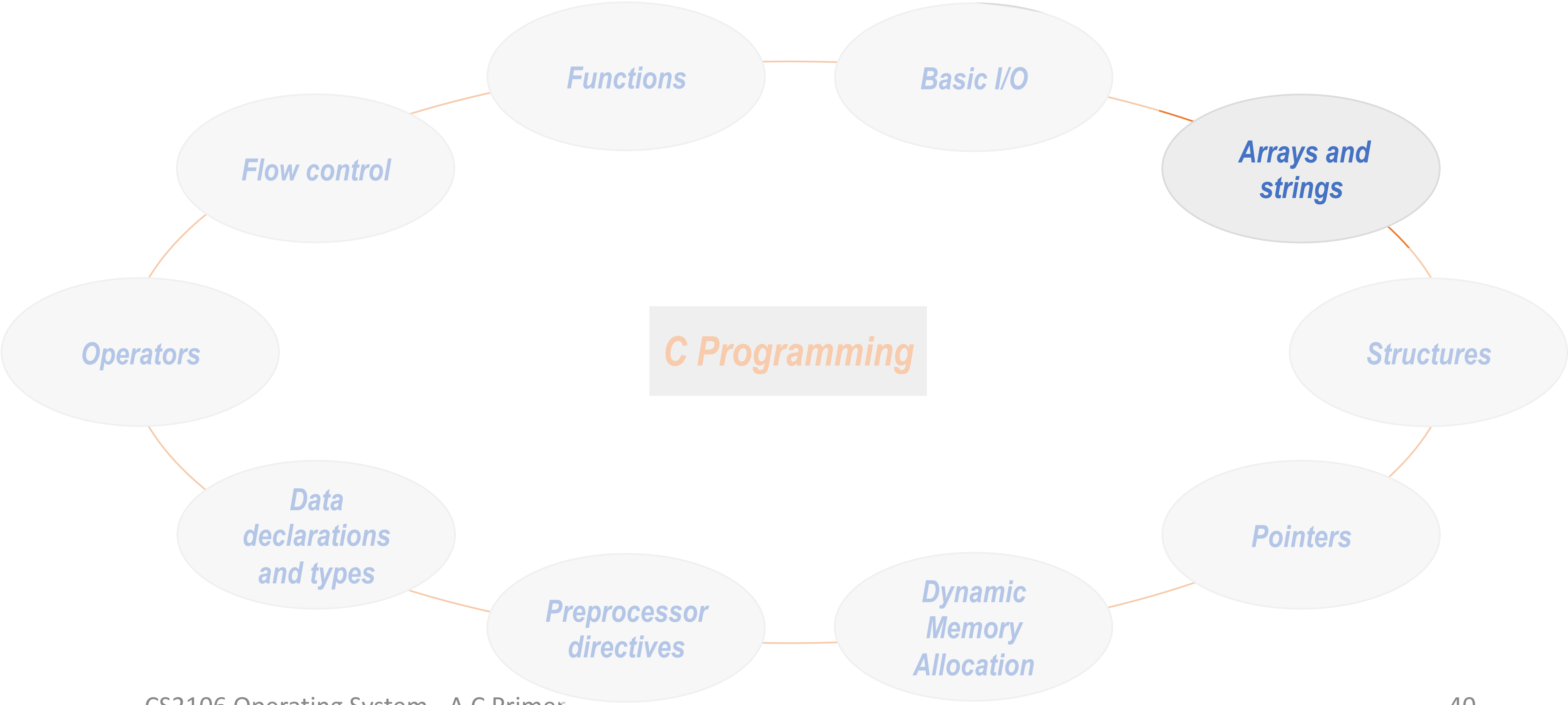
JAVA

```
import java.util.Scanner;
public class Main {
   public static void main(String[] args) {

       Scanner myInput = new Scanner(System.in);
       System.out.println("Enter number:");
       int a = myInput.nextInt();
    }
}
```

Overview


Functions
Basic I/O
Arrays and strings
Flow control
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Overview


Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control
C Programming

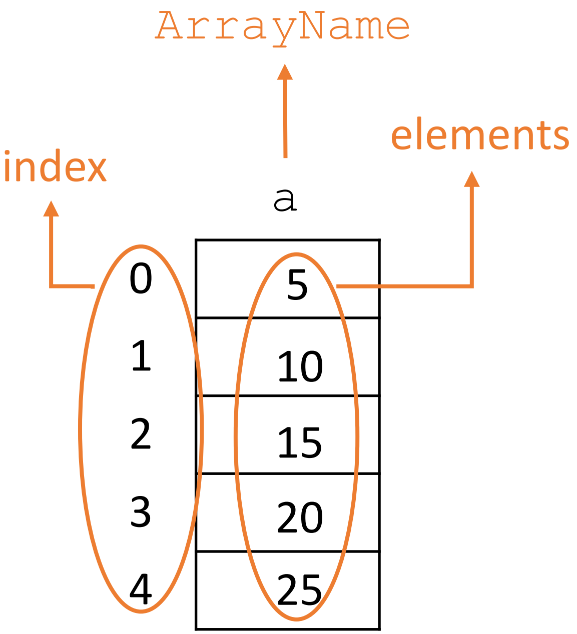
Arrays

- ***Collection*** of data items of the ***same type*** stored in ***contiguous memory location***
- Used to store ***more than one value*** at a time in a variable
- ***Declaration syntax:***

```
DataType ArrayName[Size];
int a[5];
```

- ***Initialization*** during declaration :

```
int a[5] = {5,10,15,20,25};
```



Array *Usage*

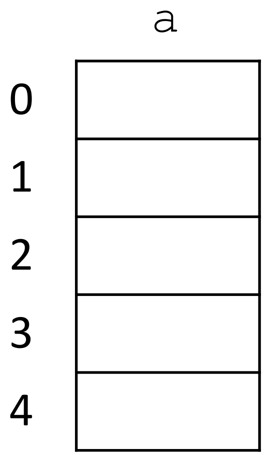
- Elements of an array can be ***accessed using index***

Sum of five elements

```
    Sum = a[0]+a[1]+a[2]+a[3]+a[4];
```

- ***Inputting/outputting*** an array:

```
int a[5];
printf(“Enter 5 values:”);
for(int i=0;i<5;i++){
     scanf(“%i”,&a[i]);
     printf(“%i”,a[i]);
}
```



Characteristics of an Array

- ***Size of array*** must be given in the declaration

```
DataType ArrayName[Size];
```

- ***Array index*** starts from ***0, number*** of elements = ***Size-1***

- ***Array to array*** assignment is ***NOT*** allowed:

```
int ia[5] = {1,2,3,4,5};
int ib[5];
ib = ia;
//compilation error
```

Need to ***assign element by element***

```
int ia[5] = {1,2,3,4,5};
int ib[5];
for(int i=0;i<5;i++){
     ib[i] = ia[i];
}
```

Strings

- An array of characters
- ***Declaration Syntax :***

```
char StringName[Size];
char str[5]= “abcd”;
```

- ***Reading*** a string:

```
scanf(”%s”,str);
```

Note the absence of &

  - Can read ***only a single word***
  - ***Alternative*** method :

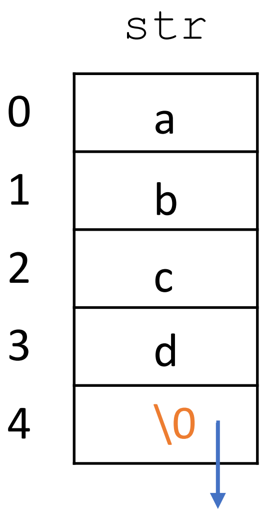
```
gets(str);
```

- ***Printing*** a string:

```
printf(”String=%s”,str);
```

  - Can print ***only a single word***
  - ***Alternative*** method:

```
puts(str);
```



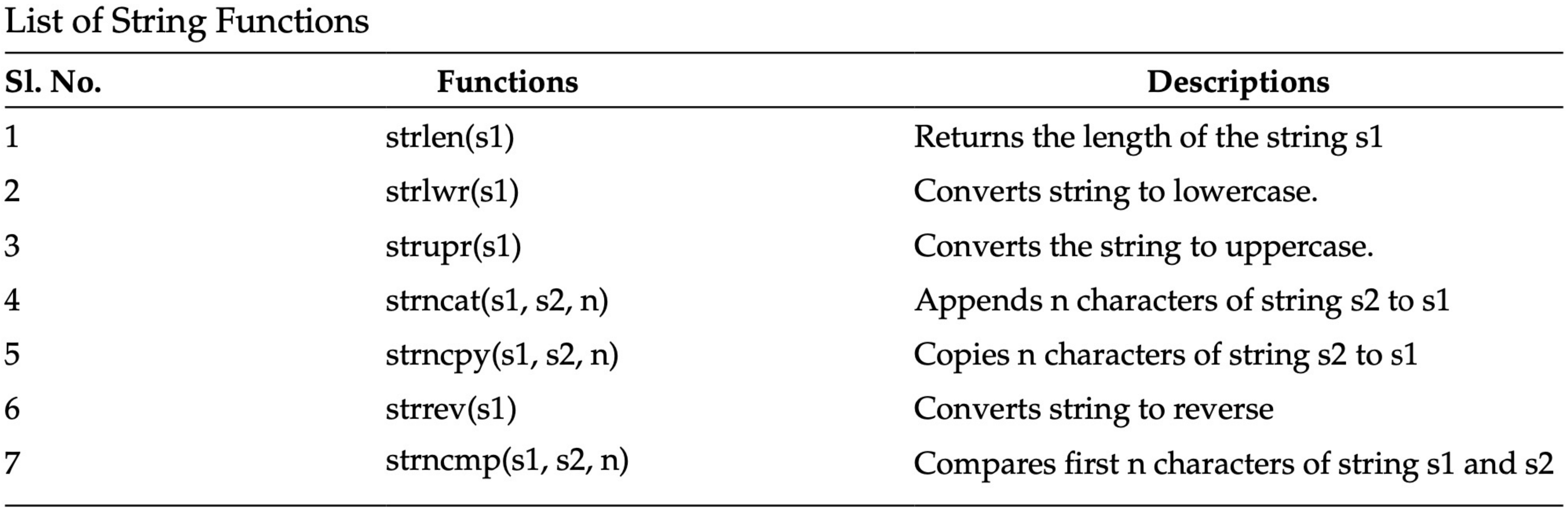
str

0	a
1	b
2	c
3	d
4	\0

Note the ‘\0’ (auto appended) – indicate the end of string

List of ***String Functions***

- Need to `#include<string.h>`



List of String Functions

Sl. No.	Functions	Descriptions
1	strlen(s1)	Returns the length of the string s1
2	strlwr(s1)	Converts string to lowercase.
3	strupr(s1)	Converts the string to uppercase.
4	strncat(s1, s2, n)	Appends n characters of string s2 to s1
5	strncpy(s1, s2, n)	Copies n characters of string s2 to s1
6	strrev(s1)	Converts string to reverse
7	strncmp(s1, s2, n)	Compares first n characters of string s1 and s2

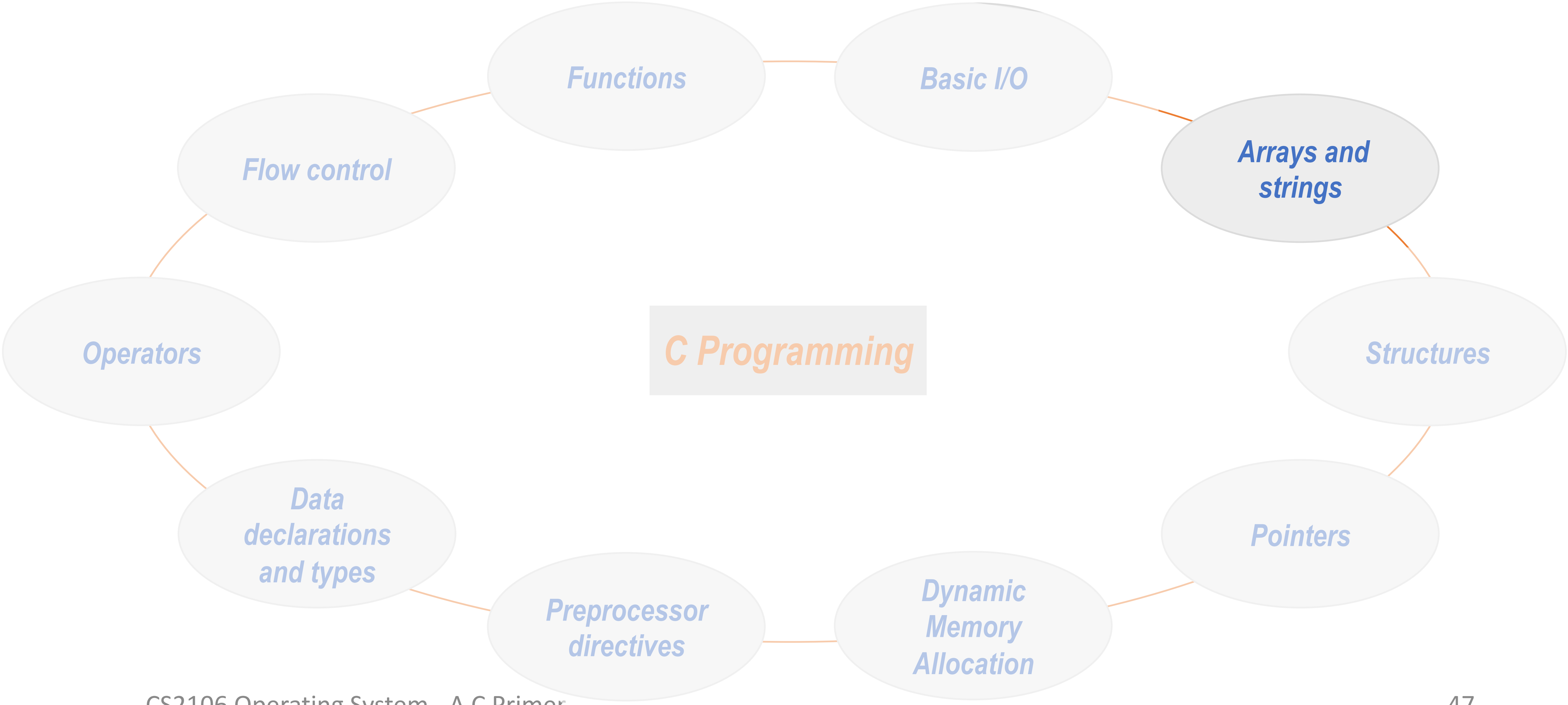
C and Java: ***Array Comparison***

- ***Array declaration***
  - C: `int a[10];`
  - Java: `int[] a = new int[10];`

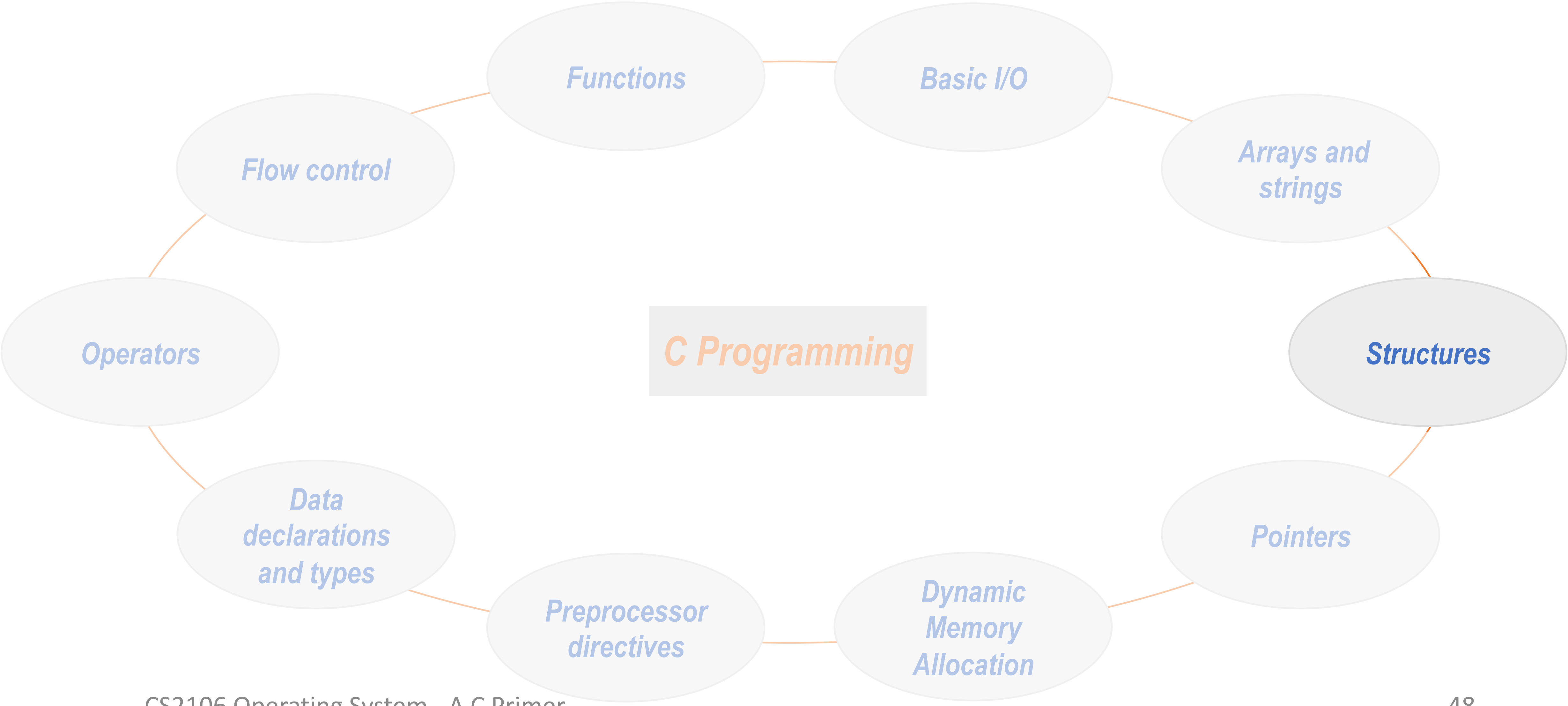
- ***Array behavior***
  - C: behave like a ***primitive data type***
  - Java: is an ***object*** and has in-build methods (length, toString())

- ***Array bound checking:***
  - C: does ***not do*** bound checking
  - Java: ***automatically*** check array bounds

Overview


Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control
C Programming

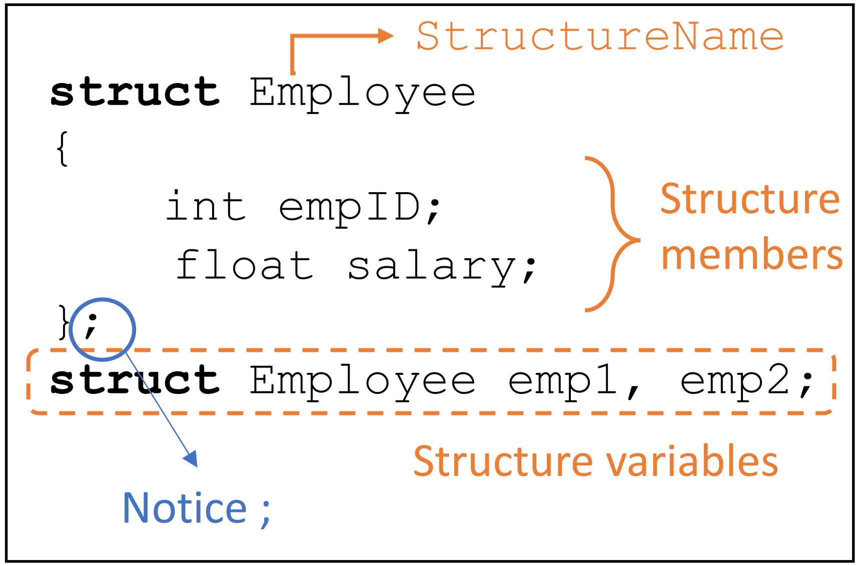
Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Structures

- Group variables of ***different data types*** under a single name
- Helps organize data in C
- **Declaration Syntax:**

```
struct StructureName
{
   DataType VariableName 1;
   DataType VariableName 2;
            …
   DataType VariableName n;
};
struct StructureName var1,…;
```



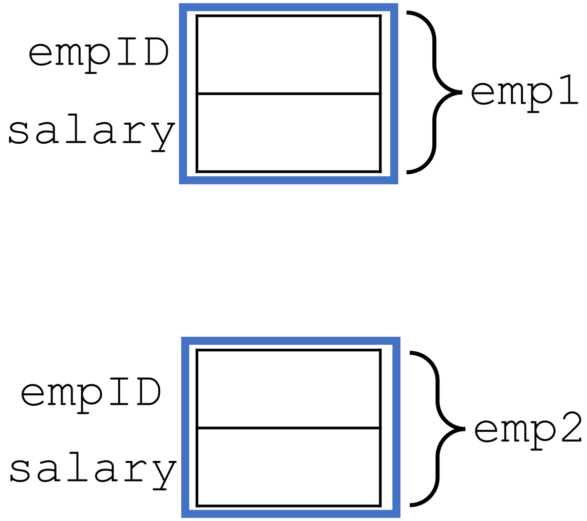
```
struct Employee
{
     int empID;
     float salary;
};
struct Employee emp1, emp2;
```

StructureName

Structure members

Structure variables

Notice ;

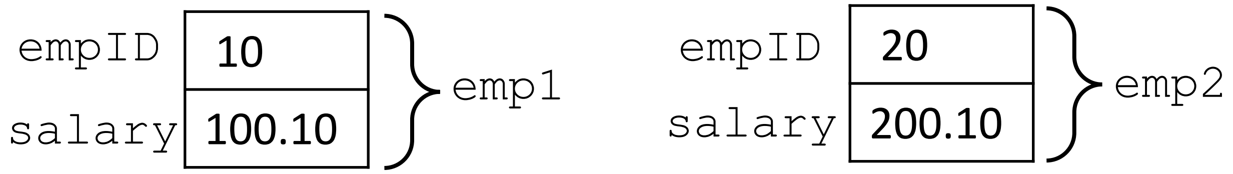


Working With Structures

- ***Assigning*** values:

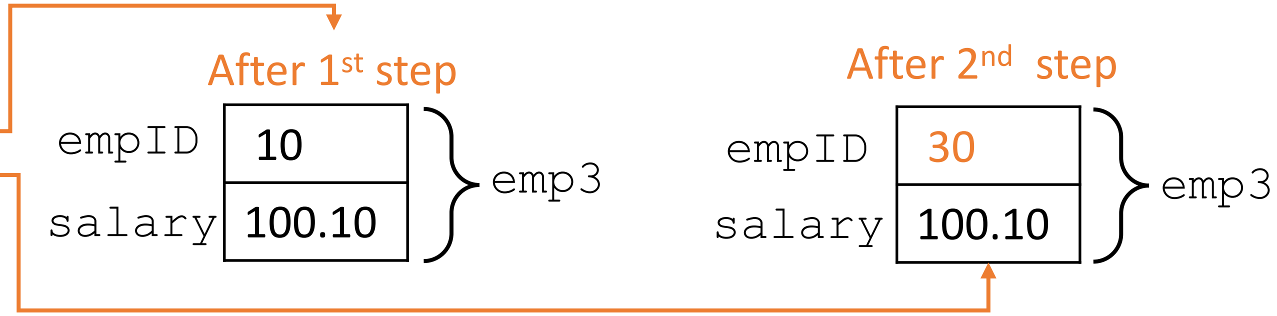
Dot operator

```
emp1.empID = 10;
emp1.salary = 100.10;

emp2.empID = 20;
emp2.salary = 200.10;
```



- Can ***copy structure variable*** to another (unlike arrays):

```
struct Employee emp3=emp1;
emp3.empID = emp1.ID + 20;
```



Input/Output a Structure

```
struct Employee
{
    int empID;
    float salary;
};
struct Employee emp1, emp2;
```

Input and print emp1 details

```
scanf(“%i”,&emp1.empID);
scanf(“%f”,&emp1.salary);

printf(“%i”,emp1.empID);
printf(“%u”,emp1.salary);
```

Input and print emp2 details

```
scanf(“%i”,&emp2.empID);
scanf(“%f”,&emp2.salary);

printf(“%i”,emp2.empID);
printf(“%i”,emp2.salary);
```

If want to store details of 100 employee?

```
struct Employee
{
    int empID;
    float salary;
};
struct Employee emp[100];
```

Array of structures

Structures and ***Functions***

```
#include <stdio.h>
#include <stdlib.h>
struct Employee
{
    int empID;
    float salary;
};   Returning structure variable          Passing structure variable
struct Employee UpdateSalary(struct Employee emp1);
int main() {
   struct Employee emp1;
   emp1.empID = 10;
   emp1.salary = 100.10;
   emp1 = UpdateSalary(emp1);   Function makes a copy of emp1
   return 0;
}
struct Employee UpdateSalary(struct Employee emp1){
    emp1.salary = emp1.salary  + 20;
    return emp1;
}
```

Passing Structure as Parameter: Pass by ***Value***

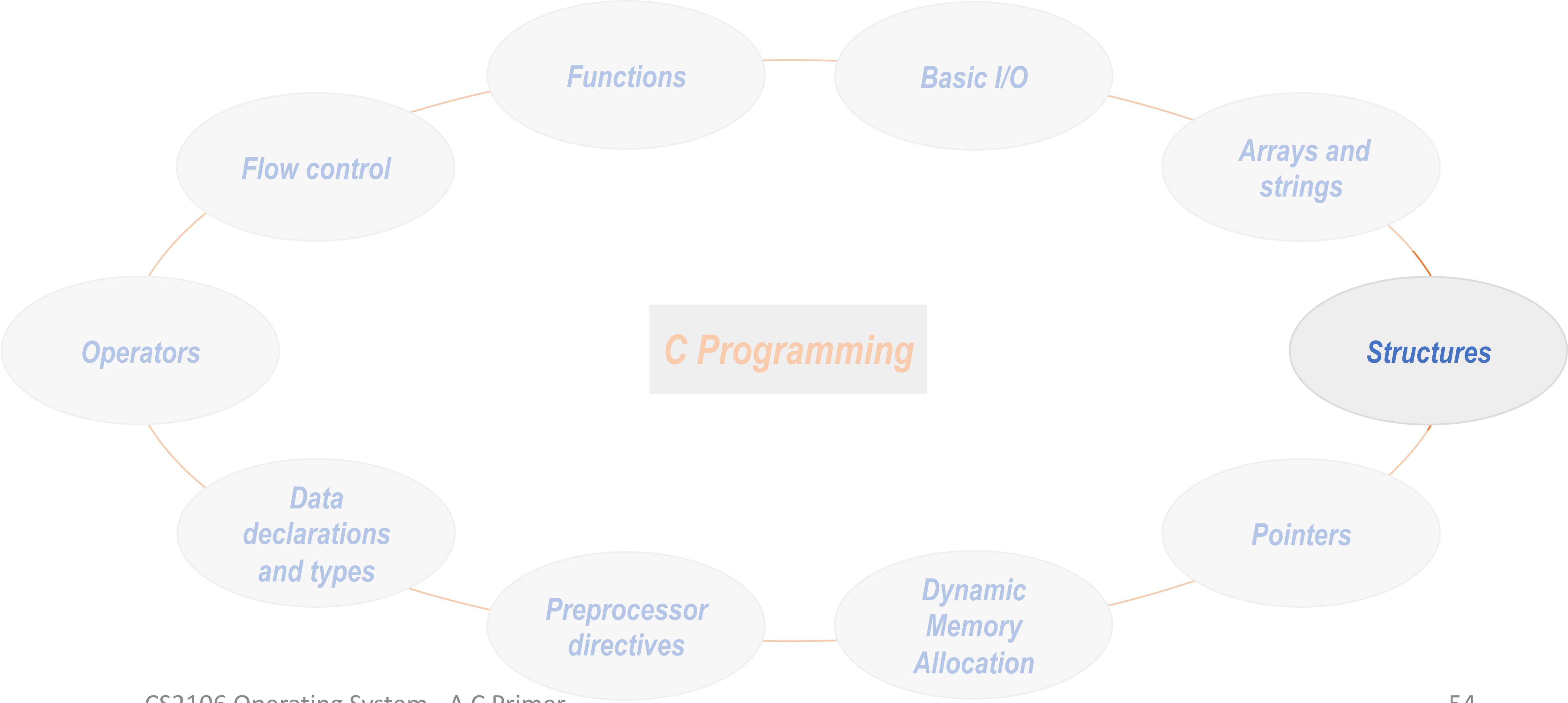
C and Java: ***Structure Comparison***

- Similar to `Class` in Java
  - Helps to organize data
- Different from `Class` in Java
  - ***Does not use*** new keyword
  - ***No function*** associated with a structure
  - All members are ***public***

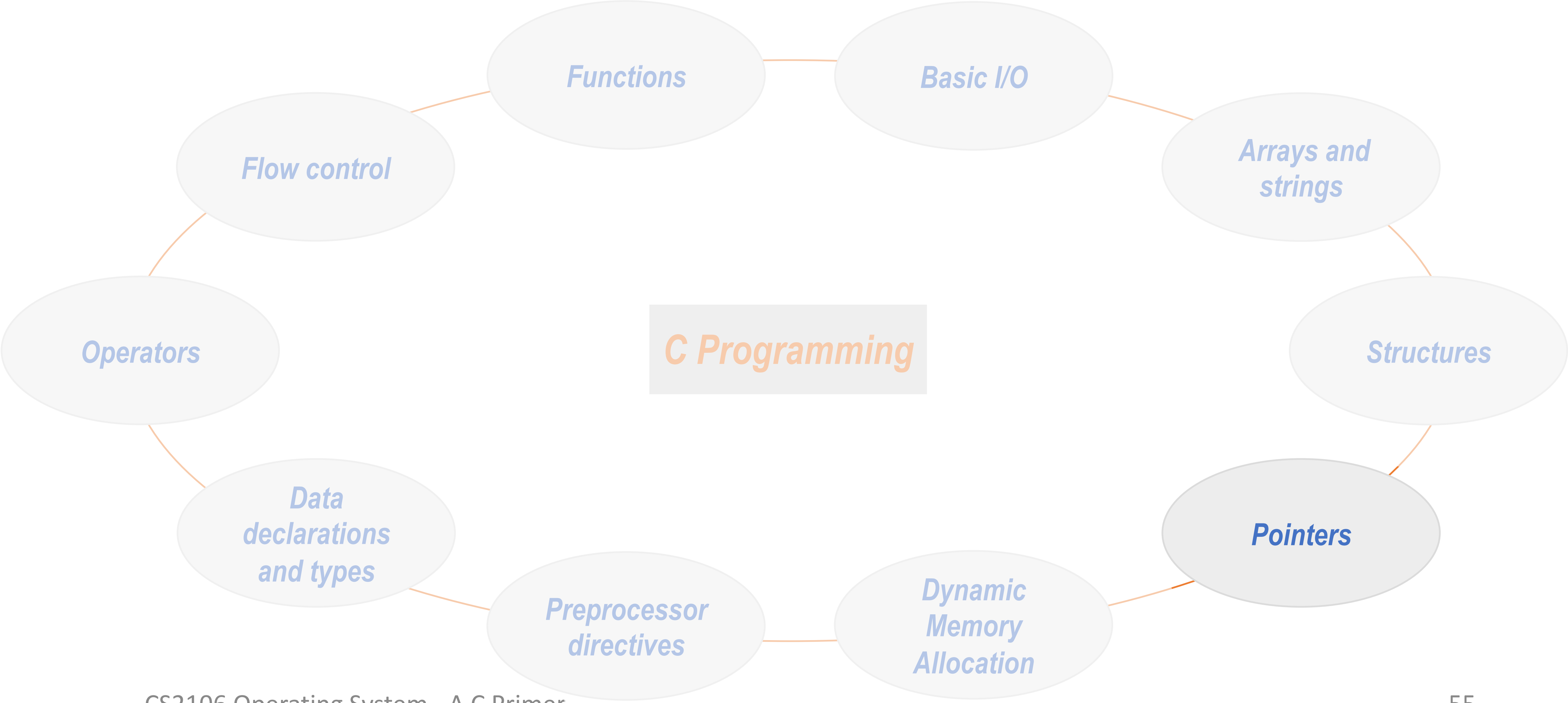
```
struct Employee
{
     int empID;
     float salary;
};
struct Employee emp;
```

```
class Employee
{
     int empID;
     float salary;
     float calculatePay() {…}
};
Employee emp = new Employee();
```

Overview


C Programming
Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control

Overview


C Programming
Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control

Pointers

- Variable that represents a ***location in memory*** rather than a value – stores ***address of another variable***

- ***Indirect means of accessing the value*** of a particular data type

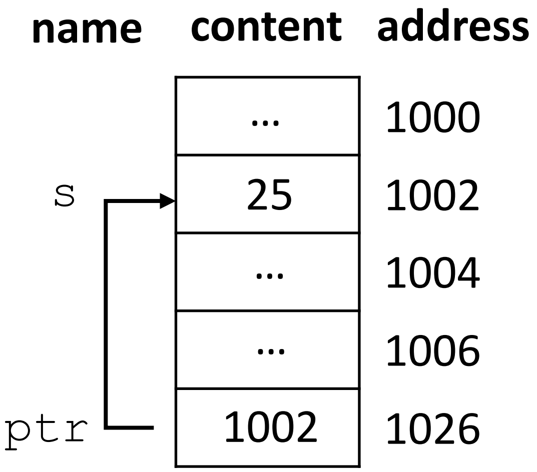
- ***Declaration Syntax:***

```
DataType *PointerVairable;
int *ptr;
```

- **Example:**

```
int s = 25;
int *ptr;
ptr = &s
```

<u>Address-of</u> operator – gives the memory address of the s



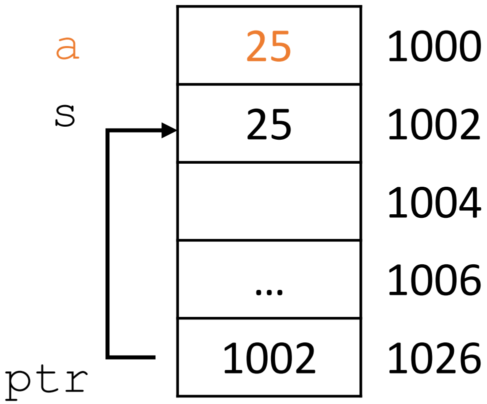
name	content	address
	…	1000
s	25	1002
	…	1004
	…	1006
ptr	1002	1026

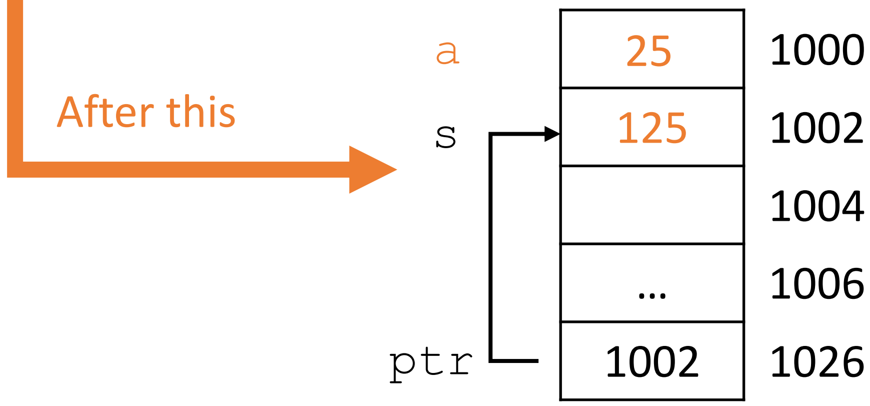
Pointer Variable ***Dereferencing***

- ***Dereferencing*** or accessing a value:

```
int s = 25;
int *ptr;
ptr = &s
int a = *ptr;
*ptr = 125;
```

Indirection operator – gives the value stored in the pointed address

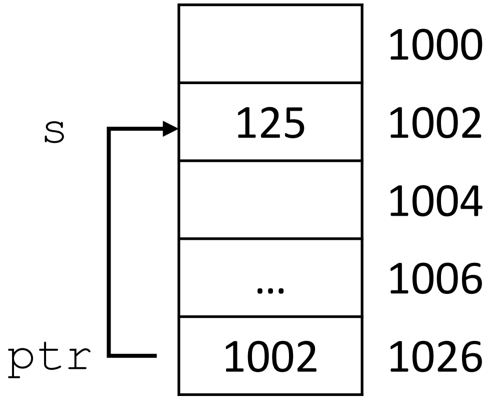




- Note * operators for
  - ***Declaration*** of pointers
  - ***Dereferencing*** the pointers

What is the ***Output?***

```
int s = 25;
int *ptr;
ptr = &s;
*ptr = 125;
```



```
printf(“s=%i”,s);
printf(“&s=%u”,&s);
printf(“ptr=%u”,ptr);
printf(“*ptr=%i”,*ptr);
printf(“&ptr=%u”,&ptr);
```

s=125 (content of s)

&s=1002 (address of s)

ptr=1002 (content of ptr)

*ptr=125 (value pointed by ptr)

&ptr=1026 (address of ptr)

Pointers and ***Operators***

- ***Addition and subtraction*** are valid operations on pointers
- ***Example***: Memory snapshot

Order of 2 bytes since int requires 2 bytes – auto handled by compiler

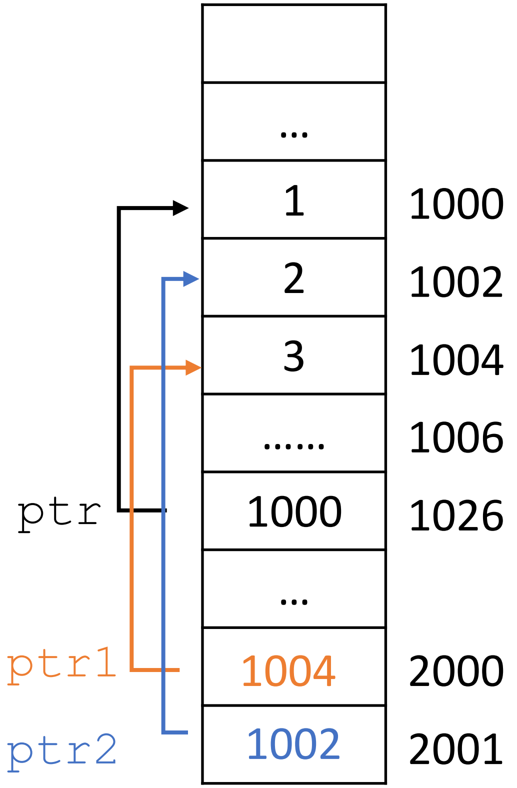
```
int *ptr1, *ptr2;
ptr1 = ptr+2;
ptr2 = ptr1-1;

printf("*ptr1=%u",*ptr1);
printf("*ptr2=%u",*ptr2);
```

=> 1000 + 2 = 1004

=> 1004 -1 = 1002

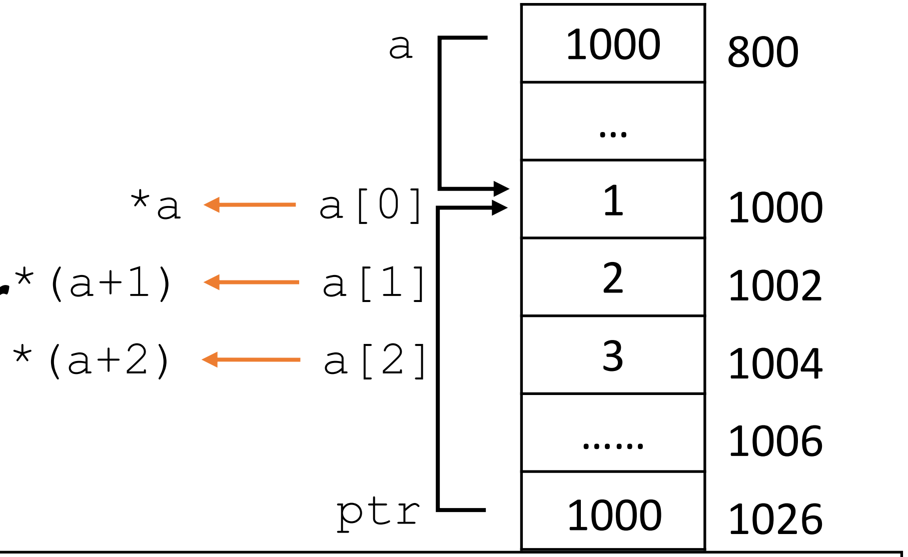
*ptr1=3

*ptr2=2



	…	
	1	1000
	2	1002
	3	1004
	……	1006
ptr	1000	1026
	…	
ptr1	1004	2000
ptr2	1002	2001

Pointers and ***Arrays***



- Array name is itself a ***constant pointer***
  - Value cannot be changed
- **Example:**

```
int a[3] = {1,2,3};
int *ptr = a;
a = a+1;                           Error       (a is constant pointer)
printf(“a=%u”,a);                  a=1000      (address of a[0])
printf(“a=%u”,&a[0]);              a=1000      (address of a[0])
printf(“*&a=%i”,*(&a[0]));         *&a=1       (content of a[0])
printf(“ptr=%u”,ptr);              ptr=1000    (content of ptr)
printf(“*ptr+1=%i”,*ptr+1);        *ptr+1=2    (value pointed by ptr +1)
```

Pointers and **Structures**

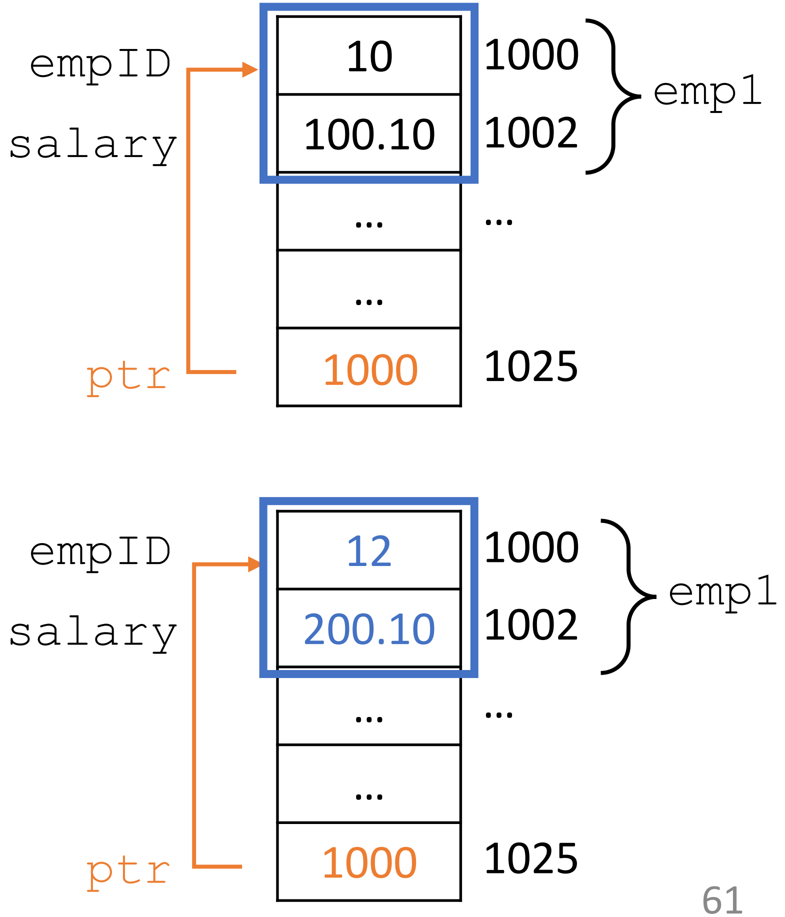
- ***Structure variable*** can be a pointer

```
struct Employee
{
     int empID;
      float salary;
};
struct Employee emp1;
emp1.empID = 10;
emp1.salary = 100.10;

struct Employee *ptr;
ptr = &emp1

ptr ->empID = 12;
ptr ->salary = 200.10;
```

Point to address of the first member

After this

arrow operator to access the member of a structure.

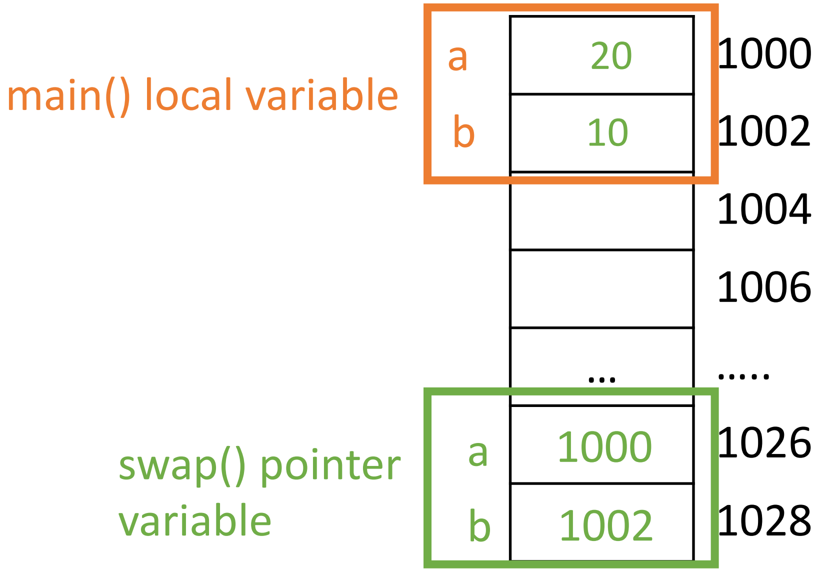


Pointer and ***Functions***: ***Pass by Reference***

- Pointers used for ***passing parameters*** to functions
- Function creates a ***reference*** for parameters
- ***Original*** parameter value will be ***affected***

```
void swap(int *a, int *b);
int main(){
   int a=10, b=20;
   swap(&a,&b);
}
void swap(int *a, int *b){

      int *t;
      *t=*a;
      *a=*b;
      *b=*t;
}
```

When swap() function is called, main() variable a, b swaps



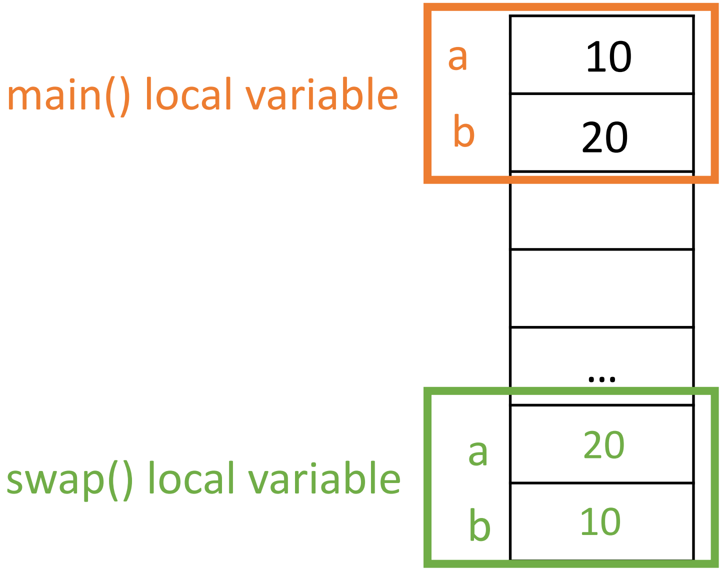
Note: A way to ***return multiple values*** from functions

(RECALL: ***Pass by Value)***

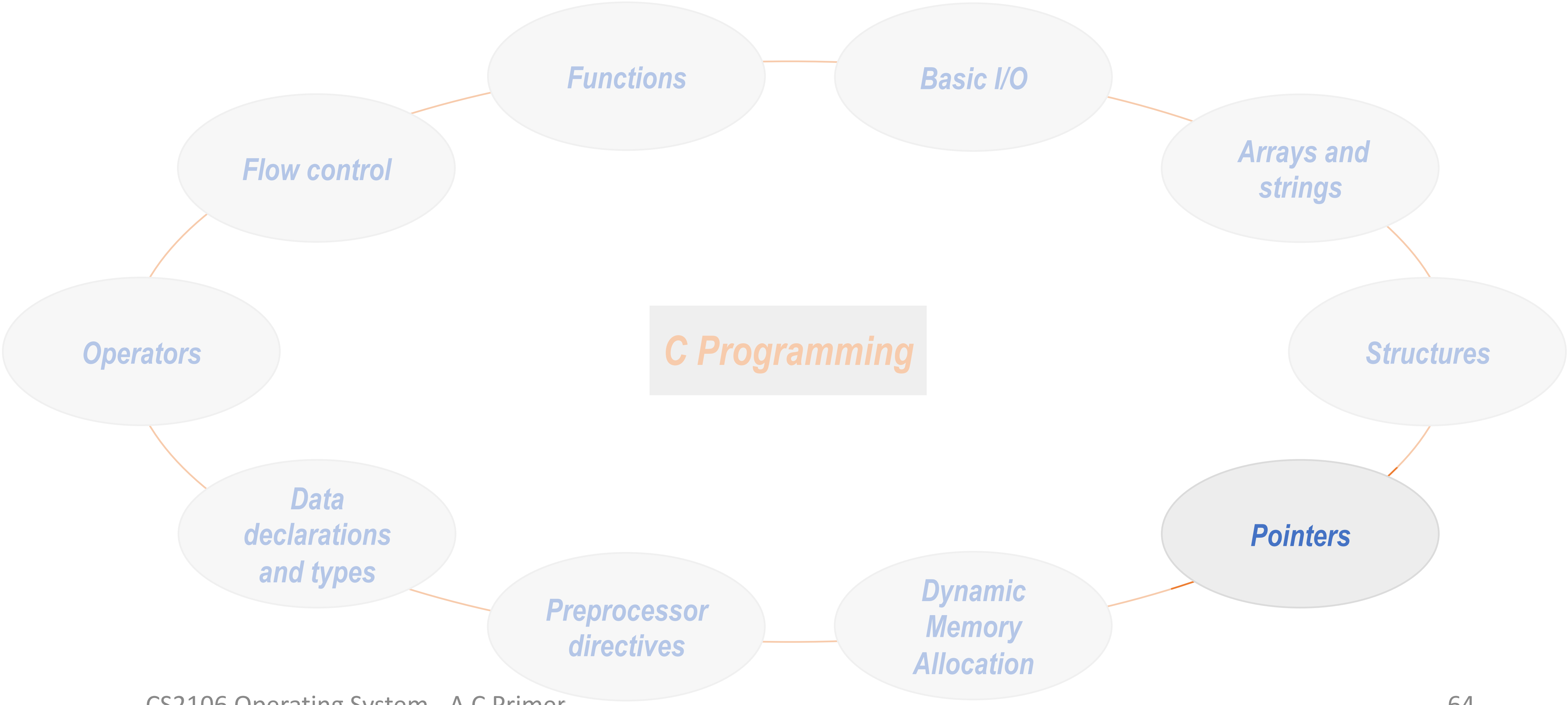
- Function ***makes a copy*** of the parameters and works on that copy.
- ***Actual*** parameter values are ***unaffected***

```
void swap(int a, int b);
int main(){
   int a=10, b=20;
   swap(a,b);
}
void swap(int a, int b){

      int t;
      t=a;
      a=b;
      b=t;
}
```

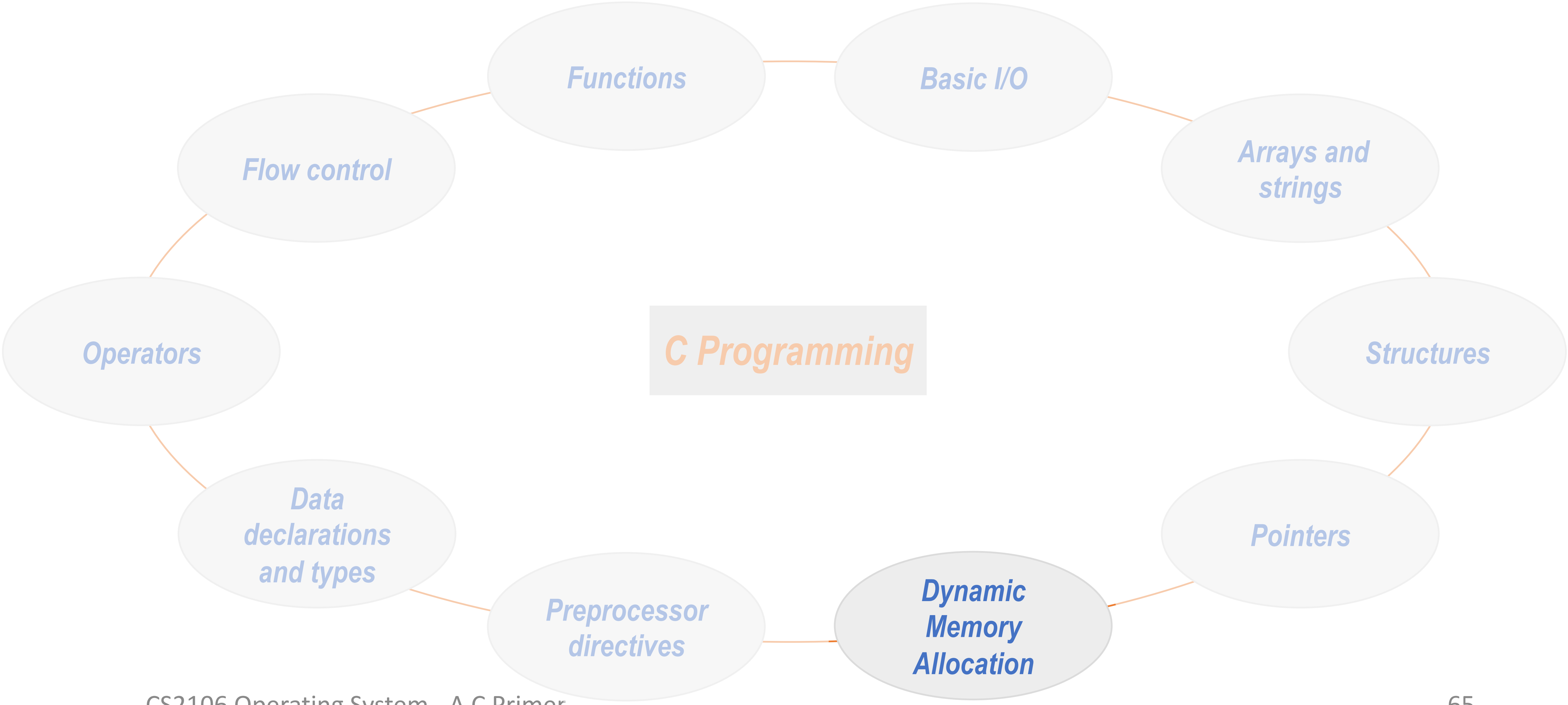
When swap() function is called, only local variable a, b swaps



Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Overview



Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Dynamic Memory Allocation

- **Static memory allocation**
  - Memory allocated at ***compile time***
  - ***Cannot modify*** its size – possibility of ***memory wastage***
  - Better way is to allocate memory ***at run time,*** when we know ***exact requirement***
- **Dynamic memory allocation**
  - Allocation of memory at ***run time***
  - Allocate memory based on the ***requirement only***
- ***Usage:*** when ***amount of memory*** required is ***unknown*** during ***compile time***

Memory Allocation Function: `malloc`

- ***Reserve*** memory and ***returns* address** of the newly allocated memory locations
- ***Header file*** - `#include<stdlib.h>`
- ***Declaration:***

```
Prtvariablename = (DataType*)malloc(size);
```

Pointer variable of type `DataType`

Cast void pointer to type `DataType`

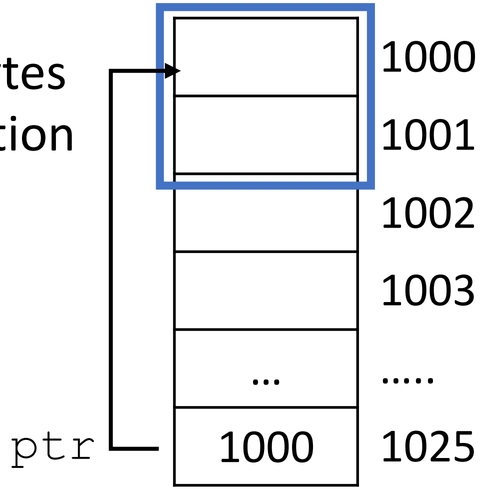
No. of bytes to allocation

- ***Example:***

Allocate memory ***for one integer***

return the size of the integer (2 bytes)

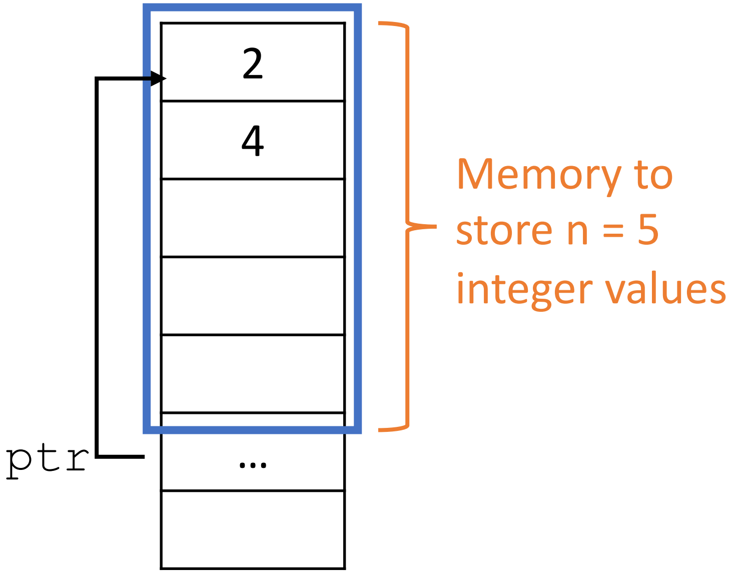
```
int *ptr = (int*)malloc(sizeof(int));
```



Dynamic Allocation for ***Array***

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

int main() {
    int *ptr,n;
    printf(“Enter size of array:”);
    scanf(“%i”,&n);
    ptr=(int*)malloc(n*sizeof(int));
    if(ptr!=NULL){
        printf("Allocation Successful");
        *ptr = 2;
        *(ptr+1) = 4;
    }
    else
        printf("! Allocation Unsuccessful");
    return 0;
}
```

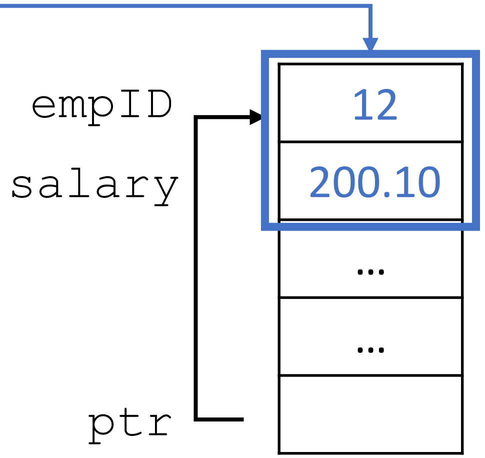
Size obtained during run time



Dynamic Allocation for *Structure*

```
#include <stdio.h>
#include <stdlib.h>
struct Employee
{
    int empID;
    float salary;
};
int main() {
    struct Employee *ptr;
    ptr=(struct Employee*)malloc(sizeof(struct Employee));
    if(ptr!=NULL){
       printf("Allocation Successful");
       ptr->empID = 12;
       ptr->salary = 200.10;
    }
    else
       printf("! Allocation Unsuccessful");
    return 0;
}
```

Allocate memory to store 1 struct Employee



Memory ***Deallocation***: `free()`

- Deallocation of memory is necessary to ***optimize memory usage***
- `free()` function to ***deallocate the memory*** space allocated by `malloc()`

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

int main() {
    int *ptr;
    ptr=(int*)malloc(5*sizeof(int));
    *ptr = 2;


}
```

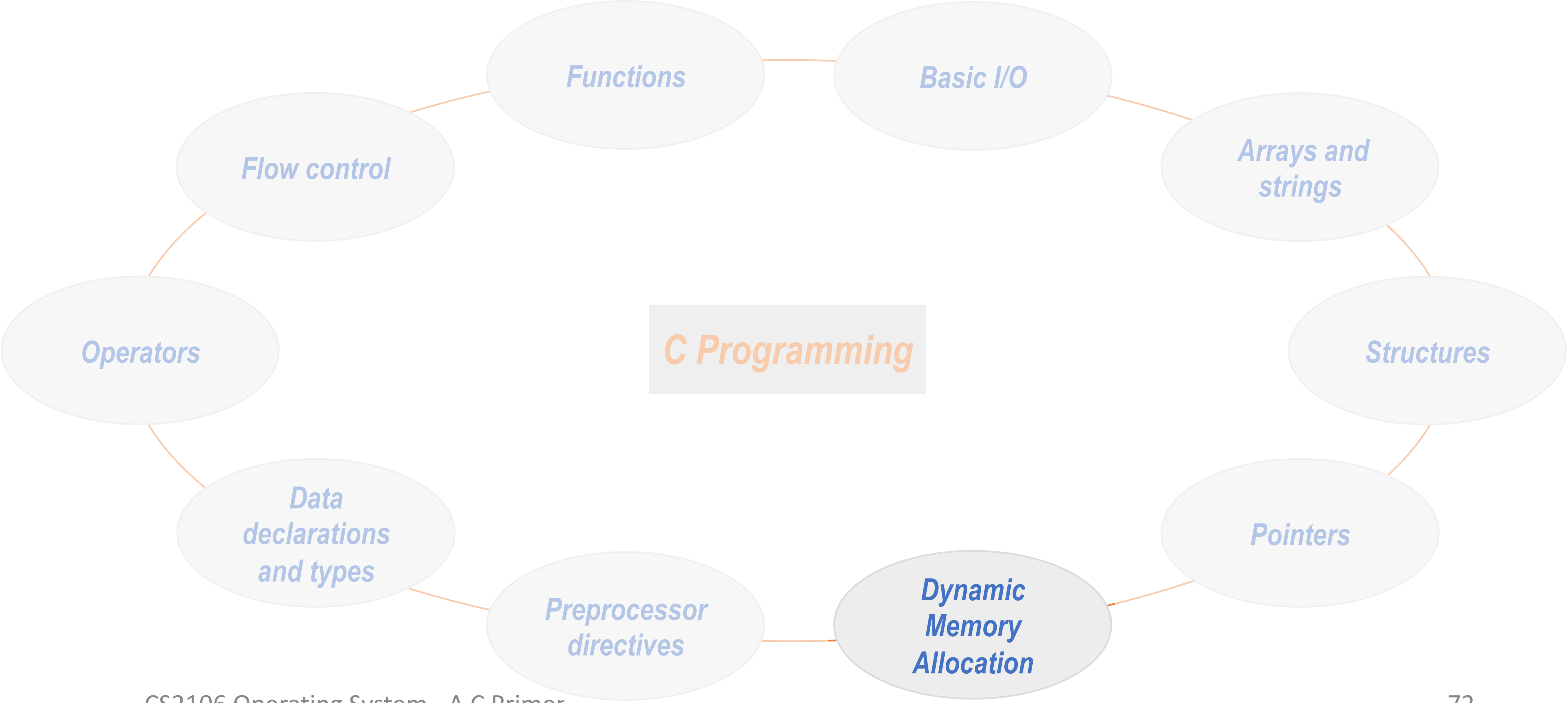
C and Java: ***Pointer Comparison***

- ***Pointers*** in C is ***similar*** to ***references*** in Java
  - C: `emp=(struct Employee*)malloc(sizeof(struct Employee));`
  - Java: `Employee emp = new Employee();` (for Employee class)

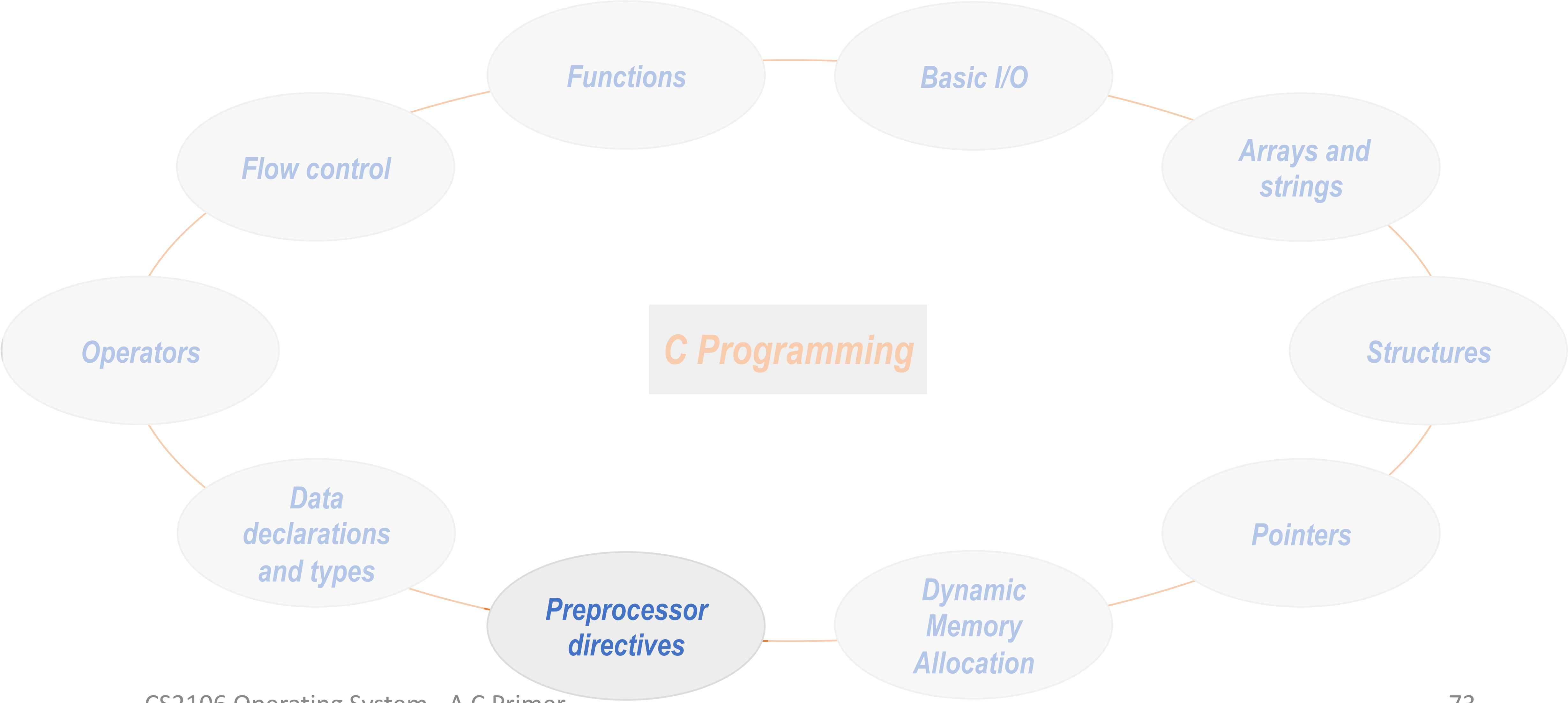
- Memory location ***access***
  - C: ***can access*** memory locations and perform ***pointer arithmetic***
  - Java: memory locations are ***hidden***

- Memory ***clean up***
  - C: need ***explicitly deallocate*** memory using `free()`
  - Java: ***automatic*** garbage collection

Overview


C Programming
Functions
Basic I/O
Arrays and strings
Structures
Pointers
Dynamic Memory Allocation
Preprocessor directives
Data declarations and types
Operators
Flow control

Overview


Functions
Basic I/O
Flow control
Arrays and strings
Operators
C Programming
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

The ***Pre-Processor***

- Pre-processing is ***executing*** some special statements ***before actual compilation***
- Included inside a directive called a ***pre-processor directive*** => ***#***
- ***Categories*** of directives :
  - ***Include file:*** includes the content of a file (`#include`)
  - ***Macro:*** assign a symbolic name to a constant (`#define`)
  - ***Conditional pre-processors:*** used for assigning conditions, whether to execute a line of code or not (`#ifdef, #else, #endif`)

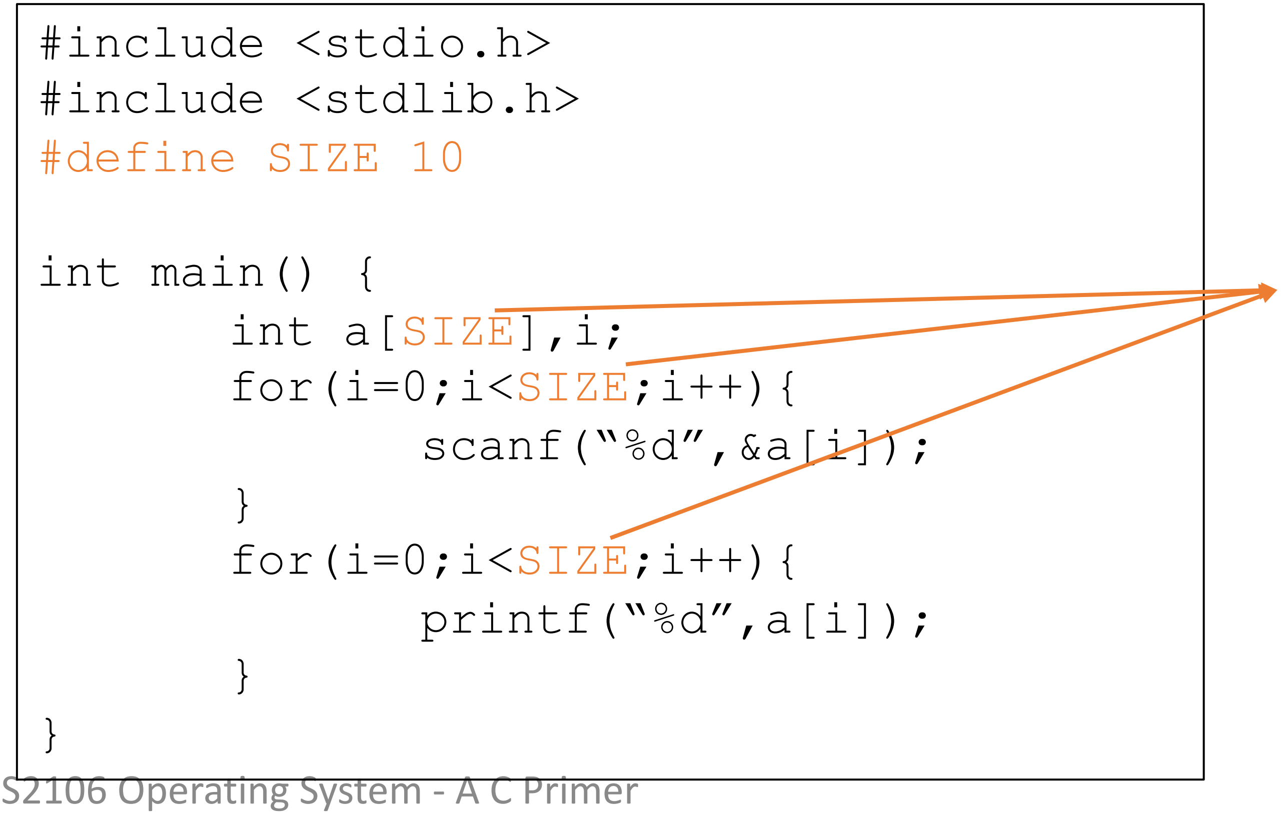
Pre-Processor Directive: `#include`

- ***Two ways*** to include files
  - `#include<XXX.h>`
    - Includes a ***standard*** C library
    - Present in the ***pre-defined directory***
  - `#include “XXX.h”`
    - Includes a ***user-defined*** header files
    - Present in the ***local directory***

Pre-Processor Directive: `#define`

```
#define MACRONAME MacroValue
```

Find occurrences of `MACRONAME` and replace with `MacroValue`



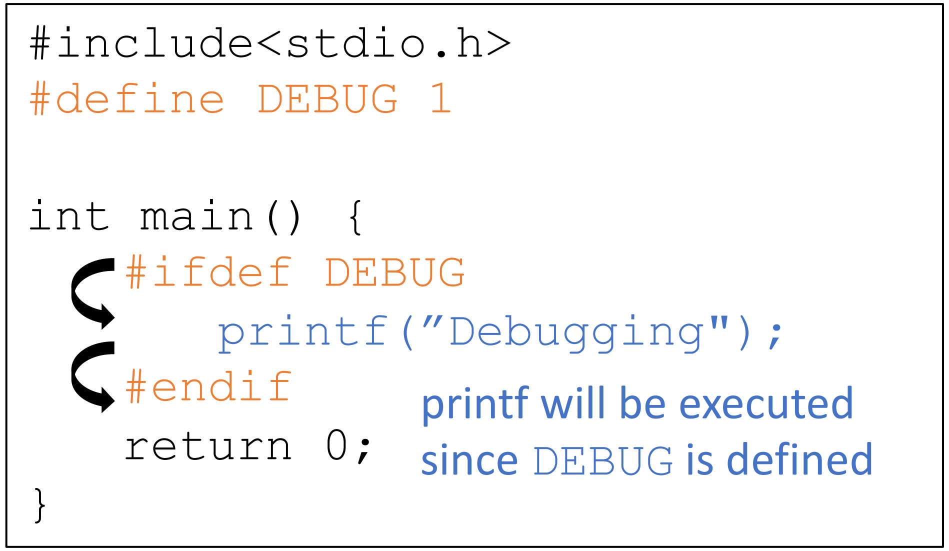
```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

int main() {
        int a[SIZE],i;
        for(i=0;i<SIZE;i++){
                scanf("%d",&a[i]);
        }
        for(i=0;i<SIZE;i++){
                printf("%d",a[i]);
        }
}
```

SIZE used in three places – will be substituted by 10 before program compiles

Adv: If you want change `SIZE`, need to do it only in one place, not three places -> easy to maintain

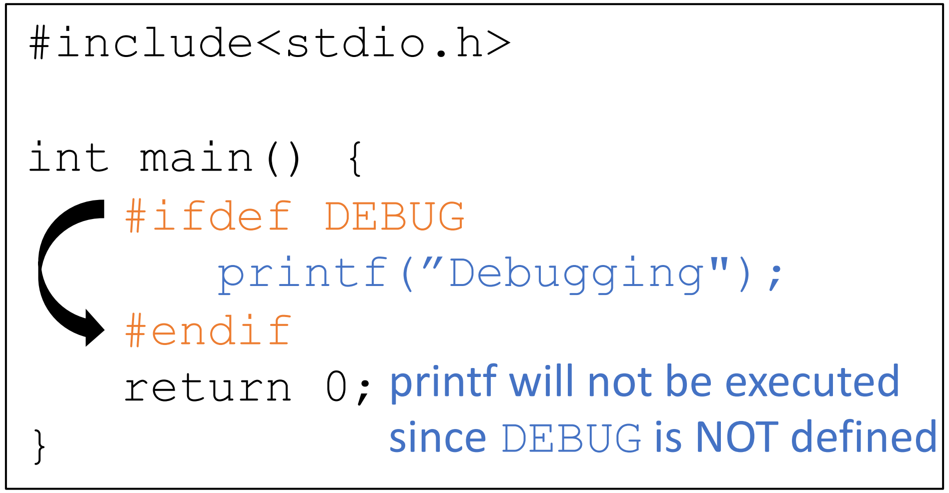
Pre-Processor Directive: `#ifdef, #endif`

- ***Control execution*** of program statements based on conditions: ***presence/absence of certain macro-definitions***



```
#include<stdio.h>
#define DEBUG 1

int main() {
  #ifdef DEBUG
      printf(”Debugging");
  #endif
  return 0;
}
```

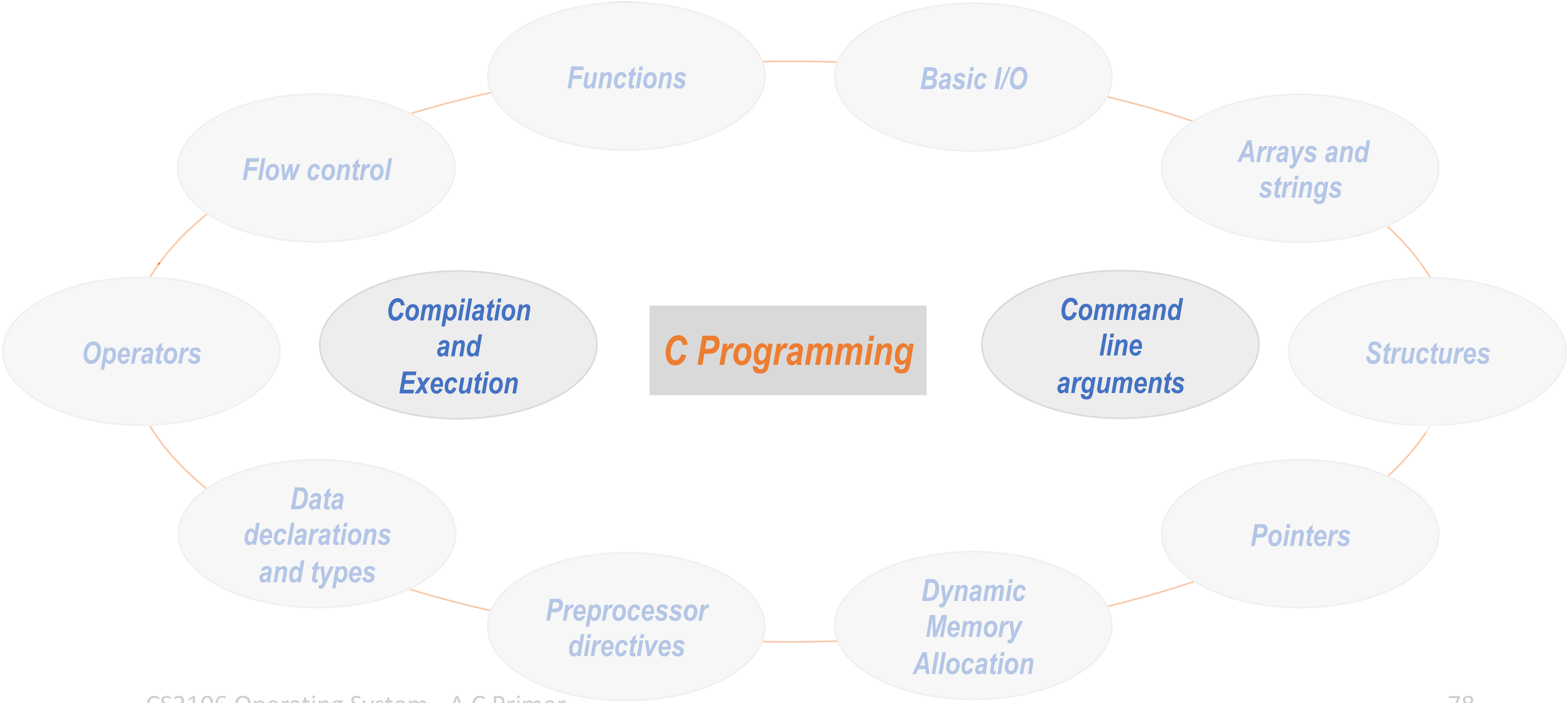
printf will be executed since DEBUG is defined



```
#include<stdio.h>

int main() {
  #ifdef DEBUG
      printf(”Debugging");
  #endif
  return 0;
}
```

printf will not be executed since DEBUG is NOT defined

Overview



Functions
Basic I/O
Flow control
Arrays and strings
Compilation and Execution
C Programming
Command line arguments
Operators
Structures
Data declarations and types
Pointers
Preprocessor directives
Dynamic Memory Allocation

Compiling C Programs

- ***Single file compilation***: `gcc filename.c`

- Compile and execute with ***warnings enabled***:`gcc -Wall filename.c`
  `./a.out`

- Produce a ***custom executable name***: `gcc filename.c -o filename`
  `./filename`

- ***Multiple files***: `gcc filename1.c filename2.c filename3.c`

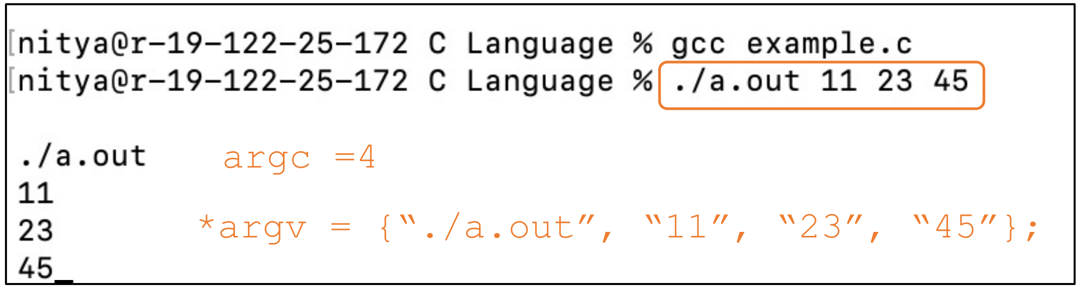
Command Line Arguments

- Supplying ***parameters*** through the ***command prompt*** to `main()`
- ***Syntax:*** `main(int argc, char *argv[])`
  - *`argc`*: ***total number of parameters*** passed through the command prompt
  - *`argv`*: a pointer to an ***array of strings***, points to the ***parameters passed***

example.c

```c
#include<stdio.h>
int main(int argc, char *argv[]) {
    int i;
    for(i=0;i<argc;i++)
        printf("\n%s",argv[i]);
    return 0;
}
```

Command prompt



```
[nitya@r-19-122-25-172 C Language % gcc example.c
[nitya@r-19-122-25-172 C Language % ./a.out 11 23 45

./a.out     argc =4
11
23         *argv = {“./a.out”, “11”, “23”, “45”};
45_
```

References

- C programming for beginners learn to code, Sisir Kumar Jena

- The C Programming Language, 2nd Edition Ritchie Kernighan

THANK YOU