

# C Primer

A crushing crash course / revision on C

# Overview

- The C Programming Language
  - Brief History
  - Programming Model
- Major C Programming Constructs
  - Control Statements
  - Data Types
    - Pointers
  - Function
- Built-in C Libraries
  - Input/Output
  - String Manipulation
- Compilation Process
  - Preprocessing Macro

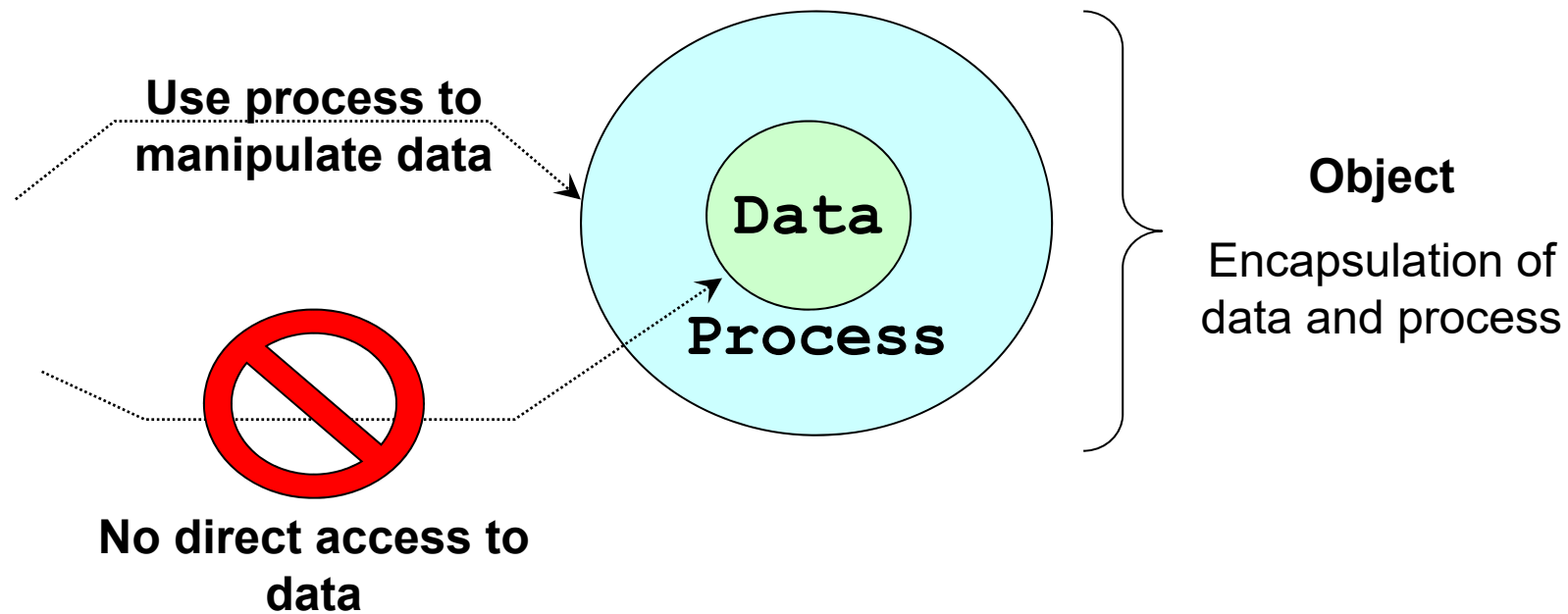
# C Programming Language: **Brief History**

- Designed by Dennis Ritchie in 1972
- First book that defines a standard C:
  - ❑ Brian Kernighan and Dennis Ritchie in 1978
  - ❑ Known as the **K & R C**
- Standardized in 1989 by ANSI
  - ❑ Known as the **ANSI C**
  - ❑ Coverage base on this version

# Programming Model

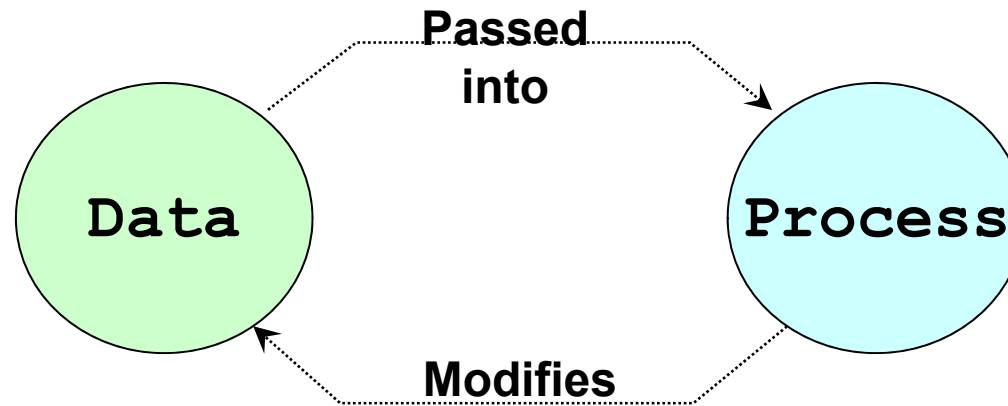
- C is designed with a ***Minimalist*** approach
  - ❑ Provide only the essential constructs
  - ❑ "Assume the programmers know what they are doing" – K & R
    - Minimal or No intervention from the compiler to ensure program correctness
  
- C uses ***Procedural*** programming model
  - ❑ Java and C++ uses ***Object-Oriented*** model
  - ❑ Less overhead in writing code
  - ❑ Harder to organize data and process

# Object Oriented Programming Model



```
class Counter
{
    private int _i; -----> Data
    ...
    public void increment() { _i++ }; -----> Process
    ...
}
```

# Procedural Programming Model



```
void increment( int* a)
{
    (*a)++;
}
```

} **Process**

```
int i; -----> Data
```

```
increment( &i ); -----> Data passed into a process
```

# Procedural Programming Model

- A program in this model consists of:
  - ❑ Data:
    - "Naked": directly accessible to everyone
    - Still possible to group data in meaningful packages through **structure** (more later)
  - ❑ Process:
    - Function:
      - ❑ Receive data and perform manipulation
      - ❑ "Transient": Do not remember information (mostly)
- It is the programmer responsibility to:
  - ❑ Introduce meaningful organization
  - ❑ Separate process and data into logical groups

# Hello World..... Again

- A simple side-by-side comparison between Java and C:

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

## Source File

HelloWorld.java

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

Header File

```
int main( )  
{  
    printf( "Hello World!" );  
    return 0;  
}
```

Function  
Call

## Source File

No restriction. Can  
use any file name  
with suffix ".c"

e.g. `hello.c` ,  
`program1.c` etc



# C Programming Construct

# Selection Statements

```
if (a > b) {  
    ...  
} else {  
    ...  
}
```

- if-else statement
- Valid conditions:
  - ❑ Comparison
  - ❑ Integer values (0 = false, others = true)

```
switch (a) {  
    case 1:  
        ...  
        break;  
    case 2:  
    case 3:  
        ...  
    default:  
}
```

- switch-case statement
- Variables in switch( ) must be integer type (or can be converted to integer)
- break : stop the fall through execution
- default : catch all unmatched cases

# Repetition Statements

```
while (a > b) {  
    ... //body  
}
```

```
do {  
    ... //body  
} while (a > b);
```

- Valid conditions:
  - ❑ Comparison
  - ❑ Integer values (0 = false, others = true)
- `while` : check condition before executing body
- `do-while`: execute body before condition checking

```
for (A; B; C) {  
    ... //body  
}
```

- A : initialization (e.g. `i = 0`)
- B : condition (e.g. `i < 10`)
- C : update (e.g. `i++`)
- Any of the above can be empty
- Execution order:
  - ❑ A, B, body, C, B, body, C ...

# Pitfall for Java Programmer

- Integer values is acceptable as condition in C
  - ❑ Can produce subtle logical bug!
- Example:

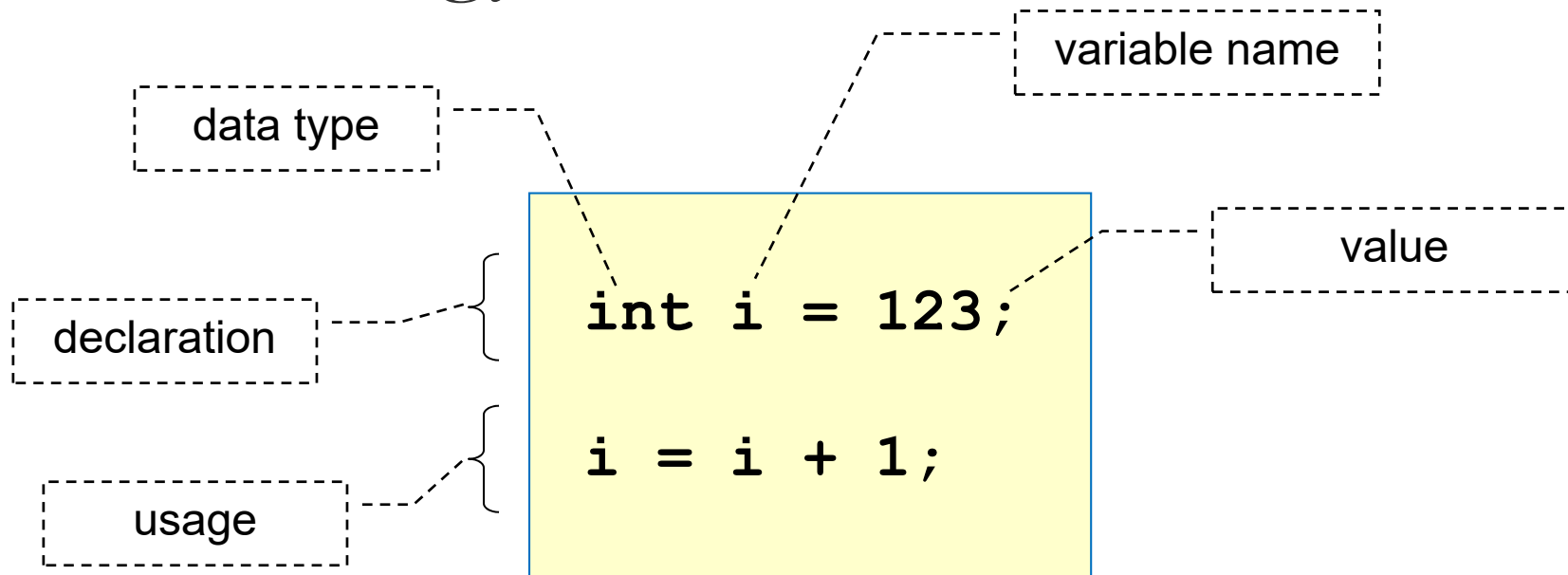
```
int i = 1;  
  
while (i = 1)  
    i++;
```

- Question:
  - ❑ How many iterations will you get from the while loop?

# C Data Types

---

# Terminology



## Memory Box Diagram

***name***   ***content***

<b>i</b>	123
----------	-----

# Simple Data Types

**int**

**unsigned int**

**char**

**unsigned char**

**float**

**double**

- Integer data
  - Similar range to Java version
  - Unsigned version can store only non-negative values
- Character data
  - Unsigned version can store only non-negative values
- Floating point data

# Simple Data Type : Comparison with Java

## ■ Key differences to Java:

### ❑ No object version

- e.g. no built-in **Integer** class equivalence in C

### ❑ **char** in C behaves like the **byte** datatype in Java:

- To store small integer numbers

### ❑ No initial value for variable

```
int i;    // i can be ANY value!!
```



# Array

- A collection of **homogeneous** data
  - ▣ Data of the same type

```
int iA[10];
```

- Declaration: An array of 10 integers

```
iA[0] = 123;
```

Store value into 1<sup>st</sup> element

```
iA[9] = 456;
```

Store value into last, 10<sup>th</sup> element

```
iA[1] = iA[0] + iA[9];
```

Store and read values

## Example Usage

## Array in C: Comparison with Java

- Size of array must be given in declaration
- Array in C behaves like primitive datatype:
  - ❑ e.g. No need for "new int[10]"
  - ❑ No built-in methods (it is not a class!)
    - No way to check size
  - ❑ No automatic check for array bounds!
- Assignment is NOT allowed:

```
int ia[10], ib[10];  
ia = ib;    //compilation error
```
- Behavior of Java array is actually closer to the pointer version in C (more later)

# Structure

- A collection of **heterogeneous** data

- ❑ Data of different type
- ❑ Should be a collection describing a common entity

```
struct Person {  
    char name[50];  
    int age;  
    char gender;  
};
```

```
struct Person s1;
```

```
typedef struct {  
    ... //same as above  
} PERSON;
```

```
PERSON s1;
```

- Declaration: A structure to store information about a person:

- ❑ **Name**: String of 50 characters
- ❑ **Age**: integer
- ❑ **Gender**: 'm' = male; 'f' = female

- **s1** is a structure variable

- Alternative declaration

- ❑ Improve readability
- ❑ Save some typing

# Structure

```
PERSON s1 = { "Potter", 13, 'm' };
```

```
PERSON s2;
```

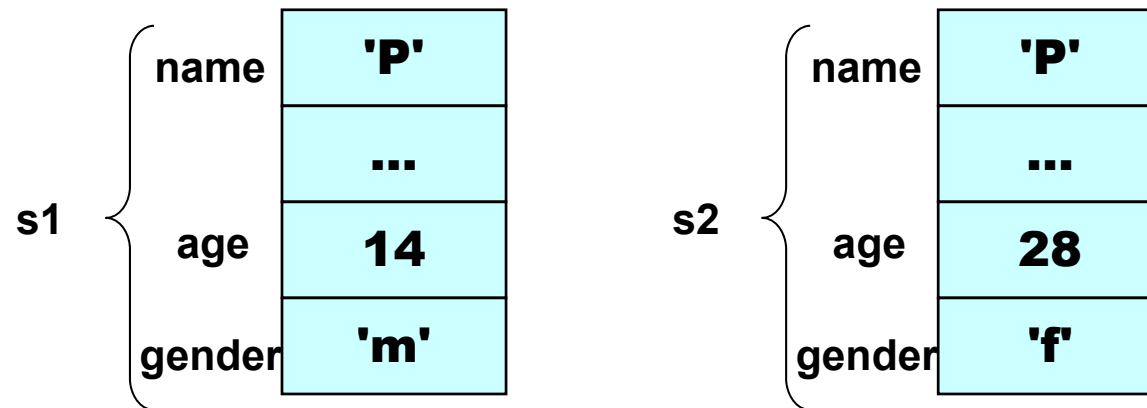
```
s2 = s1;
```

```
s1.age = 14;
```

```
s2.age = s1.age * 2;
```

```
s2.gender = 'f';
```

Example Usage



# Structure: Summary

- Assignment is possible:

```
struct Person s1, s2;
```

```
s1 = s2;    //s1 is a duplicate of s2
```

- Structure is the main way to organize information in C
- Similar to a class in Java with the following restrictions:
  - ❑ No methods declared
  - ❑ All attributes are public
  - ❑ Difference:
    - Statically allocated
    - e.g. No need for "new Person ( )" "

# Pointers in C

- Memory is a one dimensional array:
  - ❑ Each memory location has an unique index
  - ❑ Known as **memory address**
- In Java, the memory address is hidden from the programmer
  - ❑ Stored in a ***Reference***
  - ❑ Handled automatically by the system
- In C, programmer has full control / access to memory address
  - ❑ Make use of ***Pointer*** variable for address manipulation

# Pointer Variable

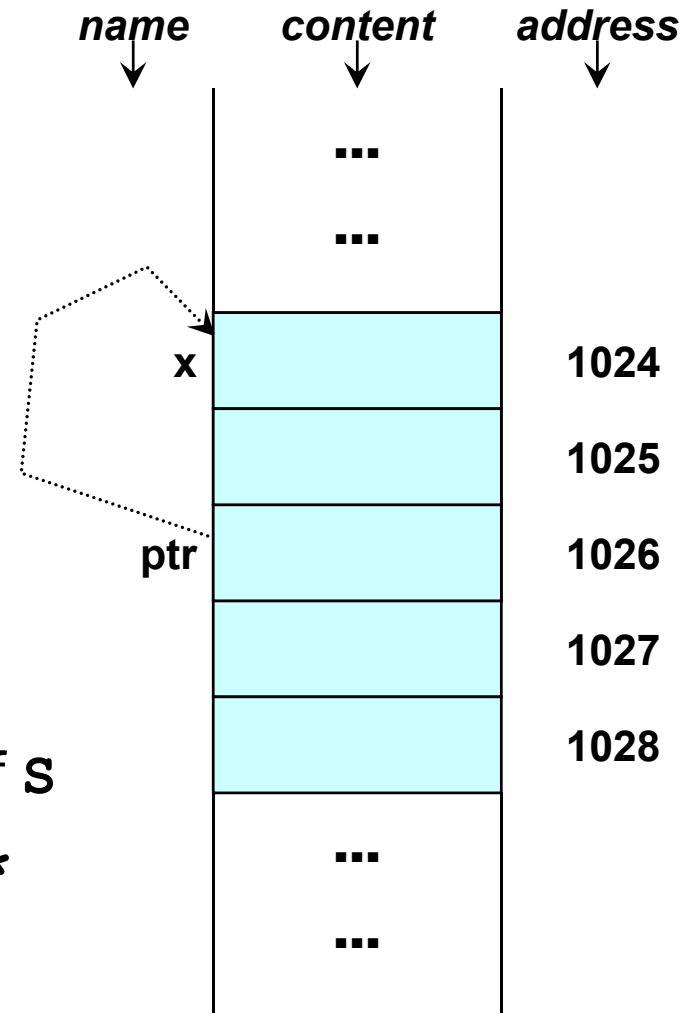
- A pointer contains the address of a memory location

```
int x;
```

```
int *ptr;    ptr is an int pointer
```

```
ptr = &x;    ptr points to x
```

```
*ptr = 123;  Dereference ptr
```



- & is the **Address-Of** operator
  - ❑ &(S) gives the memory address of S
- Note the different meanings of \*
  - ❑ Declaring a pointer
  - ❑ Dereference a pointer

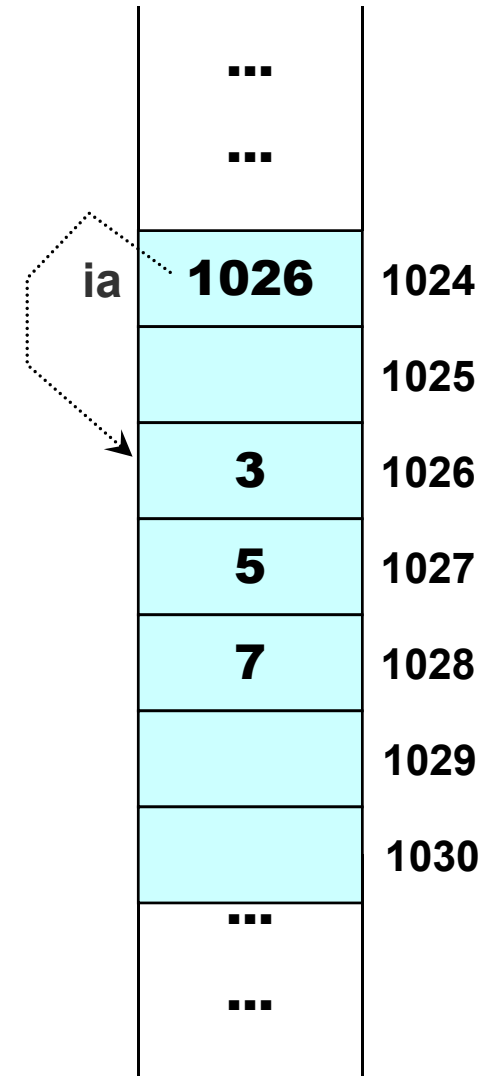
# Pointers and Arrays

- Array name is a **constant pointer**
  - Points to the zeroth element

```
int ia[3] = {3, 5, 7};
```

- Is the following valid?

```
int* ptr;  
  
ptr = ia;  
ia = ptr;  
ptr[2] = 9;  
  
ptr = &ia[1];  
ptr[1] = 11;
```





# Pointer Arithmetic

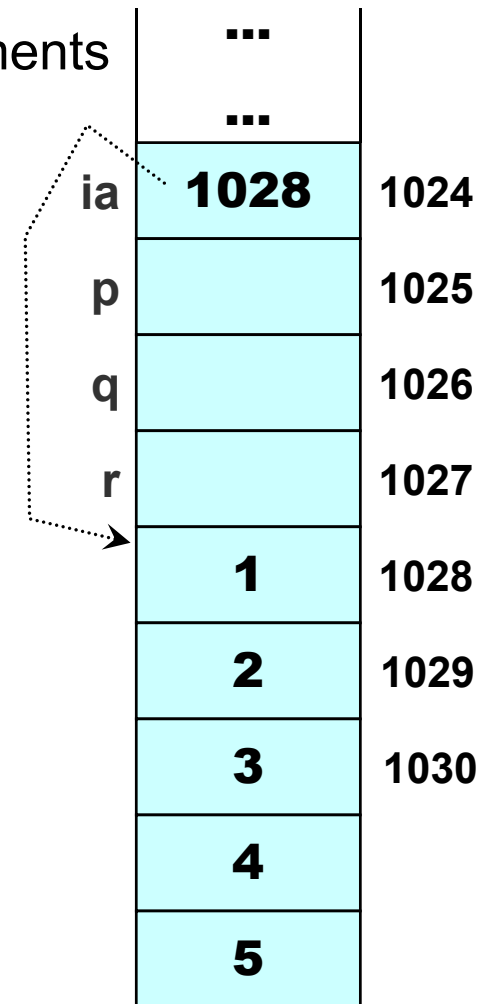
- Addition and subtraction of pointers are valid
  - $\text{Pointer} + X = \text{Move forward } X \text{ number of elements}$
  - $\text{Pointer} - X = \text{Move backward } X \text{ number of elements}$

```
int ia[5] = {1, 2, 3, 4, 5};  
int* p = ia;  
int *q, *r;
```

```
q = p + 3;    //what is q?  
r = q - 1;    //what is r?
```

```
//print *p  
//print *q  
//print *r
```

```
//print *p + 1  
//print *(p + 1)
```



# Pointer Arithmetic

- Two forms of element access for arrays:
  - ❑ Using [ ], i.e. indexing
  - ❑ Using pointer arithmetic

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

Using indexing

```
int ia[5] = {1, 2, 3, 4, 5};  
int *ptr;  
  
for (ptr = ia; ptr < ia + 5; ptr++)  
    //make use of *ptr
```

Using pointer arithmetic

# Pointer and Structure

- Pointer can points to a structure as well

```
int main()
{
    PERSON s =
        { "Potter", 13, 'm' };

    PERSON *p; //Person Pointer

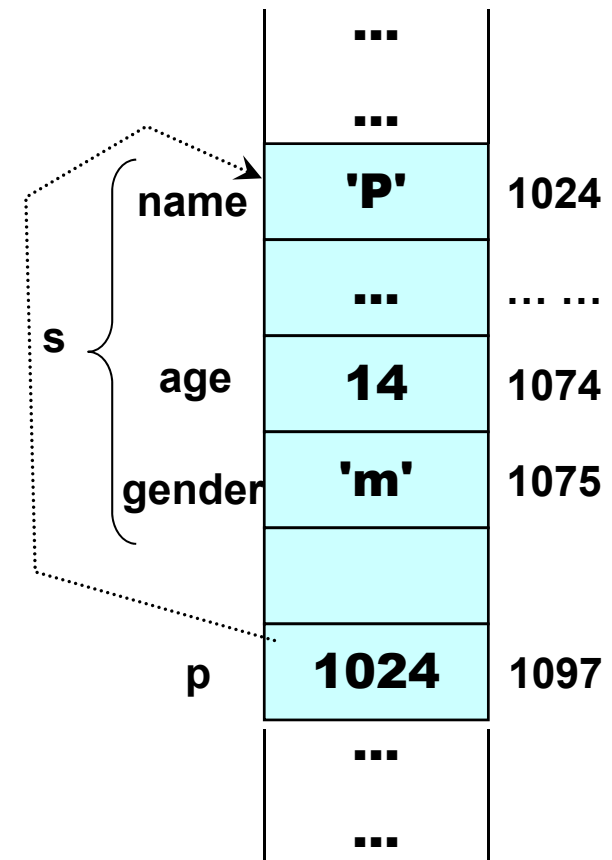
    p = &s;

    p->age = 14;

    (*p).age = 14;

}
```

Equivalent Statements



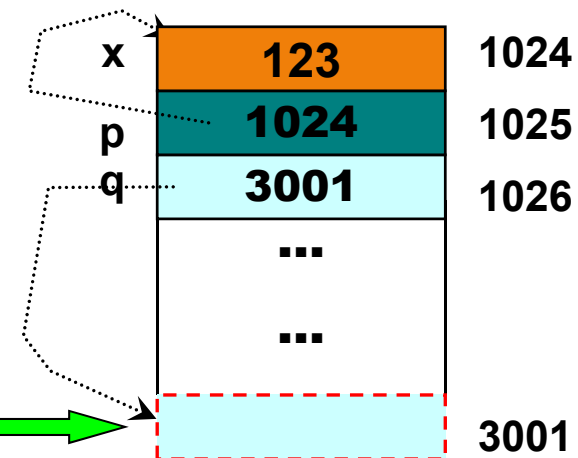
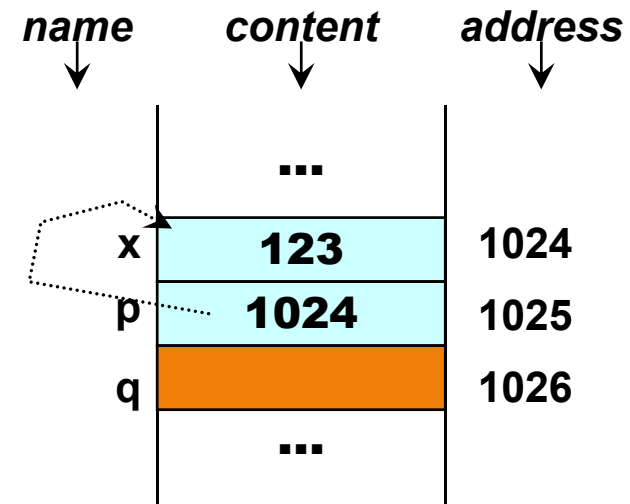
# Dynamic Memory Allocation : **malloc**

- Up to now, pointers are used to point to existing (declared) variable
- Actually, new memory box can be allocated at **runtime**
  - Using the `malloc()` library call
- Header File:  
`#include <stdlib.h>`
- Syntax:  
`malloc( size )`  
where size is the number of memory bytes required
- **Address** of the newly allocated memory locations is returned by the function
  - Use pointer variables to store the address

## malloc : Single Element – First try

```
int main()
{
    int x = 123;
    int *p, *q;

    p = &x;
    q = malloc( 4 );
}
```



# malloc : Improvements

- The `malloc ( )` function returns the address as a *datatype-less* pointer

```
void* malloc ( ... ) ;
```

- `void` → no datatype
- `void*` → pointer that has no datatype

- Use type-casting to *convert* to the correct data type:

- Example:

```
q = (int*) malloc ( 4 );
```

- Instead of memorizing the size of various datatype

- Use `sizeof ( )` library call

- Example:

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

- Improve program portability

# Memory Leak

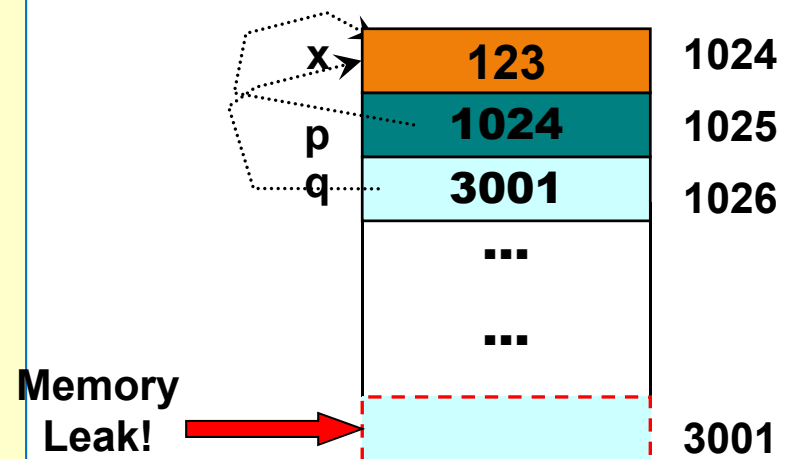
## ■ Important:

- ❑ q is the **only** variable storing the address of the new memory locations
- ❑ If q is changed, the new location is **lost** to your program, known as **memory leak**

```
int main()
{
    int x = 123;
    int *p, *q;
    p = &x;

    q = (int*)
        malloc (sizeof(int));

    q = p;
}
```



# malloc : Array of elements

- Whole array can be allocated dynamically
  - The size can be supplied at run time

```
int main()
{
    int *ia;

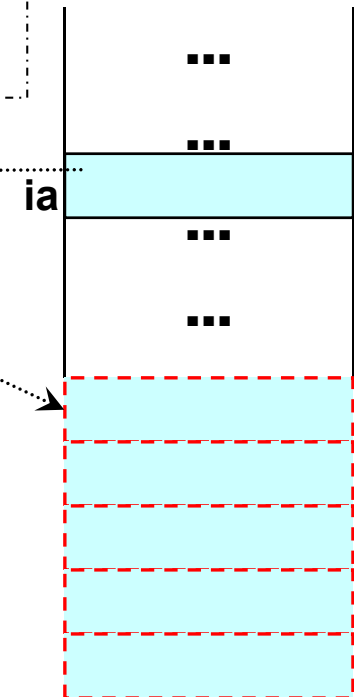
    ia = (int*)
        malloc( sizeof(int)*5 );

    ia[0] = ...
    ia[1] = ...

}
```

Number of  
elements

At this point



Assume size = 5



## malloc : Structure

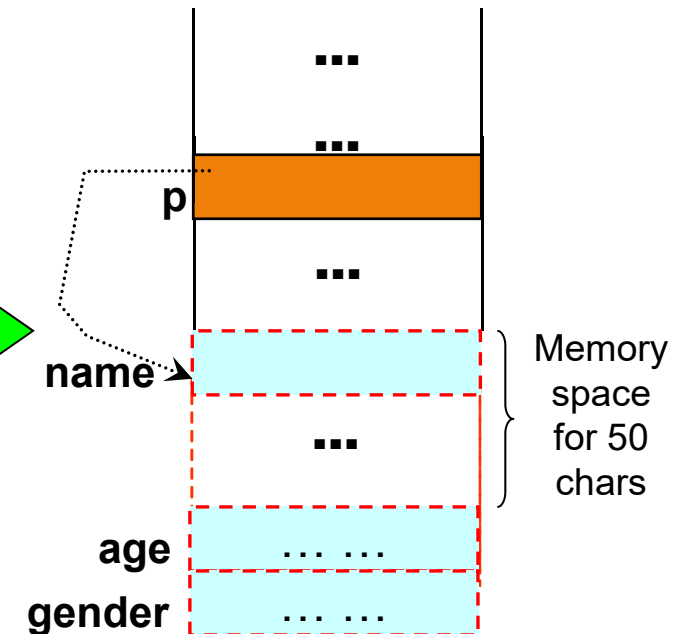
- Dynamic allocation for structure is also possible

```
int main()
{
    PERSON *p;

    p = (PERSON*)
        malloc( sizeof(PERSON) );

    p->age = 14;
}
```

At this point



## Releasing memory to system : **free**

- Dynamically allocated memory can be returned to the system (de-allocated)
  - ❑ Use **free()** library call
- Syntax:  
***free(pointer);***
- Memory location(s) pointed by *pointer* will be returned to the system
- Important:
  - ❑ Dereferencing pointer after **free** is invalid!

# free : An example

```
int main()
{
    PERSON *p;

    p = (PERSON*)
        malloc( sizeof(PERSON) );

    p->age = 14;

    free(p);

    p = NULL;

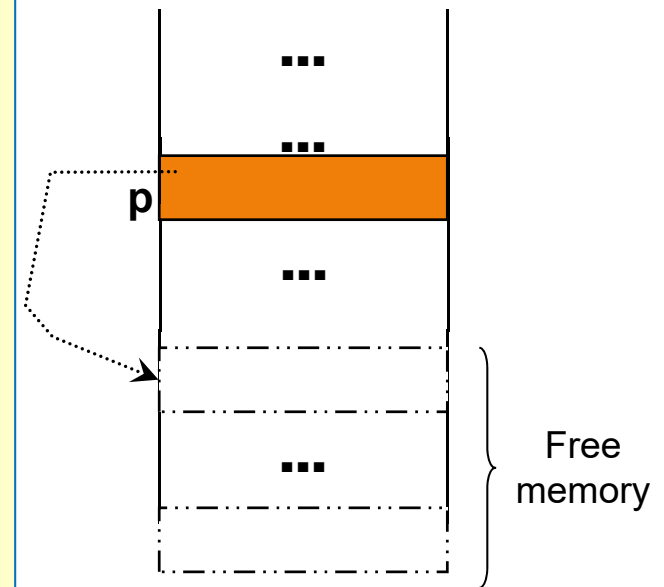
    p->age = 14;

}
```

At this point

Good Practice: **Always** set a pointer to NULL after free

Error!



# Pointer in C: Comparison with Java

- Pointer is similar to Reference in Java

- In C:

```
PERSON *p = (PERSON*) malloc( sizeof( PERSON ) );  
p->age = 30;
```

- In Java (given a similar Person class):

```
Person p = new Person();  
p.age = 30;           //assume age is public
```

- However, there is no need to clean up memory in Java:

- Automatic Garbage Collection

# C Function

# Function in C

- Similar to a ***static method*** in Java
  - i.e. not associated with an object
- Syntax:

*return\_type*    **function\_name** ( *parameters* )

- Example:

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

    return result;
}
```

# Function Prototype and Implementation

- Good practice to provide function prototypes

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

**Function prototype**

```
int main( )  
{  
    ...  
}
```

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

**Actual Implementation**

# Function : Parameter Passing

- There are **two** ways of passing a parameter into a function:
  1. Pass by value
  2. Pass by address ( Pass by pointer )
  
- Let us define a function **swap(a, b)** to swap the parameters
  - ❑ Using the two different parameter passing methods
  - ❑ Desired behavior: value of **a** and **b** swapped after function call



# Function : Pass by value

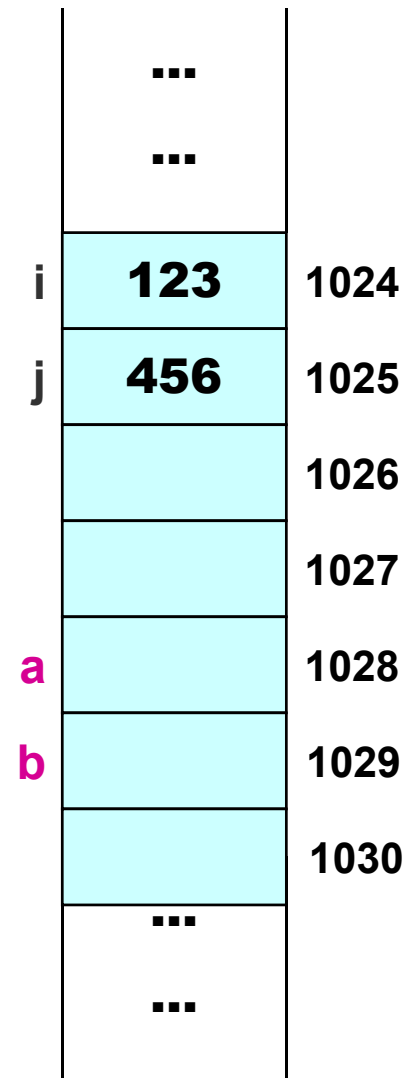
```
void swap_ByValue( int a, int b )
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByValue(i, j);

    //Print out i and j
}
```



# Function : Pass by address/pointer

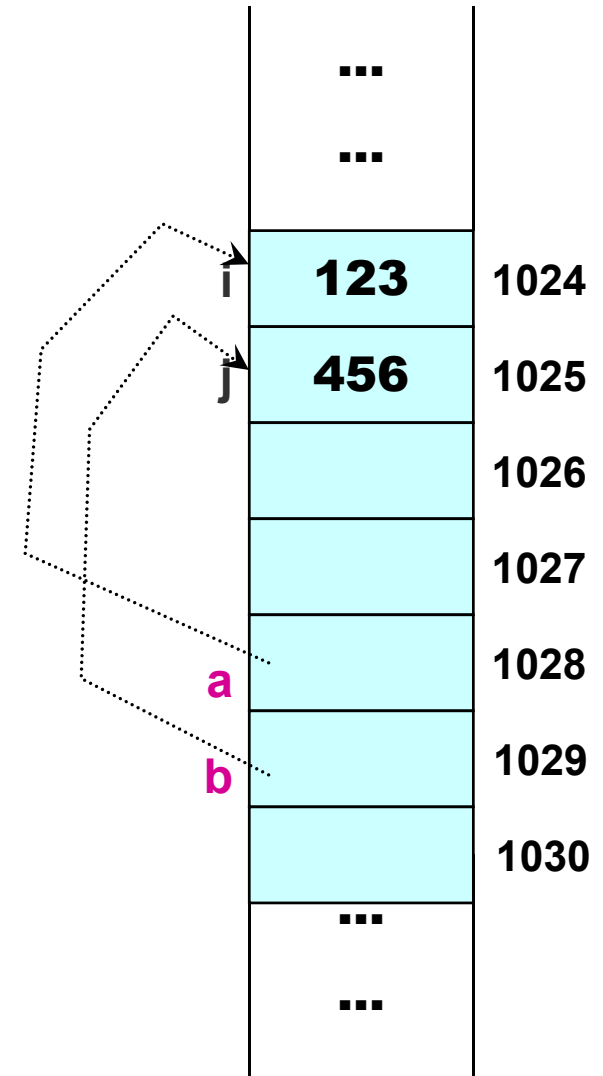
```
void swap_ByAdr( int* a, int* b )
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByAdr(&i, &j);

    //Print out i and j
}
```



# Function: Array as Parameter

- Array is passed by address

```
void f( int* a, int b[] )
{
    a[0] = b[0];
}

int main()
{
    int iA[3] = {1, 2, 3};
    int iB[3] = {4, 5, 6};

    f( iA, iB);

    printf("%i", iA[0]);
}
```

**Equivalent ways of specifying  
array parameter**

**Output is "4"**

# Function: Structure as Parameter

- Structure is passed by value

```
//Use the PERSON structure
```

```
void f( PERSON p )  
{  
    p.age = 55;  
}
```

**Attempts to modify the age**

```
int main()  
{  
    PERSON me = { "James", 12, 'm' };  
  
    f( me );  
    printf("%i", me.age);  
}
```

**Output is "12"**

# C Built-in Library

# Built-in Libraries

- Similar to Java built-in packages:
  - ❑ C has built-in libraries
    - Provides common functionalities
    - Not as extensive as Java's
- To make use of a C Library:
  - ❑ Includes the respective header file:  
`#include <XXXX.h>`
  - ❑ Some libraries require special compilation command
- Impractical to cover everything in lecture
  - ❑ Introduce the essential and basic libraries only
  - ❑ Lab question will provide extra information

# Standard Input/Output: `stdio.h`

- Header File:

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

- Major functionalities:

- ❑ Input – from keyboard, file
- ❑ Output – to screen, file

- Output to screen:

```
printf( format_string, [data] );
```

- ❑ *format\_string*:

- Specify format for data (if any)
- Also with the plain message to be printed
- Similar to `System.out.printf( )` in Java

# Standard Output: **printf( )**

## ■ Example:

```
int age = 55;  
double income = 5432.10;  
printf("Age is %i, Income is %f\n", age, income );
```

## ■ Common format specifier:

- ❑ %i : Print as integer
- ❑ %c : Print as character
- ❑ %f : Print as floating point
- ❑ %s : Print as string

## ■ Format modifier (Examples):

- ❑ %8i : print integer with default width of 8
- ❑ %6.3f : print floating point with default width of 6 with 3 digits precision



## Standard Input: `scanf ( )`

- Take input from standard input

`scanf( format_string, [addresses] );`

- *format\_string* is similar to the printf's

- Important:

- Supply the **address** of variable to store the input
- Use the "&" operator if necessary

- Example:

```
int age;
```

```
scanf("%i", &age); //note the "&"
```

## Standard Input: **fgets ( )**

- Reading string is slightly more troublesome:

```
char name[50];
```

```
scanf("%s", name);    //note the absence of "&"
```

- ❑ Read only a **single word**

- ❑ Reading ends as soon as a space is encountered

- Use **fgets ( )** to read a single sentence:

```
fgets( char* string, int size, FILE *stream);
```

- **Example:**

```
char name[50];
```

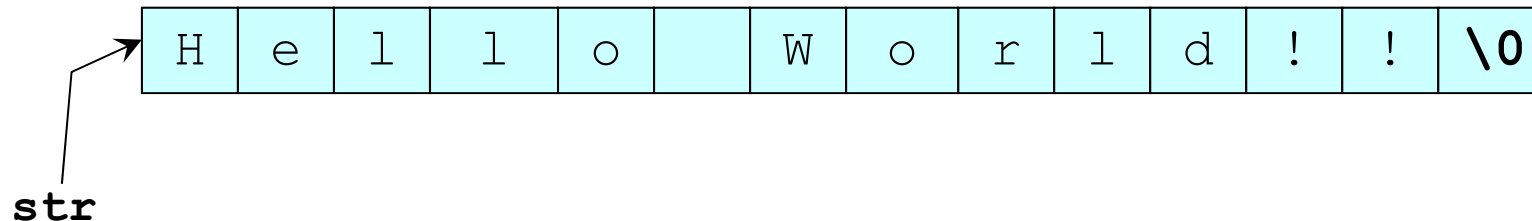
```
fgets(name, 50, stdin);
```

- ❑ **stdin**: the standard input device

# Character String in C

- String in C is represented as a *special* character array
- Example:

```
char str[14] = "Hello World!!";
```



- Note the ' \0 ' at the end:
  - Known as the string terminator
  - Very important:
    - This is **only** distinction between a string and a simple character array in C

# String Library : **string.h**

- Header File:

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

- Major functionalities:

- ❑ Strings Comparison
- ❑ Strings operation:
  - Duplicating a string
  - Strings concatenation
  - etc

# String operations: Summary

- `strcpy(char* dest, const char* source);`
  - ❑ Copy the source to `dest` string
- `strcat(char* dest, const char* source);`
  - ❑ Add the source to the end of `dest` string
- Caution:
  - ❑ Make sure the `dest` string has enough space for the above functions
- `int strcmp(char* s1, char* s2)`

Returns:

  - ❑ 0: if `s1 == s2`
  - ❑ >0: if `s1 > s2`
  - ❑ <0: if `s1 < s2`

# C Compilation

# C: How to compile?

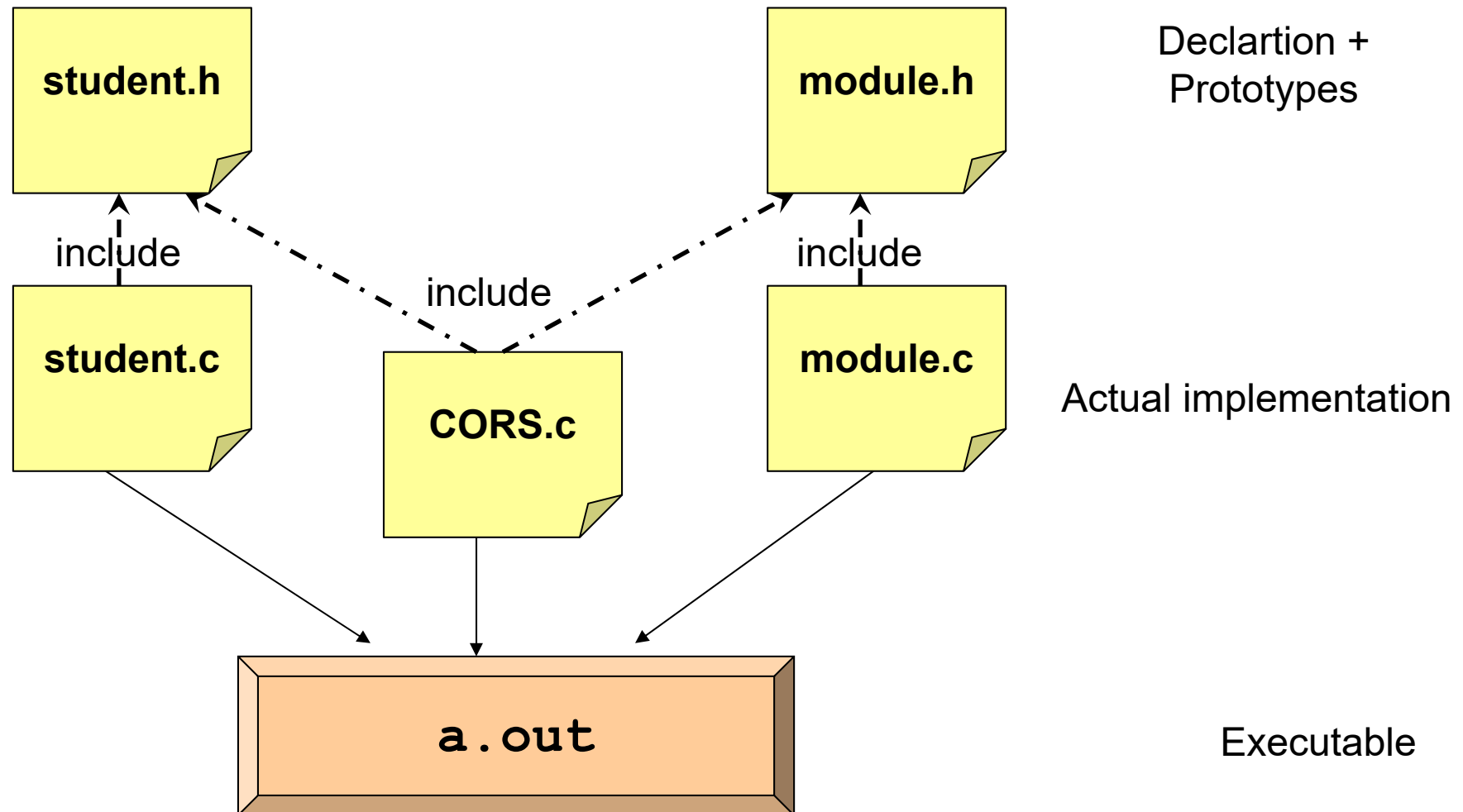
- C source code must have the extension “.c”
  - Example: `helloWorld.c`
- Steps for program compilation:
  1. Edit program using your favorite editor (`vim`, `pico` etc)
  2. Compile using the command:  
`gcc -Wall fileName.cpp` OR  
`gcc -Wall -o executable_name fileName.cpp`
  3. [Compilation Error]: Go to step 1
  4. Execute the program:  
`a.out` OR `executable_name`

# Modular Design in C

- In Java, source code is organized into:
  - packages:
    - Contain a number of classes
- In C, source code ***can*** be organized into:
  - Header files (with extension ".h")
    - Contains declaration and function prototypes
  - Implementation files (with extension ".c")
    - Contains implementation of the function(s) declared
  - No enforcement
    - Programmer's responsibility to make use of it consistently



# Modular Design: Example



## C: How to compile multiple files

- Using the example from previous page:
  - ❑ `gcc -Wall CORS.c student.c module.c`
- The above ***recompiles*** every source code
  - ❑ Inefficient:
    - what if we have change only the CORS.c?
- Can compile individual source code:  
`gcc -Wall -c xxxx.c`
  - ❑ Produce `xxxx.o` if no error
  - ❑ Can selectively recompile
- To produce the executable
  - ❑ `gcc -Wall CORS.o student.o module.o`

# Preprocessor: Macro

- A C source code is first preprocessed during compilation
- The preprocessor performs:
  - ❑ Textual substitution
  - ❑ Insert/Remove code
  - ❑ Locate and insert header file
- A command to preprocessor starts with "#"
  - ❑ Known as preprocessor ***directive***
  - ❑ Example:  
`#include <stdio.h>`
  - ❑ NOT part of C syntax technically

# Preprocessor Directive: **#include**

- **#include xxxx**

- locate and paste the file **xxxx** into the actual source code

- **#include <xxxx.h>**

- **xxxx.h** is a standard C library
  - Store in a predefined location

- **#include "xxxx.h"**

- **xxxx.h** is a file in local directory
- User defined header file

# Preprocessor Directive: **#define**

- **#define X Y**

- locate occurrences of X in the source code and replace it by Y

- Example:

```
#define MAX 1024  
int iA[ MAX ] ;
```

- After preprocessing:

```
int iA[1024] ;
```

- A simple way to provide limited maintainability:
  - Just change **MAX** to a new value and recompile

# Preprocessor Directive: **#ifdef**

- **#ifdef X**  
... .. //Region A  
**#endif**
- If **X** is defined earlier:
  - ❑ code in the Region A will be included in the compilation
- If **X** is not defined:
  - ❑ code in Region A will be removed
- Example:  
**#ifdef DEBUG**  
**printf("Code for debugging\n");**  
...  
**#endif**
- Can turn on debugging by:
  - ❑ Insert "**#define DEBUG**" before the fragment above