Intro to C

Computer Systems Organization

Intro to C Programming

Introduction
A Simple C Program: Printing a Line of Text
Another Simple C Program: Adding Two Integers
Memory Concepts
Arithmetic in C
Decision Making: Equality and Relational Operators
Pointers
Reading from Command Line
Arrays
Strings
Structure
Dynamic Memory Allocation

Introduction

C programming language

- Widely used language
- Strengths: high speed and system programming

Comparing C and C++

Similarities

- Both have **similar syntax**
  - for-loops, while-loops, if-else, function declaration, arithmetic …
- C++ is a superset of C
  - C++ extends C
  - Most C codes are also C++ codes

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)

```
try {
  // Block of code to try

  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading

What is function overloading?

```
int foo(int input)
{
    // Code
}

int foo(string input)
{
    // Code
}
```

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading
- Namespaces

```
// first name space namespace
namespace first_space
{
   int foo(string input);
   // …
}

// second name space namespace
namespace second_space
{
   int foo(string input);
   // …
}
```

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading
- Namespaces
- New and delete operators (use malloc() and free() instead)

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading
- Namespaces
- New and delete operators (use malloc() and free() instead)
- Template classes and function

A function with a generic type T

```
template<class T> T GetMax (T a, T b)
{
  T result;
  result = (a > b)? a : b;
  return result;
}
```

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading
- Namespaces
- New and delete operators (use malloc() and free() instead)
- Template classes and function
- Standard template library (STL)

Comparing C and C++

Differences

C does not support the following C++ language features

- Object oriented programming (classes …)
- C++ exception handling (try/catch)
- Function and operator overloading
- Namespaces
- New and delete operators (use malloc() and free() instead)
- Template classes and function
- Standard template library (STL)
- Reference types (e.g., int& x)

```
void swap(int& x, int& y)
{
    int z = x;
    x = y;
    y = z;
}
…
swap(x, y);
```

C++ only

```
void swap(int* x, int* y)
{
    int z = *x;
    *x = *y;
    *y = z;
}
…
swap(&x, &y);
```

C and C++

A Simple C Program: Printing a Line of Text

```
1  /* This is a comment */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```


```
Welcome to C!
```

Comments

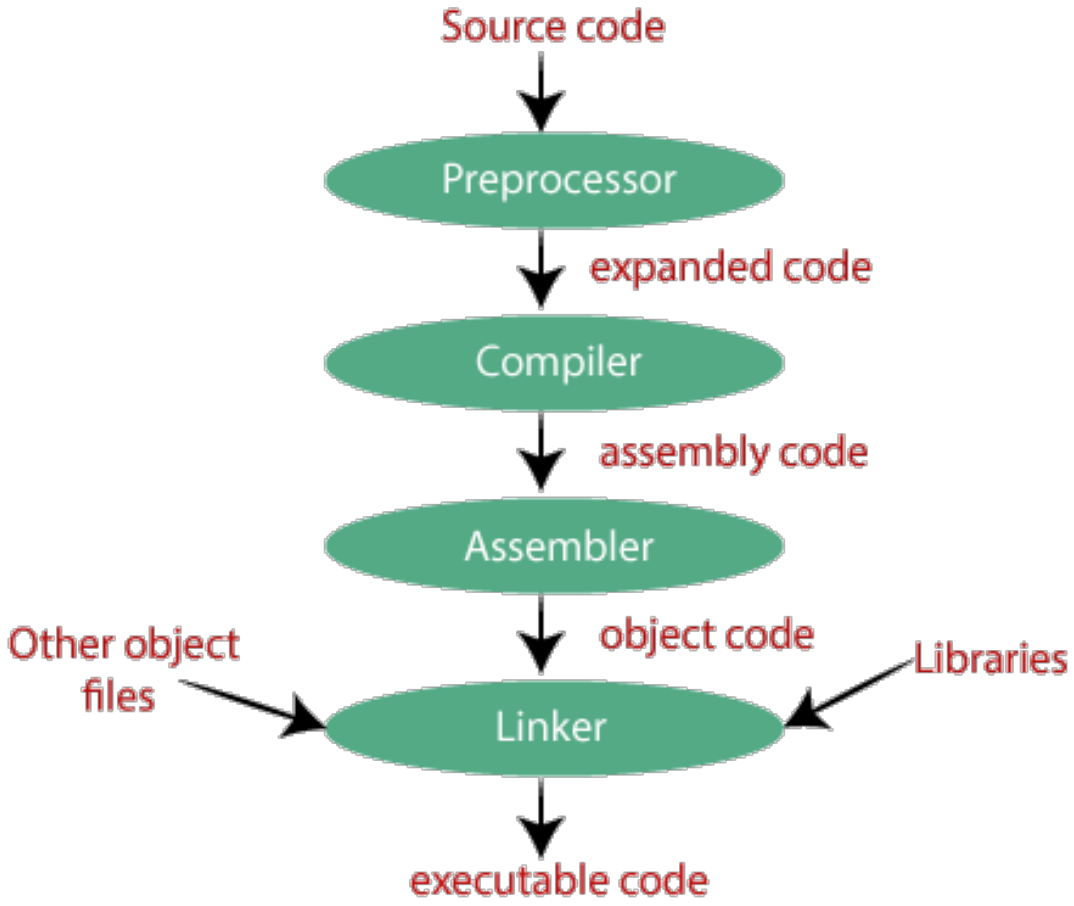
- Text surrounded by **/*** and ***/** is ignored by computer
- Used to describe program

#include <stdio.h>

- Preprocessor directive
  - Tells computer to load contents of a certain file
- **<stdio.h>** allows standard input/output operations

Compiler Steps

Preprocessor
- Macro substitution
- Stripping of comments
- Expansion of included files



A Simple C Program: Printing a Line of Text

```
1  /* This is a comment */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```


```
Welcome to C!
```

int main()

- C programs contain one or more functions, exactly one of which must be **main**
- **int** means that **main** "returns" an integer value
- Braces ({ and }) indicate a block
  - The bodies of all functions must be contained in braces

General Form of Declaring a Function

```
ReturnType Name (Type Name, Type Name, Type Name)
{


}
```

A Simple C Program: Printing a Line of Text

```
1  /* This is a comment */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C \n" );
8
9     return 0;
10 }
```

```
Welcome to C!
```

- Print a string
- "\n" means start a new line

Escape sequences is	
\a	alert (bell) character
\n	newline
\t	horizontal tab
\\	backslash
\'	single quote
\"	double quote

A Simple C Program: Printing a Line of Text

```
1  /* This is a comment */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```

```
Welcome to C!
```

- Instructs computer to perform an action
  - Specifically, prints the string of characters within quotes (“ ”)
- Entire line called a statement
  - All statements must end with a semicolon (;)

A Simple C Program: Printing a Line of Text

```
1  /* This is a comment */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```

```
Welcome to C!
```

return 0;

- A way to exit a function
- **return 0**, in this case, means that the program terminated normally

Another Simple C Program: Adding Two Integers

```
1  /* Addition program */
2  #include <stdio.h>
3
4  int main()
5  {
6     int integer1, integer2, sum;         /* declaration */
7
8     printf( "Enter first integer\n" );   /* prompt */
9     scanf( "%d", &integer1 );            /* read an integer */
10    printf( "Enter second integer\n" );  /* prompt */
11    scanf( "%d", &integer2 );            /* read and integer */
12    sum = integer1 + integer2;           /* assignment of sum
13    printf( "Sum is %d\n", sum );        /* print sum */
14
15    return 0;  /* indicate that program ended successfully */
16 }
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Another Simple C Program: Adding Two Integers

```
1  /* Addition program */
2  #include <stdio.h>
3
4  int main()
5  {
6     int integer1, integer2, sum;      /* declaration */
7
8     printf( "Enter first integer\n" );  /* prompt */
9     scanf( "%d", &integer1 );           /* read an integer */
10    printf( "Enter second integer\n" ); /* prompt */
11    scanf( "%d", &integer2 );           /* read and integer */
12    sum = integer1 + integer2;          /* assignment of sum
13    printf( "Sum is %d\n", sum );       /* print sum */
14
15    return 0;  /* indicate that program ended successfully */
16 }
```

Another Simple C Program: Adding Two Integers

```
6     int integer1, integer2, sum;          /* declaration */
```

- Declaration of variables
  - Variables: locations in memory where a value can be stored
- **int** means the variables can hold integers (**-1**, **3**, **0**, **47**)
- Variable names (identifiers)
  - **integer1**, **integer2**, **sum**
  - Identifiers: consist of letters, digits (cannot begin with a digit) and underscores(**_**)
    - Case sensitive
- Declarations appear before executable statements
  - If an executable statement references and undeclared variable it will produce a syntax (compiler) error

Another Simple C Program: Adding Two Integers

```
1  /* Addition program */
2  #include <stdio.h>
3
4  int main()
5  {
6     int integer1, integer2, sum;         /* declaration */
7
8     printf( "Enter first integer\n" );   /* prompt */
9     scanf( "%d", &integer1 );            /* read an integer */
10    printf( "Enter second integer\n" );  /* prompt */
11    scanf( "%d", &integer2 );            /* read and integer */
12    sum = integer1 + integer2;           /* assignment of sum
13    printf( "Sum is %d\n", sum );        /* print sum */
14
15    return 0;  /* indicate that program ended successfully */
16 }
```

Another Simple C Program: Adding Two Integers

```
9     scanf( "%d", &integer1 );          /* read an integer */
```

- Reads a value from the user
  - **scanf** uses standard input (usually keyboard)

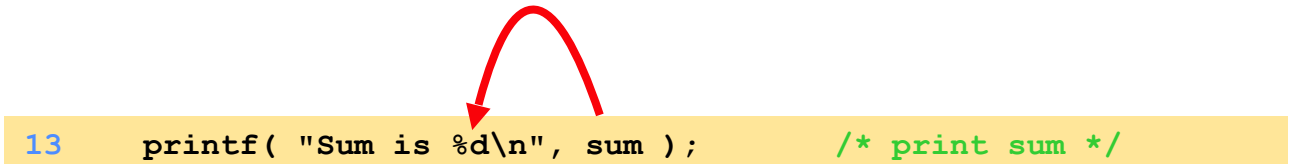
- **scanf** has two arguments
  - **"%d"**
    - a string that specifies the format of the input (e.g., its type)
    - %d indicates that data should be a decimal signed integer
    - %u indicates that data should be an unsigned integer
    - %f indicates that data should be a floating-point
  - **&integer1**
    - location in memory to store the read value
    - **&** indicates passing integer1 by reference to the scanf() function
    - & is confusing in the beginning – for now, just remember to include it with the variable name in **scanf** statements

- When executing the program the user responds to the **scanf** statement by typing in a number, then pressing the *enter* (return) key

Another Simple C Program: Adding Two Integers

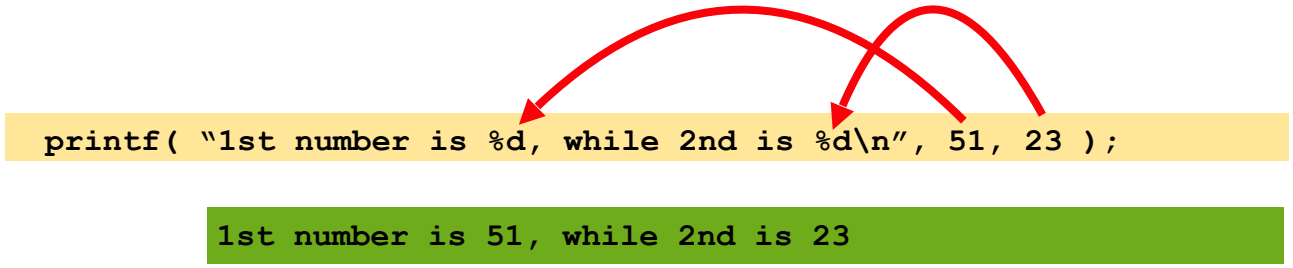
```
1  /* Addition program */
2  #include <stdio.h>
3
4  int main()
5  {
6     int integer1, integer2, sum;         /* declaration */
7
8     printf( "Enter first integer\n" );   /* prompt */
9     scanf( "%d", &integer1 );            /* read an integer */
10    printf( "Enter second integer\n" );  /* prompt */
11    scanf( "%d", &integer2 );            /* read and integer */
12    sum = integer1 + integer2;           /* assignment of sum
13    printf( "Sum is %d\n", sum );        /* print sum */
14
15    return 0;  /* indicate that program ended successfully */
16 }
```

Another Simple C Program: Adding Two Integers



```
13     printf( "Sum is %d\n", sum );        /* print sum */
```

- Similar to **scanf**
  - **%d** means decimal integer will be printed
  - **sum** specifies what integer will be printed



```
printf( "1st number is %d, while 2nd is %d\n", 51, 23 );
```

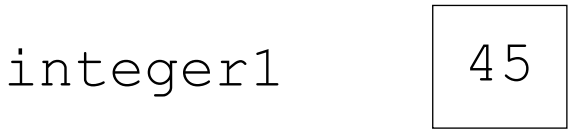
```
1st number is 51, while 2nd is 23
```

Memory Concepts

Variables

- Variable names correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- Whenever a new value is placed into a variable (through **scanf**, for example), it replaces (and destroys) the previous value
- Reading variables from memory does not change them

A visual representation



Reserved Words

auto	**double**	**int**	**struct**
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Decision Making: Equality and Relational Operators

```
if (condition)
{
    statement1;
}
```

if conditional

- If a condition is **true**, then the body of the **if** statement executed
  - **0** is **false**, non-zero is **true**
- Control always resumes after the **if** structure

Decision Making: Equality and Relational Operators

```
if (condition)
{
    statement1;
}
else
{
    statement2;
}
```

The **else** body is executed if the condition is false

Decision Making: Equality and Relational Operators

C equality or relational operator	Example of C condition	Meaning of C condition
==	`x == y`	**x** is equal to **y**
!=	`x != y`	**x** is not equal to **y**
>	`x > y`	**x** is greater than **y**
<	`x < y`	**x** is less than **y**
>=	`x >= y`	**x** is greater than or equal to **y**
<=	`x <= y`	**x** is less than or equal to **y**

Practice

- Write a C program that reads two integers and checks if they are equal

```
Enter two integers, and I will tell you
whether they are equal: 3 7
3 is not equal to 7
```

```
Enter two integers, and I will tell you
whether they are equal: 4 4
4 is equal to 4
```

- Online compiler: www.onlinegdb.com/online_c_compiler

Run locally on the machine the following command:

- **gcc** -o hello hello.c OR gcc –o hello.out hello.c
- The -o option specifies the name to be assigned to the compiled/executable program.

2.Run the following on the terminal:

- ./hello OR ./hello.out

GCC(GNU Compiler Collection): A compiler.

Practice

```
1  /*
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  int main()
7  {
8     int num1, num2;
9
10    printf( "Enter two integers, and I will tell you\n" );
11    printf( "whether they are equal: " );
12    scanf( "%d %d", &num1, &num2  );   /* read two integers */
13
14    if ( num1 == num2 )
15       printf( "%d is equal to %d\n", num1, num2 );
16
17    if ( num1 != num2 )
18       printf( "%d is not equal to %d\n", num1, num2 );
19
20    return 0;   /* indicate program ended successfully */
35 }
```

Arithmetic

Arithmetic operations

- + for addition
- - for substruction
- * for multiplication
- / for division
- % for remainder
  - **7 % 5** evaluates to **2**

Operator precedence

- Some arithmetic operators act before others (i.e., multiplication before addition)
  - Use parenthesis when needed
- Example: Find the average of three variables **a**, **b** and **c**
  - Do not use: **a + b + c / 3**
  - Use: **(a + b + c) / 3**

Arithmetic

Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication,Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Arithmetic

Full table of operator precedence (Table 2-1 in Chap.2 of the C textbook)

	Operators	Associativity
	`() [] -> .`	left to right
Unary	`! ~ ++ -- + - * & (type) sizeof`	right to left
	`* / %`	left to right
	`+ -`	left to right
	`<< >>`	left to right
	`< <= > >=`	left to right
	`== !=`	left to right
Bitwise operations	`&`	left to right
	`^`	left to right
	`|`	left to right
	`&&`	left to right
	`||`	left to right
	`?:`	right to left
	`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
	`,`	left to right

Unary +, –, and * have higher precedence than the binary forms.

How are the following evaluated?

x & MASK == 0 <equivalent to> x & (MASK == 0)

1 + x * 2 <equivalent to> 1 + (x * 2)

Common Bugs

Order of Evaluation of Arguments

What is the expected result?

```
n = 1;

printf("%d %d\n", ++n, power(2, n));
```

- The order in which function arguments are evaluated is not specified in C
- The compiler can choose in which order to evaluate them
- Better avoid this to avoid bugs (and compiler-dependent behavior)
- Another example

```
x = f() + g();
```

- The compiler can evaluate either f() first or g() first

Common Bugs

What is the problem?

```
n = 1;
if (n = 1)
  printf("Cool!");
```

Problem: use of "=" instead of "=="

"=" should be used for assignments while "==" is used in conditionals

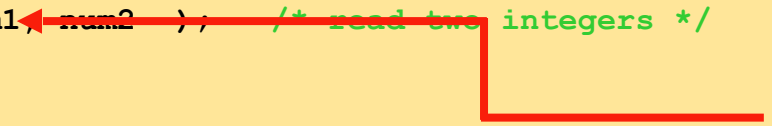
The previous code should be

```
n = 1;
if (n == 1)
  printf("Cool!");
```

Common Bugs

What is the problem?

```
1  /*
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  int main()
7  {
8     int num1, num2;
9
10    printf( "Enter two integers\n" );
11
12    scanf( "%d %d", num1, num2 );    /* read two integers */
13
14    if ( num1 == num2 )
15       printf( "%d is equal to %d\n", num1, num2 );
16
17    if ( num1 != num2 )
18       printf( "%d is not equal to %d\n", num1, num2 );
19
20    return 0;   /* indicate program ended successfully */
35 }
```



No "&"

Common Bugs

What is the problem?

```
1  /*
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  int main()
7  {
8     int num1, num2;
9
10    printf( "Enter two integers\n" );
11
12    scanf( "%d %d", &num1, &num2  );    /* read two integers */
13
14    if ( num1 == num2 )
15       printf( "%d is equal to %d\n", num1, num2 );
16
17    if ( num1 != num2 )
18       printf( "%d is not equal to %d\n", num1, num2 );
19
20    return 0;   /* indicate program ended successfully */
35 }
```

Practice (Function Call)

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

int main(){
        printf("%d\n",sum(2,3));
        return 0;
}

int sum(int a ,int b){
        return a+b;
}
```

Practice

Write a C program that contains two functions:
The first function converts the input length from meter to kilometer
The second function converts the input temperature from Fahrenheit to Celsius:

$$T(c) = (T(f) - 32) * 5/9$$

Inside the main call these two methods

Hardcode the inputs for now.

Practice

```c
#include <stdio.h>
double lengthConverter(double);
double tempratureConverter(double);
int main(){

        double input =  4500;

        double temp = 35;

        printf("%f m = %f km \n",input,lengthConverter(input));
        printf("%f Fahrenheit is %f degrees Celsius\n",temp,tempratureConverter(temp));
        return 0;
}
double lengthConverter(double a){
        //convert meter to kilometer
        return a/1000;
}
double tempratureConverter(double temp){
        //convert Fahrenheit to Celsius
        return (temp-32)*5/9;
}
```

Practice

Write a C program that scans a number “n” from the user and prints all even numbers from 1 to “n”

Practice

```c
#include <stdio.h>

int main(){
        int n;
        scanf("%d",&n);
        int i =1;
        for (i = 1; i <=n; i++)
                if(i%2==0)
                        printf("%d ",i);
        printf("\n");
        return 0;
}
```

Practice (Command Line)

Write a C program that finds and prints the maximum between three numbers. **Scan** one number from the user and pass two numbers as **command line arguments**. You should write and use a function **find_max()** to find the maximum.

Practice (Reading from Command Line)

```c
#include <stdio.h>
#include <stdlib.h>
int find_max(int,int,int);
int main (int argc, char **argv){
        if(argc !=3){
                printf("You should enter only 2 numbers as command line\n");
        }else{
                int i;
                scanf("%d",&i);
                int j = atoi(argv[1]);
                int k = atoi(argv[2]);
                printf("The max between %d, %d, and %d is %d\n",i,j,k,find_max(i,j,k));
        }
        return 0;
}
int find_max(int a, int b, int c){
        if(a > b){
                if( a > c)
                        return a;
                else
                        return c;
        }else if (b > c)
                return b;
        else
                return c;}
```

Pointers

What are pointers?

- As the name implies, a pointer “points” to a location in memory. They are variables that store memory addresses.

For what purposes?

- Retrieving value stored in memory location
- Passing large objects without copying them
- Accessing dynamically allocated memory
- Referring to functions

Notations used with pointers

- **&** → address of operator
- ***** → dereference operator

Declaration of a pointer

- **<variable_type> *<name>;**

Practice (Pointers)

```c
#include <stdio.h>

int main () {

   int  var = 20;   /* variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

Practice (Pointers)

```
int a = 1;
int b = 2;
int *c = 3;
int *d = b;
int *e = &b;
int *f = *b;
```

WHY?
Pointers stores addresses not values

3 is a value

b is equal to 2 which is a value

b is not a pointer

Practice (Pointers)

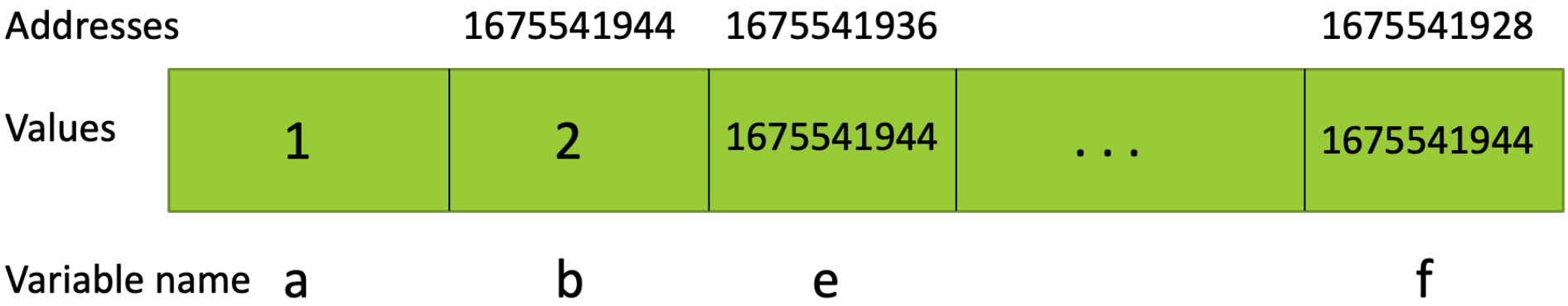
What is the output

```
int a = 1;

int b = 2;

int *e = &b;

int *f;
f=&b;

printf("a = %d\nb = %d\ne = %d\n*e = %d\n&e = %d\n&b = %d\nf = %d\n*f = %d\n&f = %d\n"
                                        ,a,b,e,*e,&e,&b,f,*f,&f);
```

Practice (Pointers)

a = 1
b = 2
e = 1675541944
*e = 2
&e = 1675541936

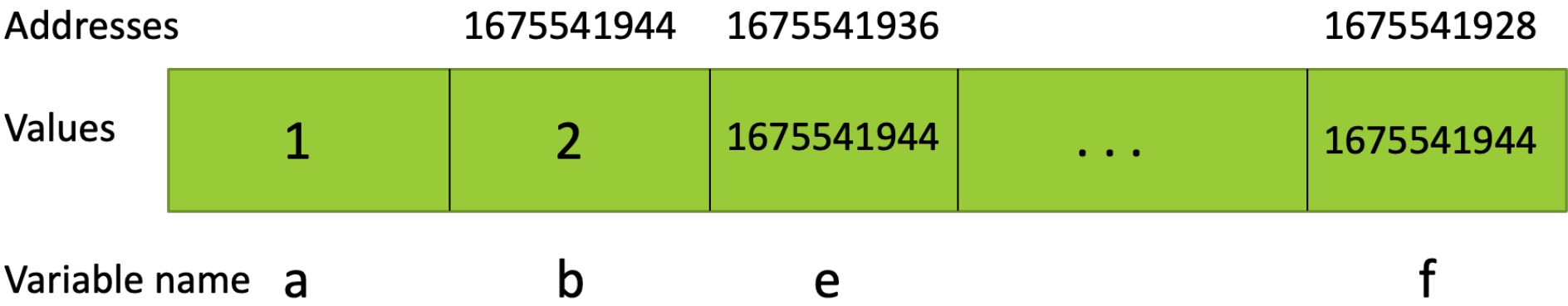
&b = 1675541944
f = 1675541944
*f = 2
&f = 1675541928



Practice (Pointers)

a = 1
b = 2
e = 1675541944
*e = 2
&e = 1675541936

&b = 1675541944
f = 1675541944
*f = 2
&f = 1675541928

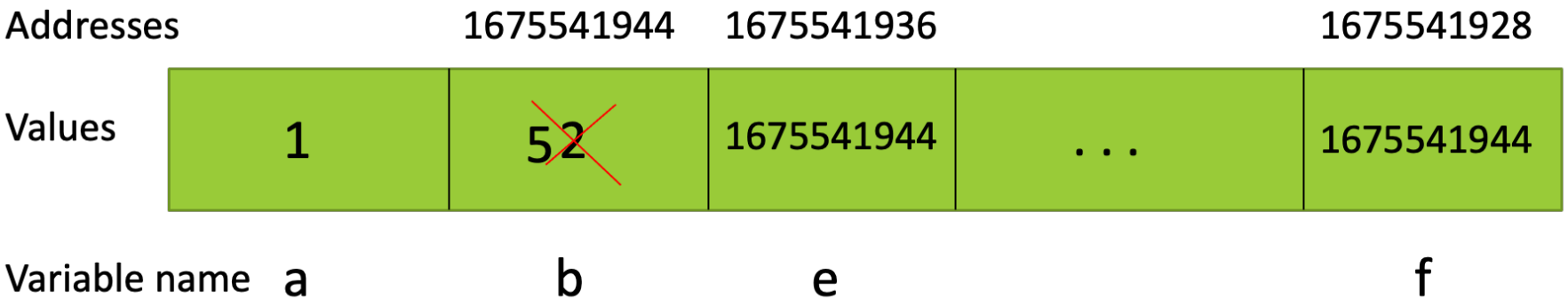


*f = 5
What will happen?!

Practice (Pointers)

```
root@kali:~# ./c
b = 5
e = 0x7ffe86738f18
*e = 5
&e = 0x7ffe86738f10
f = 0x7ffe86738f18
 *f = 5
&f = 0x7ffe86738f08
```

Notice that e, f, &f, &e contains memory addresses to view this address we used %p instead of %d in printf



C Parameters Passing

By value:

- A duplicate copy of the argument is created and passed to the called function.
- Any update of the variable inside the function wont affect the original variable.

By reference

- The address of the variable is passed to the called function.
- Any update of the variable inside the function will affect the original value since the value is directly changed in its exact memory location.

By Value

```
int sum(int a,int b)
```

By Reference

```
int sum(int *a,int *b)
```

Practice (Parameter Passing)

```c
#include <stdio.h>
void swap(int *,int *);
int main(){
        int i,j;
        scanf("%d%d",&i,&j);
        printf("Originally i = %d, j = %d\n",i,j);
        swap (&i ,&j);
        printf("Now i = %d, j = %d\n",i,j);
        return 0;
}
void swap (int *a, int *b){
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```c
#include <stdio.h>

int main(){
        char A[4] = {'a','b','c','\0'};
        int i = 0;

        A[0] = 'A';
        printf("%c\n",*(A+2));
        for(i = 0; i<4 ; i++)
                printf("%c\n",A[i]);
        return 0;

}
```

Brackets specifies the count of elements. Elements are set in the braces.

Access element at index 0 in array

Print element at index 2 using pointer notation. A evaluates to the address of the first element in the array.

For loops that iterates over the elements.

Practice (Arrays)

```c
#include <stdio.h>

int main(){
        int array[10];
        int n;
        printf("Enter n: \n");
        scanf("%d",&n);
        int i = 0, sum = 0;
        for (i=0;i<n;i++)
        {
                printf("%d - Enter a number: \n",i);
                scanf("%d",&array[i]);
                sum +=array[i];
        }
        printf("Average = %f\n",(double) sum/n);
        return 0;
}
```

Practice (Arrays)

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

int main(){

        int *vec;
        vec = malloc(sizeof(int)*3);
        *(vec) = 1;
        *(vec + 1) = 2;
        *(vec + 2) = 3;
        printf("vec[2]=%d\n", *(vec + 2));
        free(vec);
        return 0;
}
```

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

int main(){

        int vec[3];
        vec[0] = 1;
        vec[1] = 2;
        vec[2] = 3;
        printf("vec[2]=%d\n", vec[2]);
        return 0;
}
```

Practice (Arrays)

Write a C program that searches for a number in an array of integers using the following function's signature: *bool search(int A[], int num, int size)*.

Practice (Arrays)

```c
#include <stdio.h>
#include <stdbool.h>
bool search(int[],int, int);
int main(){
        int size;
        printf("Enter the size of the array:\n");
        scanf("%d",&size);
        int arr[size];
        int i =0;
        printf("Enter the elements of the array\n");
        for (i=0 ; i<size;i++)
                scanf("%d",&arr[i]);
        printf("Enter the number you are searching for\n");
        int num ;
        scanf("%d", &num);
        if (search(arr, num, size))
                printf("FOUND\n");
        else printf("NOT FOUND\n");
}
bool search(int a[], int num, int size){
        int i = 0;
        for (i=0;i<size;i++)
                if(a[i]==num)
                        return true;
        return false;
}
```

Strings

A string in C is a one dimensional array of characters terminated by a null character '\0'.

```c
#include <stdio.h>

int main (){
        char str1[6] = {'H','e','l','l','o','\0'};
        printf("%s\n",str1);

        char str2[]="hello";
        printf("%s\n",str2);

        char *str3 = "hello";
        printf("%s\n",str3);

        return 0;
}
```

Create an array of characters terminated by a null character.

Create an array of characters using a string constant. Terminating character will be added automatically.

Create an array of characters using the pointer notation.

Strings

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

int main() {

  char buff[4];
  int age;
  printf("Enter your age:\n");

  if(fgets(buff, sizeof(buff), stdin) == NULL)
    return 1;

  /* ... validate input (new line, length, content ... */

  age = atoi(buff);

  printf("Your age is %d\n", age);
  return 0;
}
```

Scan a string from the user

Convert String to integer

Strings

sizeof (x): returns the number of bytes occupied by x

sizeof(int) : returns the number of bytes of an int which is 4 bytes in the 32-bit architecture

int a[10]; sizeof(a) = 10*sizeof(int)

strlen(str): returns the actual size of the str

strtok(str, "\n"): removes the trailing new line

atoi: array to integer

Practice (Strings)

```c
#include <stdio.h>
//#include <stdlib.h>
#include <string.h>

int strln (char[], int);
int main (){

        char str[10];
        if(fgets ( str, sizeof(str), stdin) == NULL)
                return 1;
        if(str [strlen (str) - 1] != '\n'){
                printf("Input is too long\n");
        }

        printf("str is %d" , strlen(str));
;


        return 0;
```

Structures

A Struct is a way to compose existing types into structure. Structures provide a way of storing many different values in variables of potentially different types under the same name.

Makes the program more modular.

Makes the program more compact and easier to modify.

In generally, structs are useful whenever a lot of data needs to be grouped together. For example, Store information about an address book.

Defining a structure: **struct struct_name name_of_a_single_struct;**

Accessing variable of the structure: **name_of_a_single_struct.name_of_variable;**

Structures (Practice)

```c
#include <stdio.h>

struct person{
        int age;
        float salary;
};
int main(){

        struct person bob;

        bob.age = 25;
        bob.salary = 900.5;

        printf("%s%d %.2f\n", "person bob: ", bob.age, bob.salary);

        struct person *bob_junior;
        bob_junior = &bob;

        printf("%s%d %.2f\n","person bob_junior", bob_junior->age, bob_junior->sallary);
        return 0;
}
```

- Defining a structure with 2 members.
- Create a single structure.
- Access the elements of the structure.
- Define a pointer to a structure.
- Point to the memory address of the “bob” structure.
- Use the “.” operator when dealing with a structure. Use “->” operator when dealing with a pointer to a structure.

Structures (Practice)

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

typedef struct {
  int year;
  int month;
```

g/cbootcamp/8_Pointers.html

```c
  int day;
} date;

int main(void) {

  date *today;
  today = malloc(sizeof(date));

  // the explicit way of accessing fields of our struct
  (*today).day = 15;
  (*today).month = 6;
  (*today).year = 2012;

  // the more readable shorthand way of doing it
  today->day = 15;
  today->month = 6;
  today->year = 2012;

  printf("the date is %d/%d/%d\n", today->day, today->month, today->year);

  free(today);

  return 0;
}
```

Structures (Practice)

The C keyword “typedef” is used to give a type a new name.

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

typedef unsigned char BYTE;

typedef struct persons{

        int age;
        char *name;
}Person;

int main(){
        BYTE a = 'a';
        printf("%c\n" , a);

        Person bob;
        bob.age = 25;
        bob.name = "bob";
        printf("%s %d\n",bob.name, bob.age);
        return 0;
}
```

Define a term BYTE for one byte.

Define a term Person for the structure.

Use the new term BYTE instead of “unsigned char”.

Use the new term Person to declare a structure instead of the previous notation we looked at.

Structures (Practice)

Write a C program that scans numbers from the user representing x and y coordinates of 2 points on an x-y axis. Create a Struct called "Point" to store the values. Create a function that takes 2 Point arguments by reference to check whether 2 points are equal or no

Note: if you want to use Boolean include <stdbool.h>

Declaring Boolean in c is for example
bool x = true;

Structures (Practice)

```c
#include <stdio.h>
//#include <stdlib.h>
#include <stdbool.h>
typedef struct p {
        int x;
        int y;

}Point;
bool check(Point*,Point*);
int main(){
        Point p1,p2;
        scanf("%d%d%d%d",&(p1.x),&(p1.y),&(p2.x),&(p2.y));
        if(check(&p1,&p2)) printf("True, same point");
        else printf("False, not same point");
        return 0;
}
bool check(Point *a, Point *b){
        if(a->x==b->x && a->y==b->y)
                return true;
        return false;
}
```

Dynamic Memory Allocation (using malloc)

Write a C program that:

1. Allocates an array of **n** integers and fills it with random integers between 0 and 1000. The size of the array, **n**, should be even and given as a command line argument.
2. Computes the sum of the array elements by a function called "Calculate_Array_Sum" that takes a pointer to an integer.
3. Computes the sum of only the second half of the array using the same function (without modifying its signature).

Dynamic Memory Allocation (using malloc)

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

int calculate_array_sum(int *, int);

int main(int argc, char*argv[]) {

    if (argc != 2) {
        printf("proper usage <%s> <length of array>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int array_size = atoi(argv[1]);

    int *array = (int*)malloc(array_size * sizeof(int));

    if (array == NULL) {
        perror("Error while allocating memory");
        exit(EXIT_FAILURE);
    }
```

Dynamic Memory Allocation (using malloc)

```
    srand(time(NULL));

    int i=0;

    for (i=0; i<array_size; i++)
        array[i] = rand() %1000 +1;

    for (i=0; i<array_size; i++)
        printf("%d ", array[i]);

    printf("\nSum of elements is %d\n", calculate_array_sum(array, array_size));

    printf("Sum of half of the element is %d\n", calculate_array_sum(array+array_size/2,
array_size/2));

    return 0;
}

int calculate_array_sum(int *A, int size) {

    int i=0;
    int sum=0;
    for (i=0; i<size; i++) {
        sum+=A[i];
    }
    return sum;
}
```

Dynamic Memory Allocation (using malloc)

Write a program that initializes a matrix of size **n*m** given as a command line argument and prints the matrix in a 2D readable format after filling it with random integers.

Sample Output

```
$ ./a.out 3 5
7 11 2 32 47
36 13 21 3 42
8 12 2 31 43
```

Dynamic Memory Allocation (using malloc)

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

int main( int argc, char *argv[] ) {

    if (argc != 3) {
        printf("proper usage <%s> <number of rows> <number of columns>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    int i=0, j=0;

    int **matrix = (int**)malloc(x * sizeof(int *));

    if (matrix == NULL) {
        perror("Error in allocating memory for the matrix");
        exit(EXIT_FAILURE);
    }
```

Dynamic Memory Allocation (using malloc)

```
for (j=0; j<x; j++)
        matrix[j] = (int*)malloc(y * sizeof(int));

    if (matrix[j] == NULL) {
        perror("Error in allocating memory for the matrix");
        exit(EXIT_FAILURE);
    }

    for (i=0; i<x; i++) {
        for (j=0; j<y; j++) {
            matrix[i][j] = j;
        }
    }

    for (i=0; i<x; i++) {
        for (j=0; j<y; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    for (i=0; i<x; i++)
        free(matrix[i]);
    free(matrix);

    return 0;
}
```