C

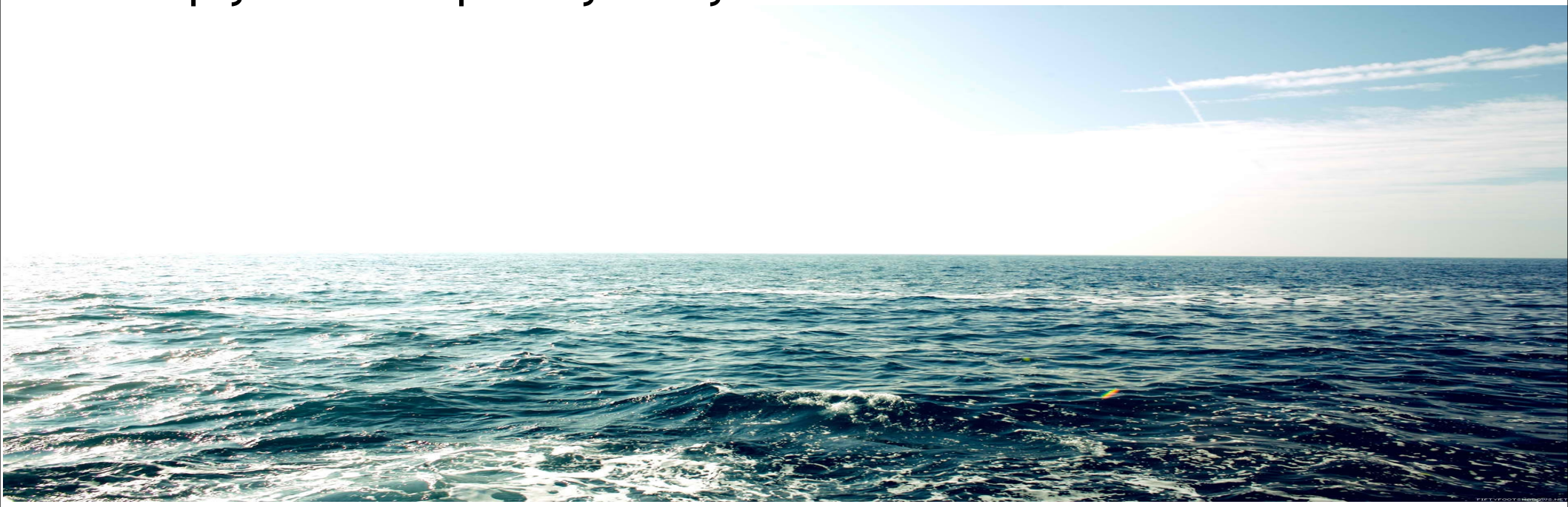
Come let's see how deep it is!!

Team Emertxe

Introduction

C

Have you ever pondered how
- powerful it is?
- efficient it is?
- flexible it is?
- deep you can explore your system?



C

Where is it used?

- System Software Development
- Embedded Software Development
- OS Kernel Development
- Firmware, Middle-ware and Driver Development
- File System Development

And many more!!

C

Important Characteristics

- It is a general-purpose language, even though it is applied and used effectively in various specific domains
- It is a free-formatted language (and not a strongly-typed language)
- Efficiency and portability are the important considerations
- Library facilities play an important role

C

Keywords - Categories

Type	Keyword
Data Types	char int float double
Modifiers	signed unsigned short long
Qualifiers	const volatile
Loops	for while do
Jump	goto break continue

Type	Keyword
Decision	if else switch case default
Storage Class	auto register static extern
Derived	struct unions
User defined	enums typedefs
Others	void return sizeof

C

Typical C Code Contents

Documentation

Preprocessor Statements

Global Declaration

The Main Code:

Local Declarations
Program Statements
Function Calls

One or many Function(s):

The function body

- A typical code might contain the blocks shown on left side
- It is generally recommended to practice writing codes with all the blocks

C

Anatomy of a Simple C Code

```
/* My first C code */
#include <stdio.h>
int main()
{
    /* To display Hello world */
    printf("Hello world\n");
    return 0;
}
```

File Header

Preprocessor Directive

The start of program

Comment

Statement

Program Termination

Data Representations

Embedded C

Number Systems

- A number is generally represented as
  - Decimal
  - Octal
  - Hexadecimal
  - Binary

Type	Range (8 Bits)
Decimal	0 - 255
Octal	000 - 0377
Hexadecimal	0x00 - 0xFF
Binary	0b00000000 - 0b11111111

Type	Dec	Oct	Hex	Bin
Base	10	8	16	2
	0	0	0	0 0 0 0
	1	1	1	0 0 0 1
	2	2	2	0 0 1 0
	3	3	3	0 0 1 1
	4	4	4	0 1 0 0
	5	5	5	0 1 0 1
	6	6	6	0 1 1 0
	7	7	7	0 1 1 1
	8	10	8	1 0 0 0
	9	11	9	1 0 0 1
	10	12	A	1 0 1 0
	11	13	B	1 0 1 1
	12	14	C	1 1 0 0
	13	15	D	1 1 0 1
	14	16	E	1 1 1 0
	15	17	F	1 1 1 1

Embedded C

Data Representation - Bit

- Literally computer understand only two states HIGH and LOW making it a binary system
- These states are coded as 1 or 0 called binary digits
- “**Bi**nary Digi**t**” gave birth to the word “**Bit**”
- Bit is known a basic unit of information in computer and digital communication

Value	No of Bits
0	0
1	1

Embedded C

Data Representation - Byte

- A unit of digital information
- Commonly consist of 8 bits
- Considered smallest addressable unit of memory in computer

Value	No of Bits
0	0 0 0 0 0 0 0 0
1	0 0 0 0 0 0 0 1

Embedded C

Data Representation - Word

- Amount of data that a machine can fetch and process at one time
- An integer number of bytes, for example, one, two, four, or eight
- General discussion on the bitness of the system is references to the word size of a system, i.e., a 32 bit chip has a 32 bit (4 Bytes) word size

Value	No of Bits
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Embedded C

Integer Number - Positive

- Integers are like whole numbers, but allow negative numbers and no fraction
- An example of 13_{10} in 32 bit system would be

Bit	No of Bits																													
Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1

Embedded C

Integer Number - Negative

- Negative Integers represented with the 2's complement of the positive number
- An example of -13_{10} in 32 bit system would be

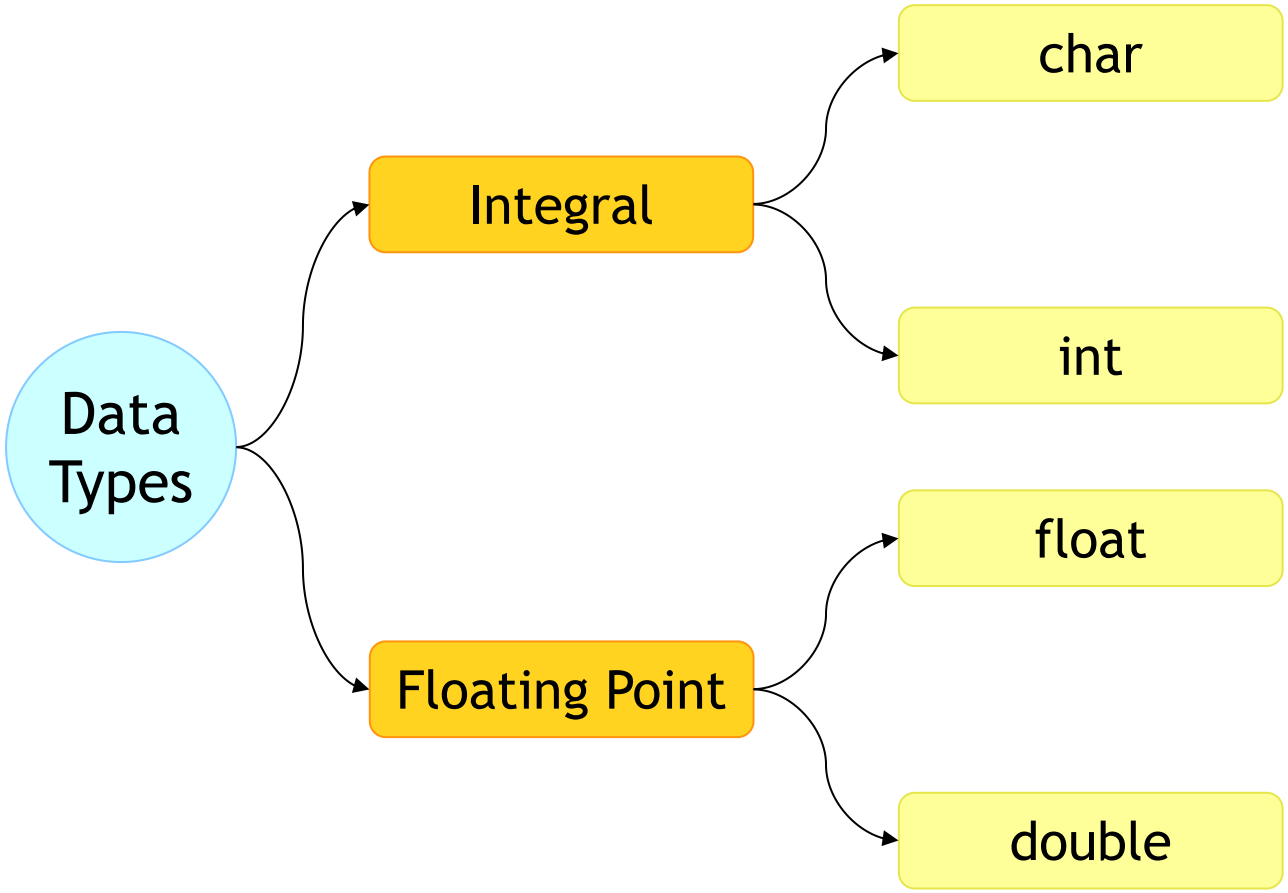
Bit	No of Bits																													
Position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
1's Compli	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0
Add 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
2's Compli	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1

- Mathematically : $-k \equiv 2^n - k$

Basic Data Types

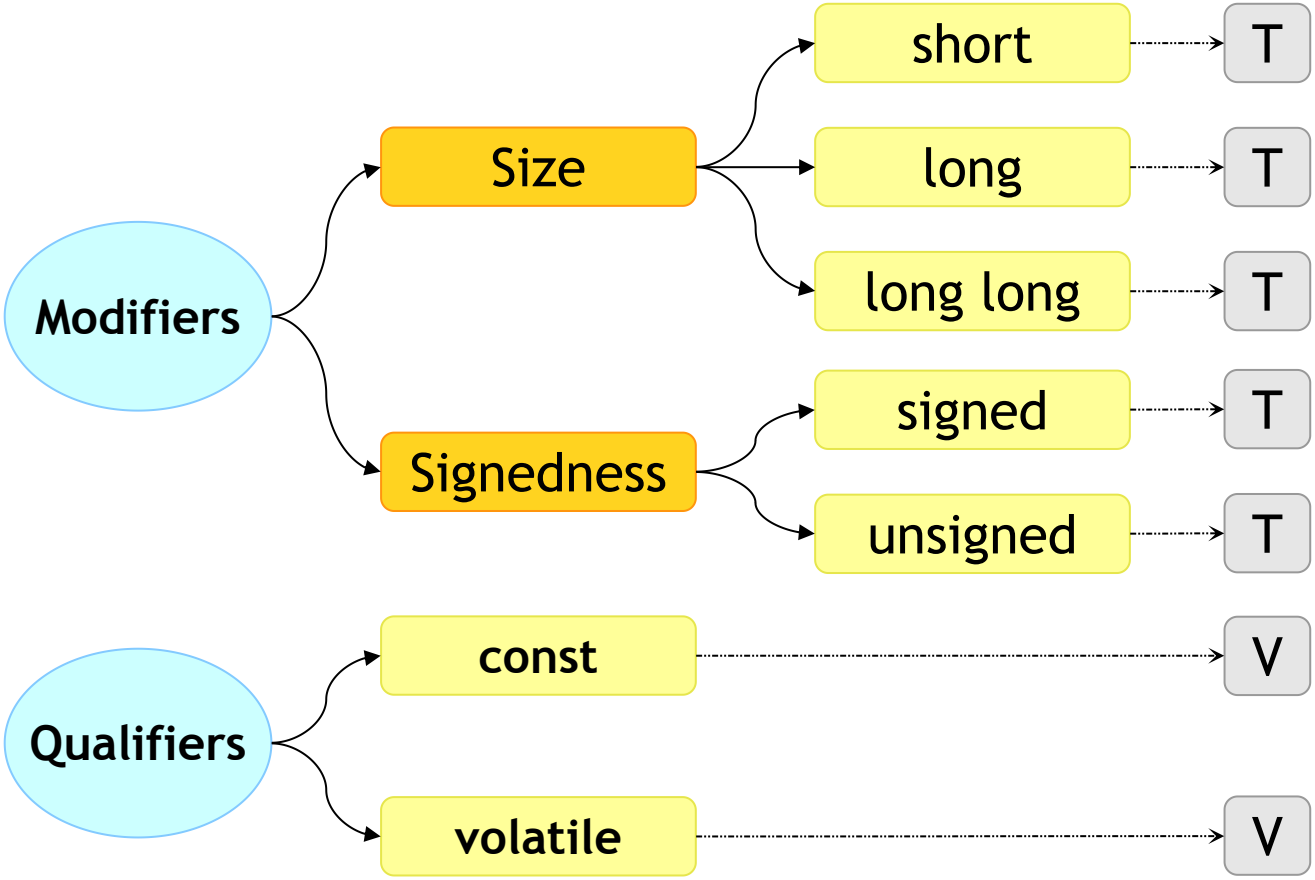
Embedded C

Basic Data Types



Embedded C

Data Type Modifiers and Qualifiers



V Variables

T Data Types

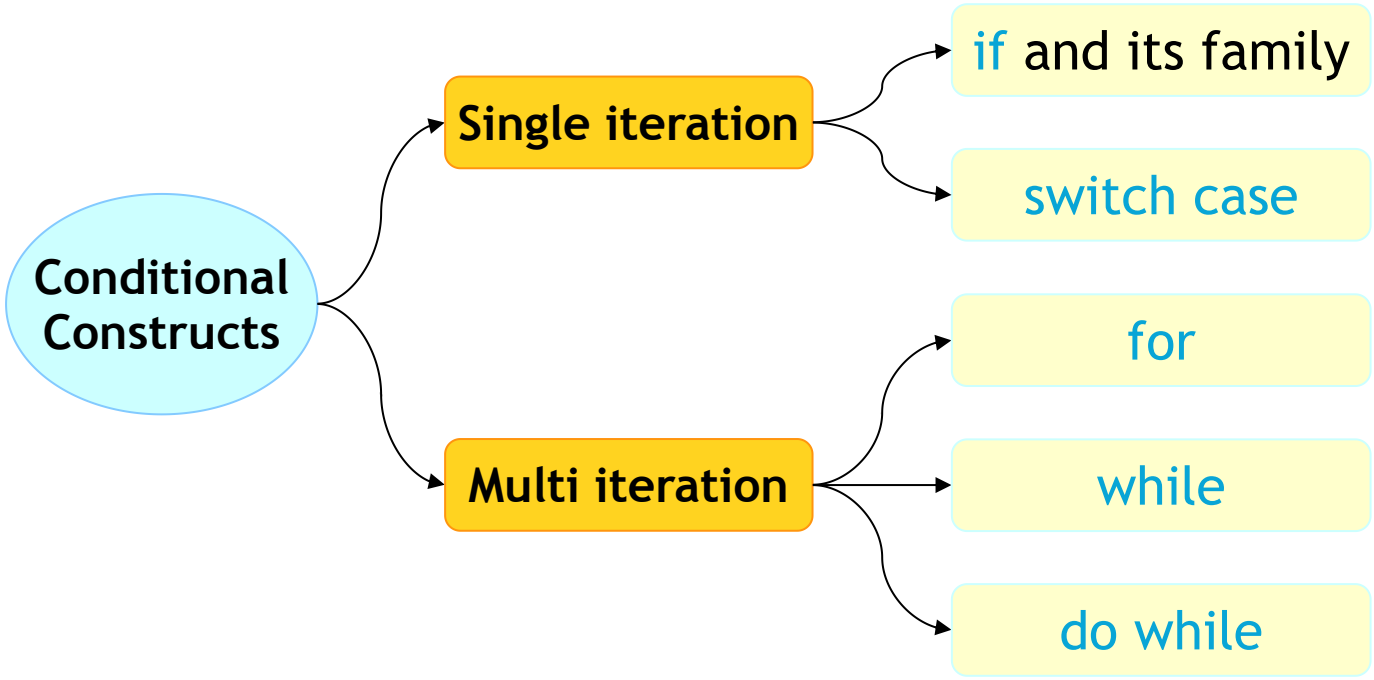
F Functions

- ANSI says, ensure that: **char ≤ short ≤ int ≤ long**

Conditional Constructs

Embedded C

Conditional Constructs



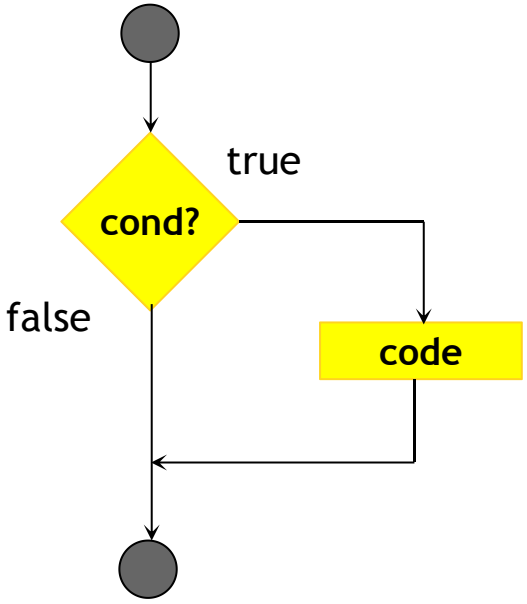
Embedded C

Conditional Constructs - if

Syntax

```
if (condition)
{
    statement(s);
}
```

Flow



Example

```
#include <stdio.h>

int main()
{
    int num1 = 2;

    if (num1 < 5)
    {
        printf("num1 < 5\n");
    }
    printf("num1 is %d\n", num1);

    return 0;
}
```

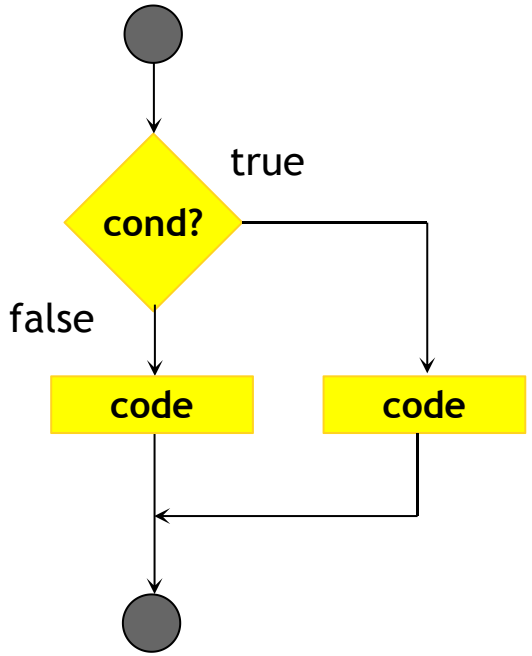
Embedded C

Conditional Constructs - if else

Syntax

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

Flow



Embedded C

Conditional Constructs - if else

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10;

    if (num1 < 5)
    {
        printf("num1 < 5\n");
    }
    else
    {
        printf("num1 > 5\n");
    }

    return 0;
}
```

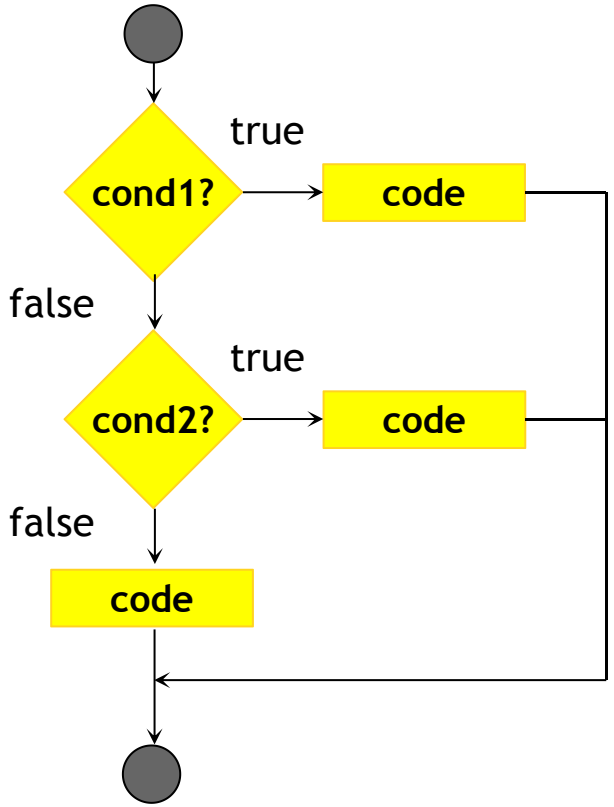
Embedded C

Conditional Constructs - if else if

Syntax

```
if (condition1)
{
    statement(s);
}
else if (condition2)
{
    statement(s);
}
else
{
    statement(s);
}
```

Flow



Embedded C

Conditional Constructs - if else if

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10;

    if (num1 < 5)
    {
        printf("num1 < 5\n");
    }
    else if (num1 > 5)
    {
        printf("num1 > 5\n");
    }
    else
    {
        printf("num1 = 5\n");
    }


    return 0;
}
```

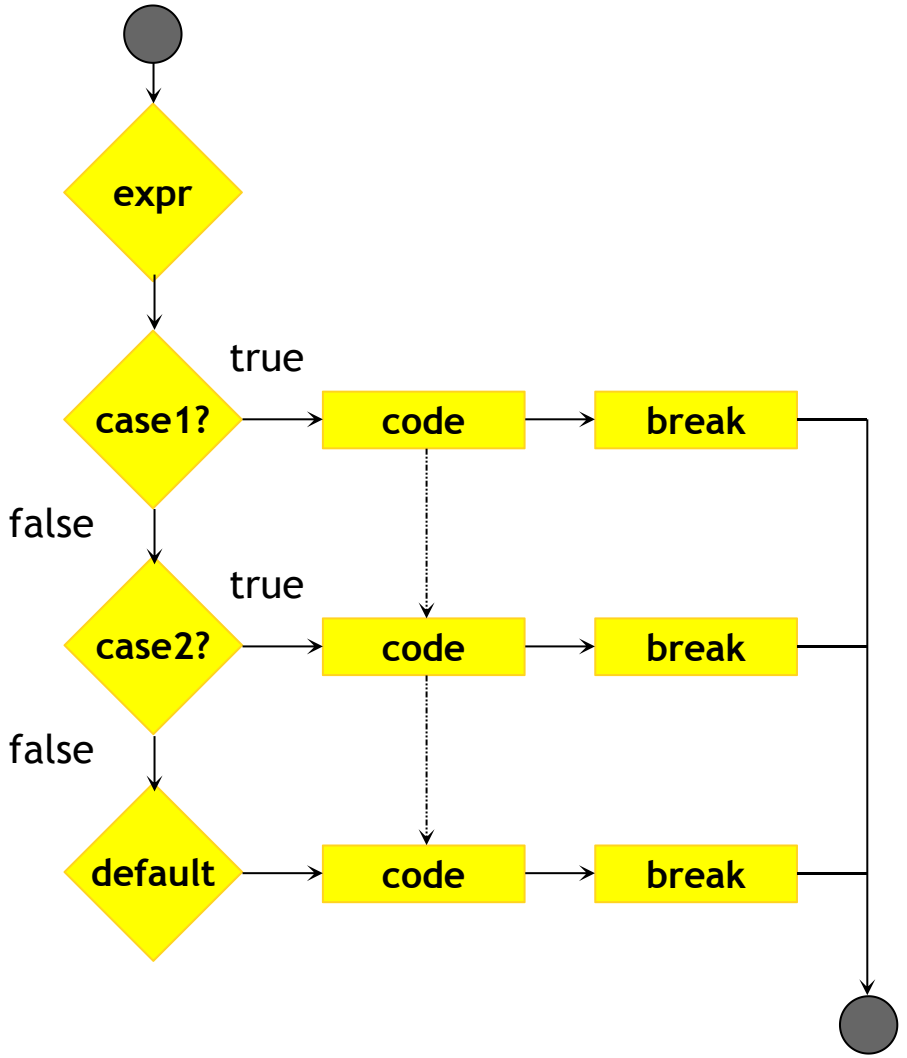
Embedded C

Conditional Constructs - switch

Syntax

```
switch (expression)
{
    case constant:
        statement(s);
        break;
    case constant:
        statement(s);
        break;
    case constant:
        statement(s);
        break;
    default:
        statement(s);
}
```

Flow



Embedded C

Conditional Constructs - switch

Example

```c
#include <stdio.h>

int main()
{
    int option;
    printf("Enter the value\n");
    scanf("%d", &option);

    switch (option)
    {
        case 10:
            printf("You entered 10\n");
            break;
        case 20:
            printf("You entered 20\n");
            break;
        default:
            printf("Try again\n");
    }

    return 0;
}
```

Embedded C

Conditional Constructs - while

Syntax

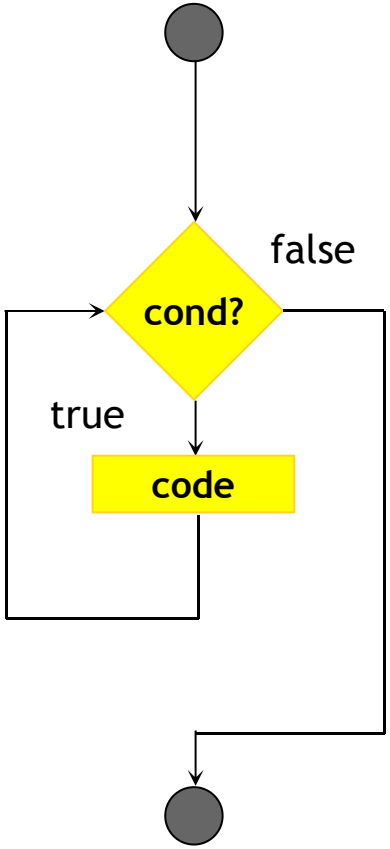
```
while (condition)
{
    statement(s);
}
```

- **Controls** the loop.
- Evaluated **before** each execution of loop body

Example

```
#include <stdio.h>

int main()
{
    int i;

    i = 0;
    while (i < 10)
    {
        printf("Looped %d times\n", i);
        i++;
    }

    return 0;
}
```

Flow



Embedded C

Conditional Constructs - do while

Syntax

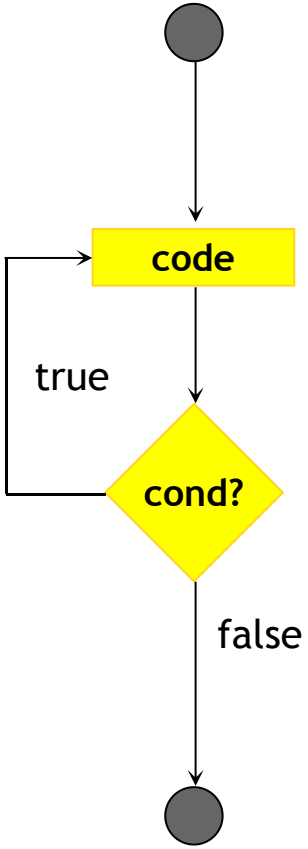
```
do
{
    statement(s);
} while (condition);
```

- **Controls** the loop.
- Evaluated **after** each execution of loop body

Example

```
#include <stdio.h>

int main()
{
    int i;

    i = 0;
    do
    {
        printf("Looped %d times\n", i);
        i++;
    } while (i < 10);

    return 0;
}
```

Flow



Embedded C

Conditional Constructs - for

Syntax

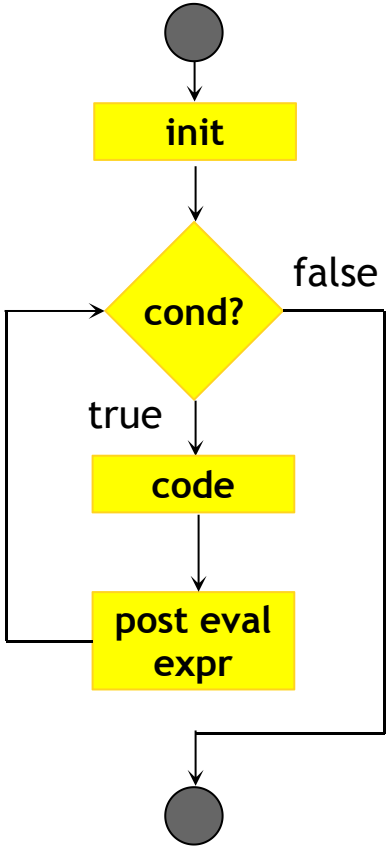
```
for (init; condition; post evaluation expr)
{
    statement(s);
}
```

- **Controls** the loop.
- Evaluated **before** each execution of loop body

Example

```
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 10; i++)
    {
        printf("Looped %d times\n", i);
    }

    return 0;
}
```

Flow



Embedded C

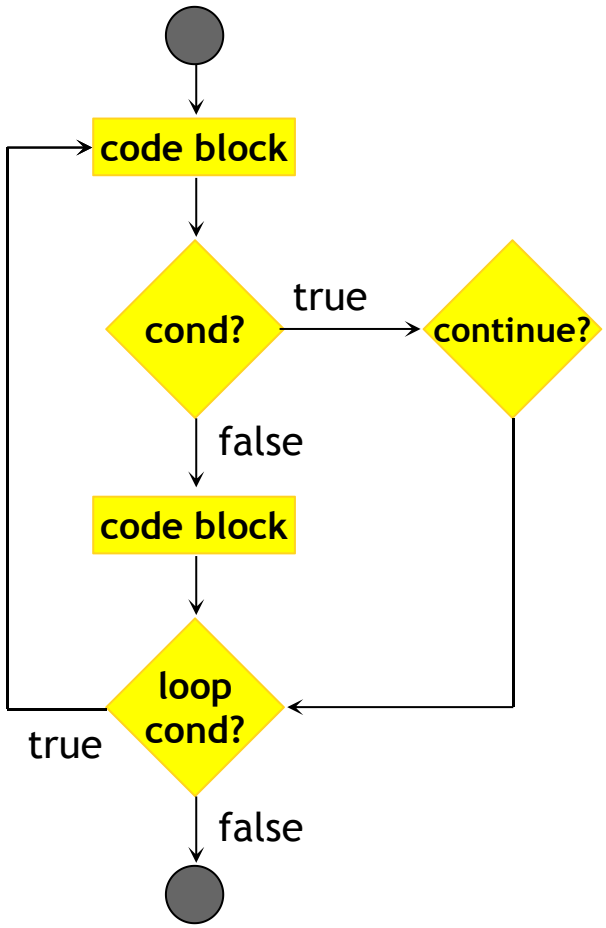
Conditional Constructs - continue

- A *continue* statement causes a jump to the loop-continuation portion, that is, to the end of the loop body
- The execution of code appearing after the continue will be skipped
- Can be used in any type of multi iteration loop

Syntax

```
do
{
    conditional statement
        continue;
} while (condition);
```

Flow



Embedded C

Conditional Constructs - continue

Example

```c
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            continue;
        }
        printf("%d\n", i);
    }

    return 0;
}
```

Embedded C

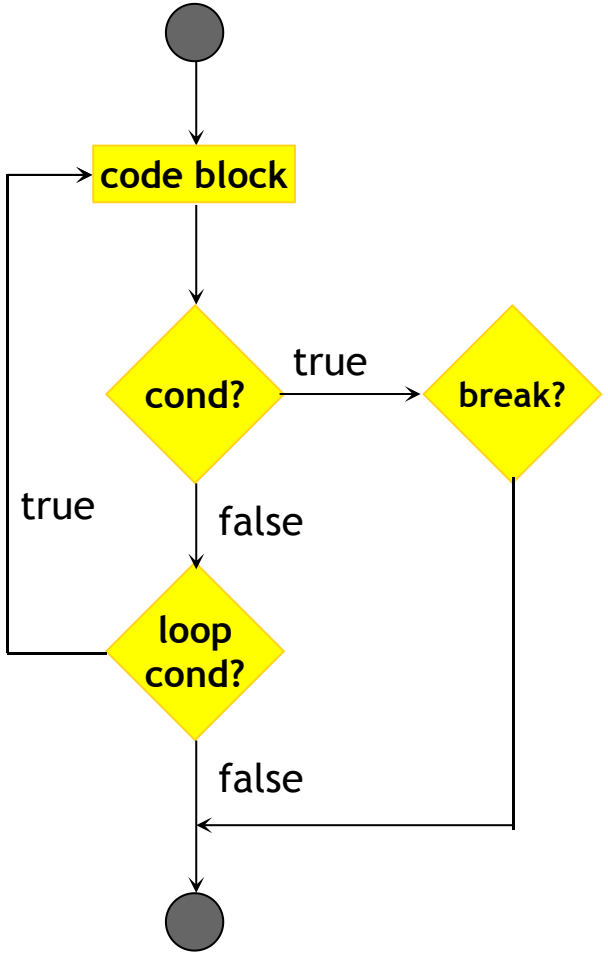
Conditional Constructs – break

- A break statement shall appear only in “switch body” or “loop body”
- *“break”* is used to exit the loop, the statements appearing after break in the loop will be skipped

Syntax

```
do
{
    conditional statement
        break;
} while (condition);
```

Flow



Embedded C

Conditional Constructs - break

Example

```c
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            break;
        }
        printf("%d\n", i);
    }

    return 0;
}
```

Embedded C

Conditional Constructs - break

Example

```c
#include <stdio.h>

int main()
{
    int i;

    for (i = 0; i < 10; i++)
    {
        if (i == 5)
        {
            break;
        }
        printf("%d\n", i);
    }
    printf("%d\n", i);

    return 0;
}
```

Operators

Embedded C

Operators

- Symbols that instructs the compiler to perform specific arithmetic or logical operation on operands
- All C operators do 2 things
  - Operates on its Operands
  - Returns a value

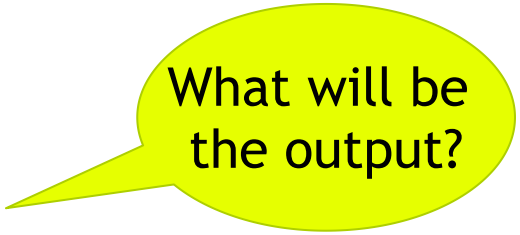
Embedded C

Operators - Arithmetic

Operator	Description	Associativity
* / %	Multiplication Division Modulo	L to R
+ -	Addition Subtraction	R to L

Example

```
#include <stdio.h>

int main()
{
    int num1 = 0, num2 = 0;

    printf("sum is %d\n", num1++ + ++num2);

    return 0;
}
```



Embedded C

Operators - Logical

Operator	Description	Associativity
!	Logical NOT	R to L
&&	Logical AND	L to R
||	Logical OR	L to R

Example

```
#include <stdio.h>

int main()
{
     int num1 = 1, num2 = 0;

     if (++num1 || num2++)
     {
          printf("num1 is %d num2 is %d\n", num1, num2);
     }
     num1 = 1, num2 = 0;
     if (num1++ && ++num2)
     {
          printf("num1 is %d num2 is %d\n", num1, num2);
     }
     else
     {
          printf("num1 is %d num2 is %d\n", num1, num2);
     }
     return 0;
}
```

What will be the output?

Embedded C

Operators - Relational

Operator	Description	Associativity
>	Greater than	L to R
<	Lesser than	
>=	Greater than or equal	
<=	Lesser than or equal	
==	Equal to	
!=	Not Equal to	

Example

```
#include <stdio.h>

int main()
{
    float num1 = 0.7;

    if (num1 == 0.7)
    {
        printf("Yes, it is equal\n");
    }
    else
    {
        printf("No, it is not equal\n");
    }

    return 0;
}
```

What will be the output?

Embedded C

Operators - Assignment

Example

```c
#include <stdio.h>

int main()
{
     int num1 = 1, num2 = 1;
     float num3 = 1.7, num4 = 1.5;

     num1 += num2 += num3 += num4;

     printf("num1 is %d\n", num1);

     return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
     float num1 = 1;

     if (num1 = 1)
     {
          printf("Yes, it is equal!!\n");
     }
     else
     {
          printf("No, it is not equal\n");
     }

     return 0;
}
```

Embedded C

Operators - Bitwise

- Bitwise operators perform operations on bits
- The operand type shall be integral
- Return type is integral value

Embedded C

Operators - Bitwise

&	**Bitwise AND**	Bitwise ANDing of all the bits in two operands
|	**Bitwise OR**	Bitwise ORing of all the bits in two operands
^	**Bitwise XOR**	Bitwise XORing of all the bits in two operands

Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
B	0x13	0 0 0 1 0 0 1 1
A & B	0x01	0 0 0 0 0 0 0 1

Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
B	0x13	0 0 0 1 0 0 1 1
A | B	0x73	0 1 1 1 0 0 1 1

Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
B	0x13	0 0 0 1 0 0 1 1
A ^ B	0x72	0 1 1 1 0 0 1 0

Embedded C

Operators - Bitwise

~	**Compliment**	Complimenting all the bits of the operand
>>	**Right Shift**	Shift all the bits right n times by introducing zeros left
<<	**Left Shift**	Shift all the bits left n times by introducing zeros right

Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
~A	0x9E	1 0 0 1 1 1 1 0

Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
A >> 2	0x18	0 0 0 1 1 0 0 0

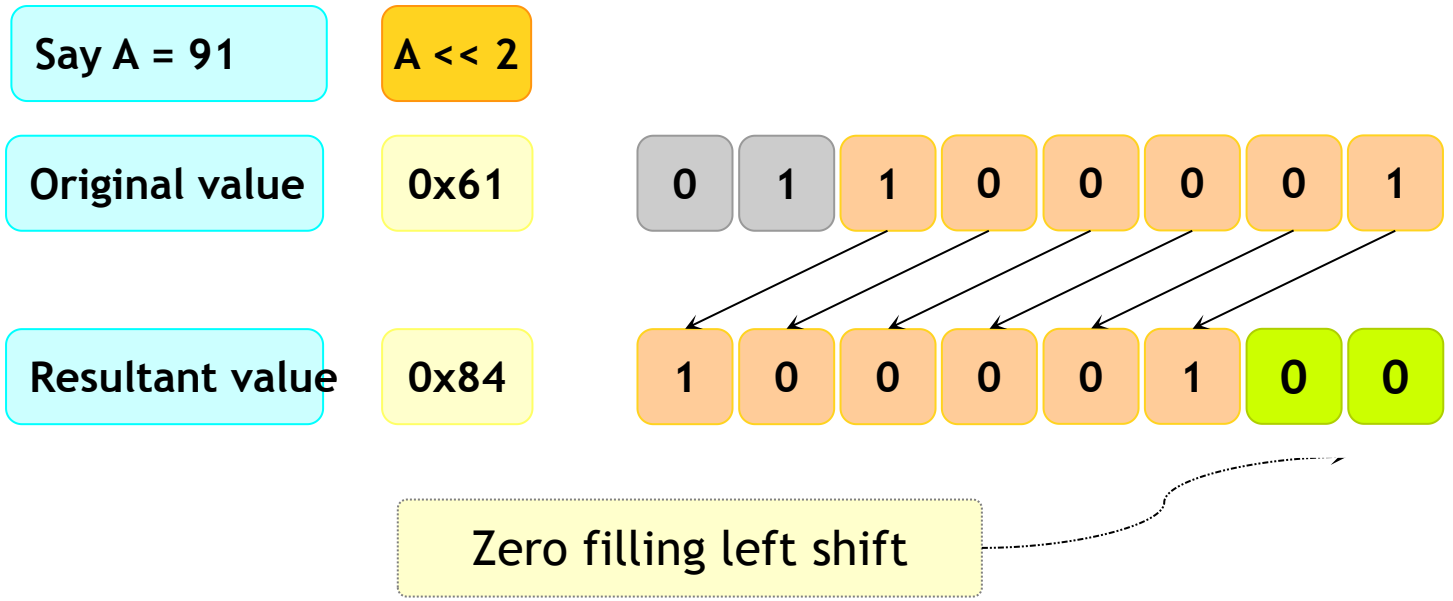
Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
A << 2	0x84	1 0 0 0 0 1 0 0

Embedded C

Operators – Bitwise – Left Shift

'Value' << 'Bits Count'

- Value : Is shift operand on which bit shifting effect to be applied
- Bits count : By how many bit(s) the given “Value” to be shifted

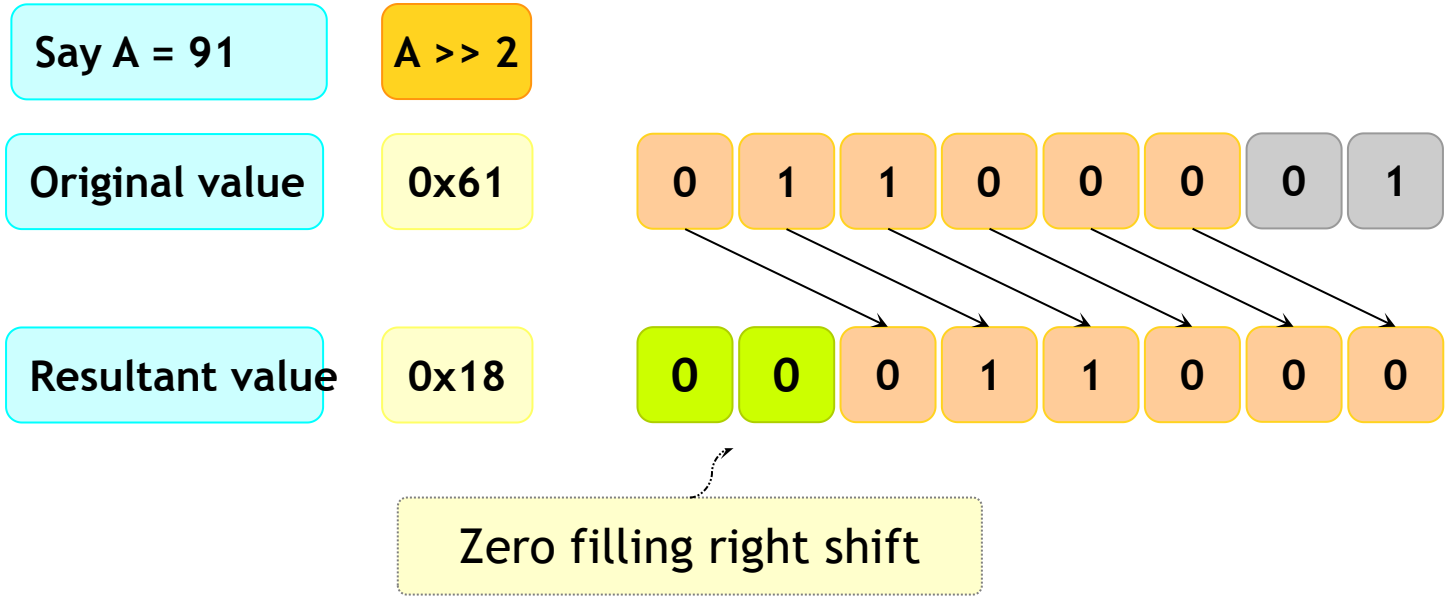


Embedded C

Operators – Bitwise – Right Shift

'Value' >> 'Bits Count'

- Value : Is shift operand on which bit shifting effect to be applied
- Bits count : By how many bit(s) the given “Value” to be shifted

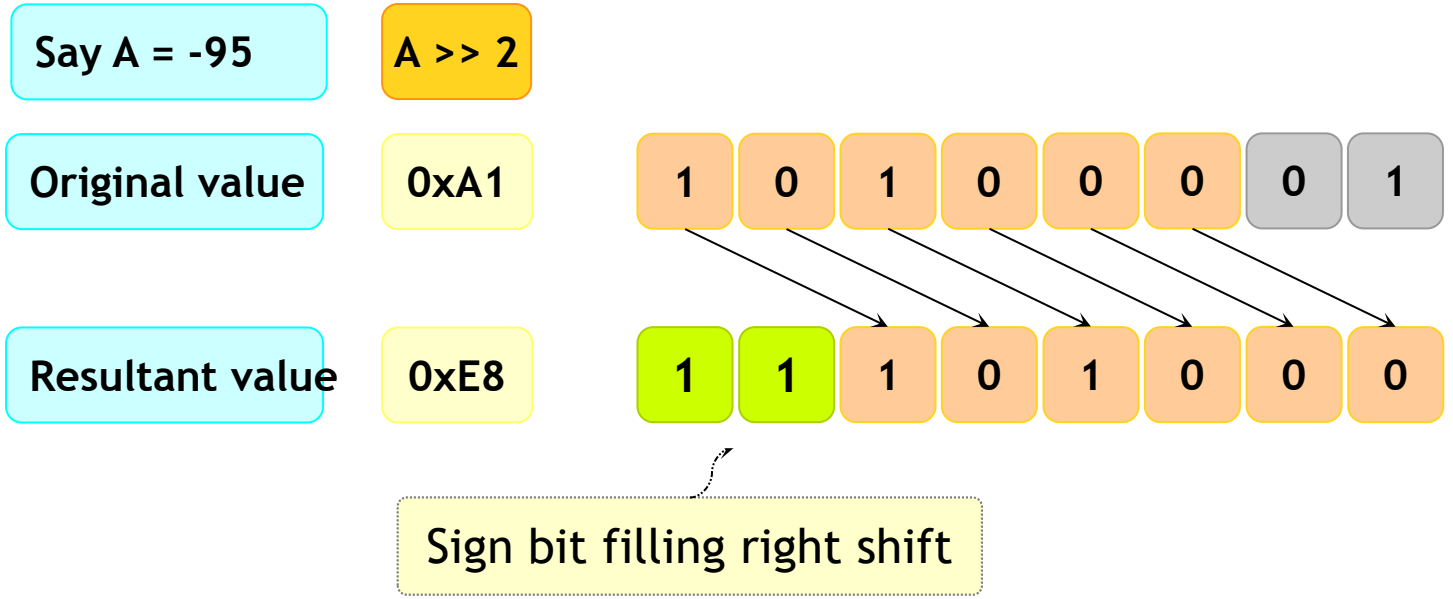


Embedded C

Operators – Bitwise – Right Shift – Signed Valued

“Signed Value' >> 'Bits Count'

- Same operation as mentioned in previous slide.
- But the sign bits gets propagated.



Embedded C

Operators - Language - sizeof()

Example

```c
#include <stdio.h>

int main()
{
    int num = 5;

    printf("%u:%u:%u\n", sizeof(int), sizeof num, sizeof 5);

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 5;
    int num2 = sizeof(++num1);

    printf("num1 is %d and num2 is %d\n", num1, num2);

    return 0;
}
```

Embedded C

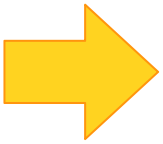
Operators - Ternary

Syntax

```
Condition ? Expression 1 : Expression 2;
```

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3;

    if (num1 > num2)
    {
        num3 = num1;
    }
    else
    {
        num3 = num2;
    }
    printf("%d\n", num3);

    return 0;
}
```

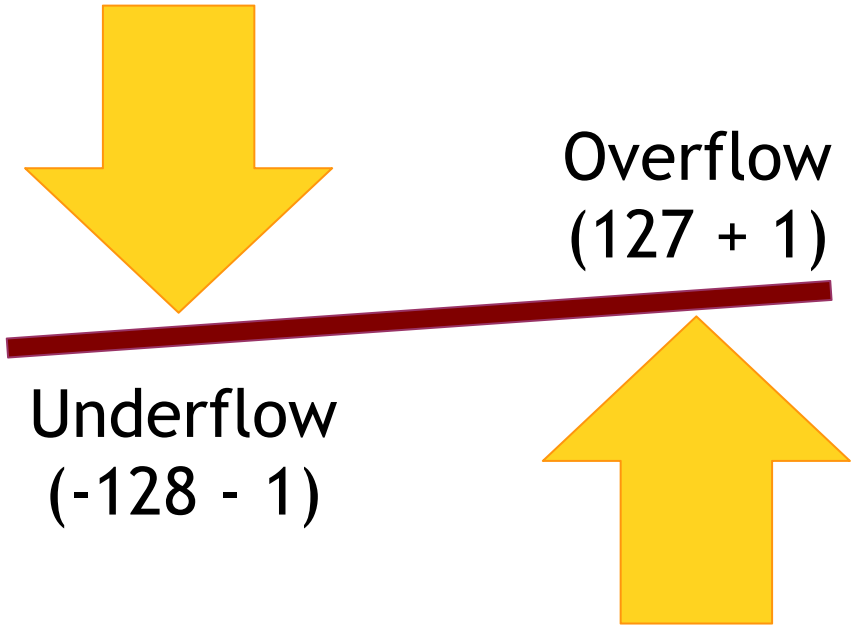


```c
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3;

    num3 = num1 > num2 ? num1 : num2;
    printf("Greater num is %d\n", num3);

    return 0;
}
```

Embedded C

Over and Underflow

- 8-bit Integral types can hold certain ranges of values
- So what happens when we try to traverse this boundary?



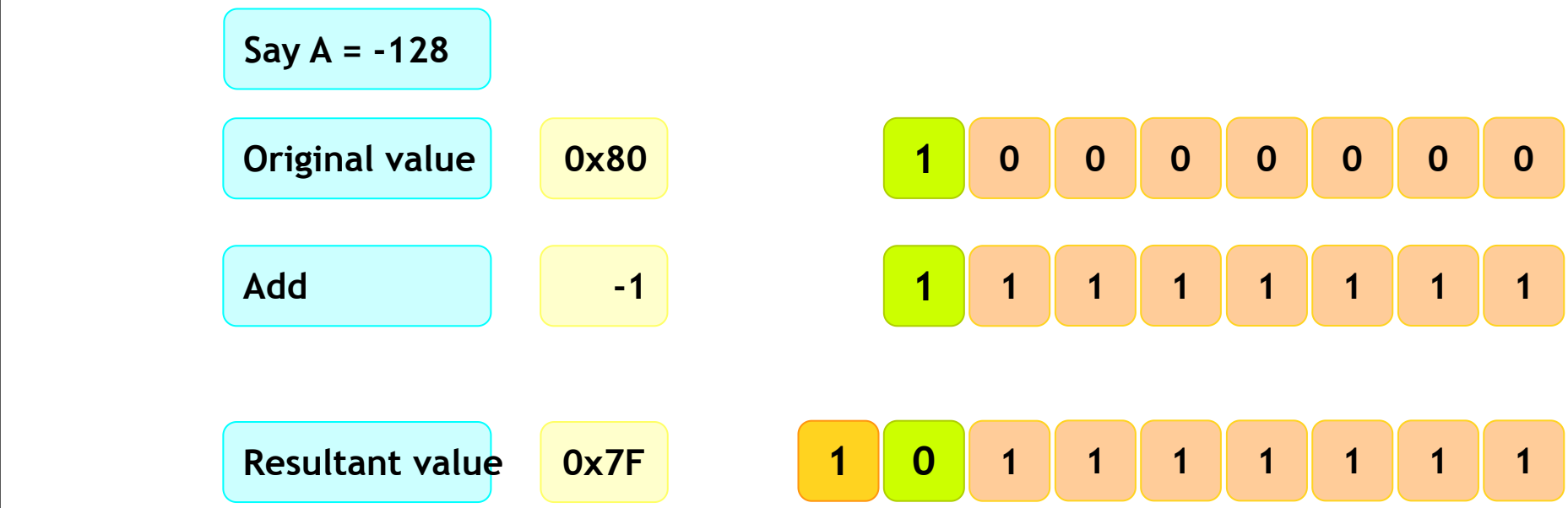
Embedded C

Overflow – Signed Numbers

Say A = +127		
Original value	0x7F	0 1 1 1 1 1 1 1
Add	1	0 0 0 0 0 0 0 1
Resultant value	0x80	1 0 0 0 0 0 0 0

Embedded C

Underflow – Signed Numbers



Array

Embedded C

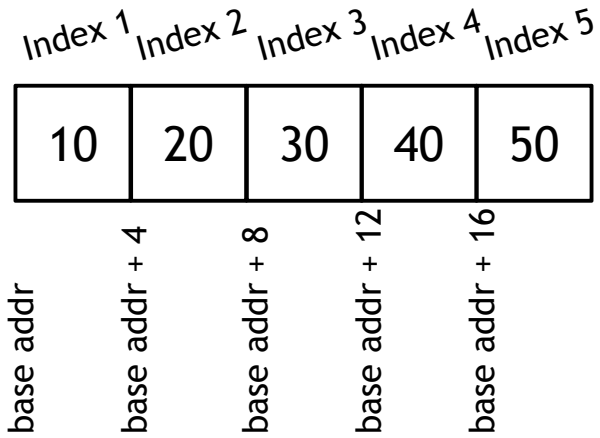
Arrays

Syntax

```
data_type name[SIZE];

Where SIZE is number of elements
The size for the array would be SIZE * <size of data_type>
```

Example

```
int age[5] = {10, 20, 30, 40, 50};
```



Embedded C

Arrays – Point to be noted

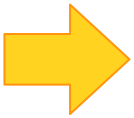
- An array is a collection of data of same data type.
- Addresses are sequential
- First element with lowest address and the last element with highest address
- Indexing starts from 0 and should end at array SIZE – 1. Example say array[5] will have to be indexed from 0 to 4
- Any access beyond the boundaries would be illegal access Example, You should not access array[-1] or array[SIZE]

Embedded C

Arrays - Why?

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3 = 30;
    int num4 = 40;
    int num5 = 50;

    printf("%d\n", num1);
    printf("%d\n", num2);
    printf("%d\n", num3);
    printf("%d\n", num4);
    printf("%d\n", num5);

    return 0;
}
```



```c
#include <stdio.h>

int main()
{
    int num_array[5] = {10, 20, 30, 40, 50};
    int index;

    for (index = 0; index < 5; index++)
    {
        printf("%d\n", num_array[index]);
    }

    return 0;
}
```

Embedded C

Arrays - Reading

Example

```c
#include <stdio.h>

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

    index = 0;
    do
    {
        printf("Index %d has Element %d\n", index, array[index]);
        index++;
    } while (index < 5);

    return 0;
}
```

Embedded C

Arrays - Storing

Example

```c
#include <stdio.h>

int main()
{
    int num_array[5];
    int index;

    for (index = 0; index < 5; index++)
    {
        scanf("%d", &num_array[index]);
    }

    return 0;
}
```

Embedded C

Arrays - Initializing

Example

```c
#include <stdio.h>

int main()
{
    int array1[5] = {1, 2, 3, 4, 5};
    int array2[5] = {1, 2};
    int array3[] = {1, 2};
    int array4[]; /* Invalid */

    printf("%u\n", sizeof(array1));
    printf("%u\n", sizeof(array2));
    printf("%u\n", sizeof(array3));

    return 0;
}
```

Embedded C

Arrays - Copying

- Can we copy 2 arrays? If yes how?

Example

```c
#include <stdio.h>

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

    array_bak = array_org;

    if (array_bak == array_org)
    {
        printf("Copied\n");
    }

    return 0;
}
```

Pointers

Embedded C

Pointers – Why?

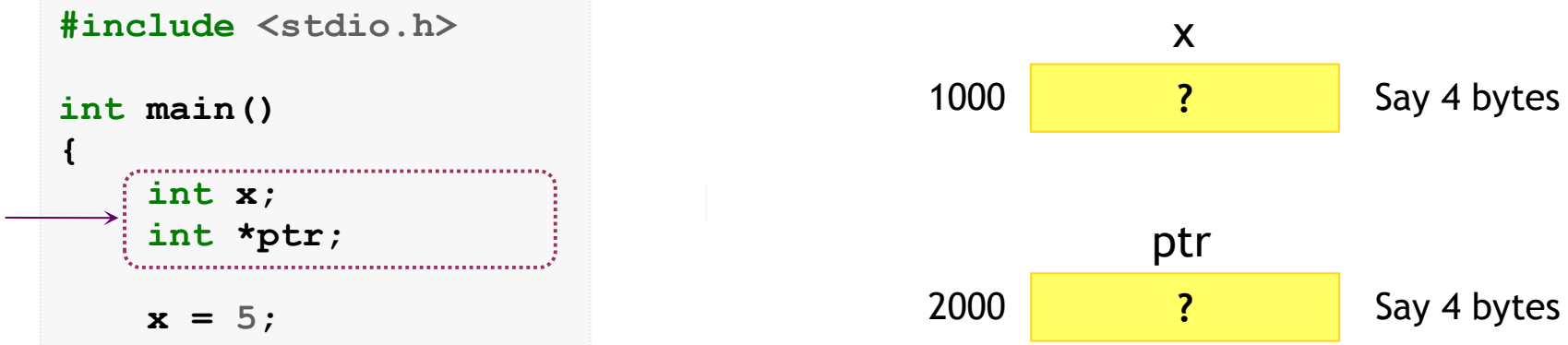
- To have C as a low level language being a high level language
- Returning more than one value from a function
- To achieve the similar results as of ”pass by variable”
- parameter passing mechanism in function, by passing the reference

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```

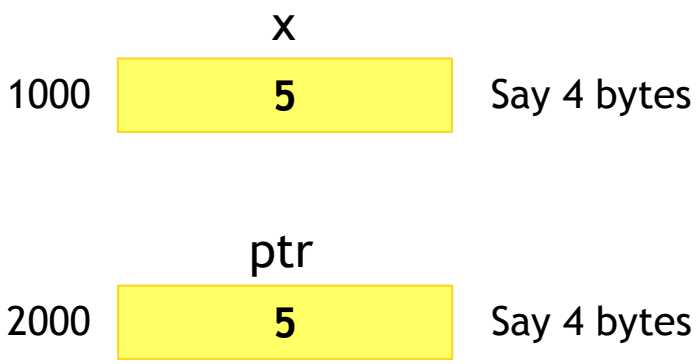


Embedded C

Pointers

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```



- So pointer is an integer
- But remember the “They may not be of same size”

32 bit system = 4 Bytes

64 bit system = 8 Bytes

Embedded C

Pointers

- Rule : “Referencing and Dereferencing”

&

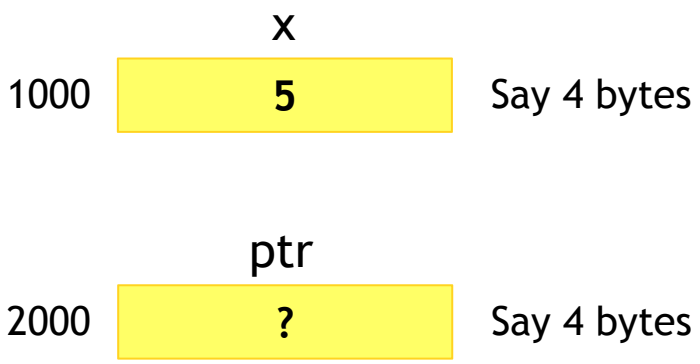
*

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

* 1000

- Goto to the location 1000 and fetch its value, so

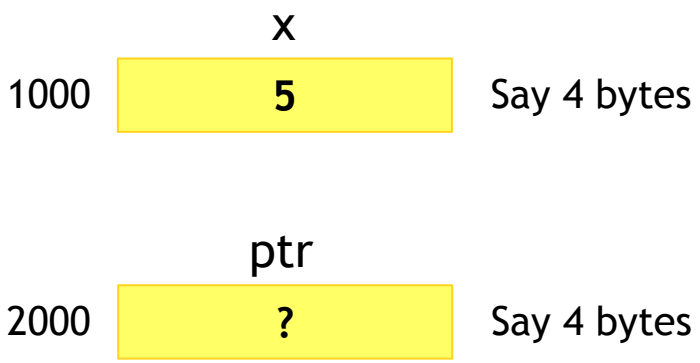
* 1000 → 5

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```



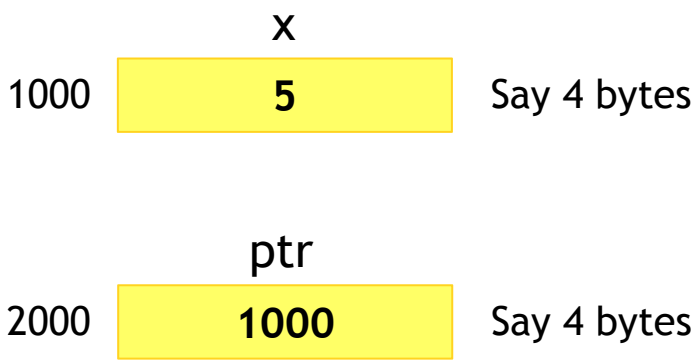
- What should be the change in the above diagram for the above code?

Embedded C

Pointers

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```



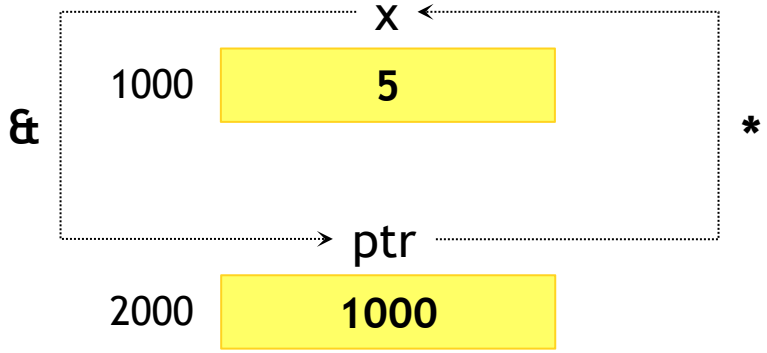
- So pointer should contain the address of a variable
- It should be a valid address

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x;

    return 0;
}
```



"Add a & on variable to store its address in a pointer"

"Add a * on the pointer to extract the value of variable it is pointing to"

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("Address of number is %p\n", &number);
    printf("ptr contains %p\n", ptr);

    return 0;
}
```

Embedded C

Pointers

Example

```c
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

Embedded C

Pointers

Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;
    *ptr = 100;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

- By compiling and executing the above code we can conclude

  "*ptr = number"

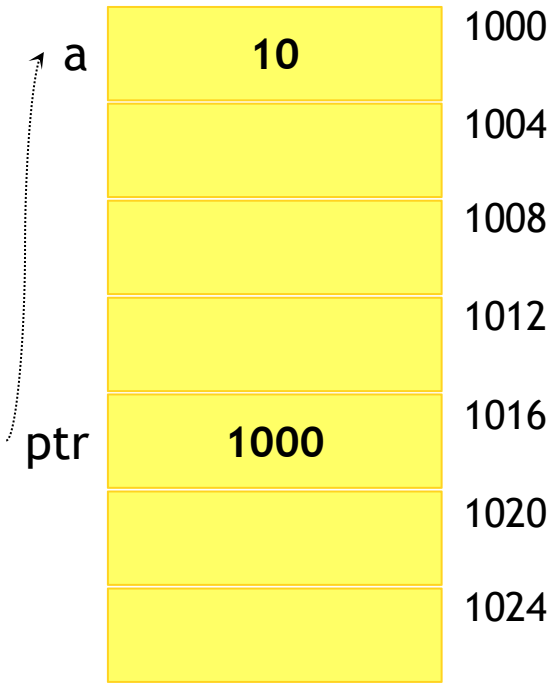
Embedded C

Pointers

- Pointer pointing to a Variable = Pointer contains the Address of the Variable
- Rule: “Pointing means Containing”

Example

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int *ptr;

    ptr = &a;

    return 0;
}
```



Embedded C

Pointers

- Since the discussion is on the data fetching, its better we have knowledge of storage concept of machines
- The Endianness of the machine
- What is this now!!?
  - Its nothing but the byte ordering in a word of the machine
- There are two types
  - Little Endian – LSB in Lower Memory Address
  - Big Endian – MSB in Lower Memory Address

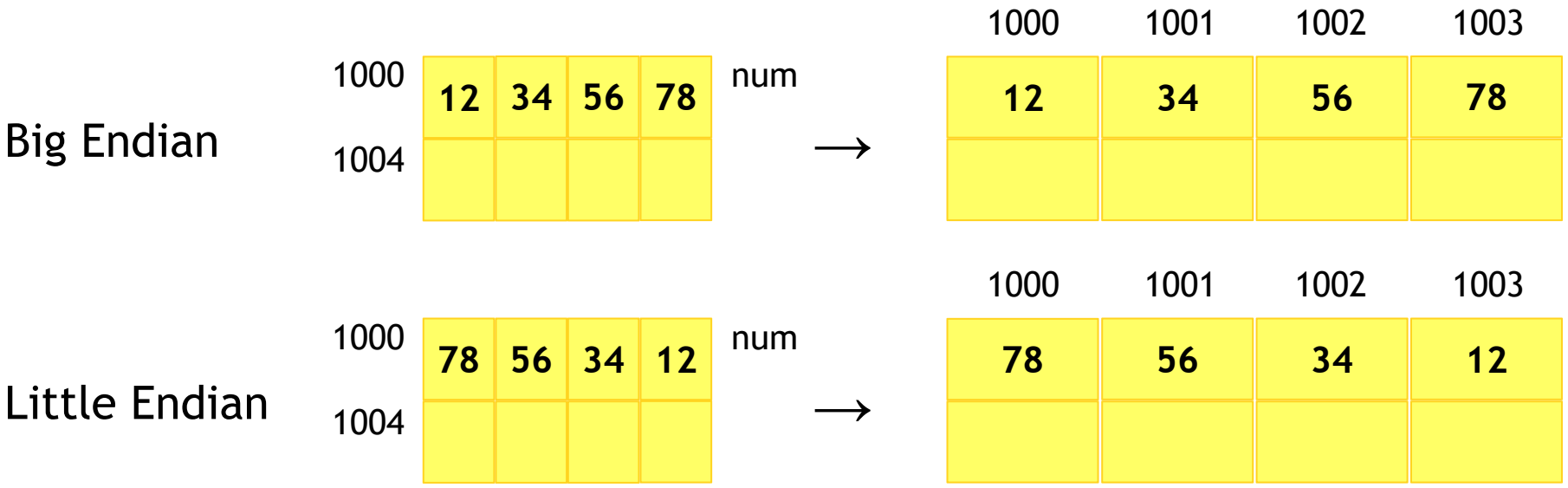
Embedded C

Pointers

Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    return 0;
}
```

- Let us consider the following example and how it would be stored in both machine types



Embedded C

Pointers

- So as a summary the type to the pointer does not say its type, but the type of the data its pointing to
- So the size of the pointer for different types remains the same

Example

```
#include <stdio.h>

int main()
{
    if (sizeof(char *) == sizeof(long long *))
    {
        printf("Yes its Equal\n");
    }

    return 0;
}
```

Embedded C

Pointers

- Rule: “Pointer value of NULL or Zero = Null Addr = Null Pointer = Pointing to Nothing”

Embedded C

Pointers

- Is it pointing to the valid address?
- If yes can we read or write in the location where its pointing?
- If no what will happen if we access that location?
- So in summary where should we point to avoid all this questions if we don't have a valid address yet?
- The answer is Point to Nothing!!

Embedded C

Pointers

- Now what is Point to Nothing?
- A permitted location in the system will always give predictable result!
- It is possible that we are pointing to some memory location within our program limit, which might fail any time! Thus making it bit difficult to debug.
- An act of initializing pointers to 0 (generally, implementation dependent) at definition.
- 0??, Is it a value zero? So a pointer contain a value 0?
- Yes. On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system

Embedded C

Pointers

- So by convention if a pointer is initialized to zero value, it is logically understood to be point to nothing.
- And now, in the pointer context, 0 is called as **NULL**
- So a pointer that is assigned NULL is called a Null Pointer which is **Pointing to Nothing**
- So dereferencing a NULL pointer is illegal and will always lead to segment violation, which is better than pointing to some unknown location and failing randomly!

Will meet again