C

Come let's see how deep it is!!
Day 1

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
Brief History

- Prior to C, most of the computer languages (such as Algol)
  - Academic oriented, unrealistic and were generally defined by committees.
  - Designed having application domain in mind (Non portable)
- It has lineage starting from CPL
  - Martin Richards implemented BCPL
  - Ken Thompson further refined BCPL to a language named as B
  - Dennis M. Ritchie added types to B and created a language C
- With just 32 keywords, C established itself in a very wide base of applications.

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

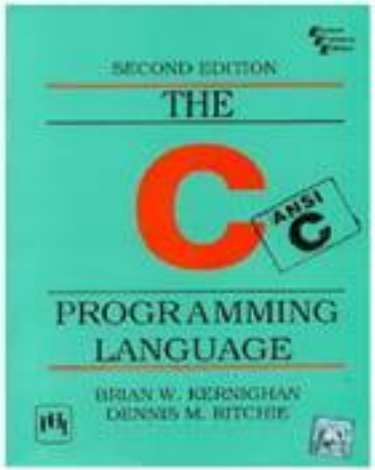
- Considered as a middle level language
- Can be considered as a pragmatic language.
- It is indented to be used by advanced programmers, for serious use, and not for novices and thus qualify less as an academic language for learning
- Gives importance to curt code.
- It is widely available in various platforms from mainframes to palmtops and is known for its wide availability

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
Standard



- "The C programming language" book served as a primary reference for C programmers and implementers alike for nearly a decade
- However it didn't define C perfectly and there were many ambiguous parts in the language
- As far as the library was concerned, only the C implementation in UNIX was close to the 'standard'

- So many dialects existed for C and it was the time the language has to be standardized and it was done in 1989 with ANSI C standard
- Nearly after a decade another standard, C9X, for C is available that provides many significant improvements over the previous 1989 ANSI C standard

C
Keywords

- In programming, a keyword is a word that is reserved by a program because the word has a special meaning
- Keywords can be commands or parameters
- Every programming language has a set of keywords that cannot be used as variable names
- Keywords are sometimes called reserved names

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

- 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

C
Compilation

- Assuming your code is ready, use the following commands to compile the code
- On command prompt, type

  $ gcc <file_name>.c
- This will generate a executable named a.out
- But it is recommended that you follow proper conversion even while generating your code, so you could use

  $ gcc <file_name>.c -o <file_name>
- This will generate a executable named <file_name>

C
Execution

- To execute your code you shall try

  ```
  $ ./a.out
  ```

- If you have named you output file as your <file_name> then

  ```
  $ ./<file_name>
  ```

- This should the expected result on your system

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
- "Binary Digit" 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 - Character

- One byte represents one unique character like 'A', 'b', '1', '$' ...
- Its possible to have 256 different combinations of 0s and 1s to form a individual character
- There are different types of character code representation like
  - ASCII → American Standard Code for Information Interchange - 7 Bits (Extended - 8 Bits)
  - EBCDIC → Extended BCD Interchange Code - 8 Bits
  - Unicode → Universal Code - 16 Bits and more

Embedded C

Data Representation - Character

- ASCII is the oldest representation
- Please try the following on command prompt to know the available codes

  $ man ascii

- Can be represented by char datatype

Value	No of Bits
0	0 0 1 1 0 0 0 0
A	0 1 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 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
No of Bits / 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 : $-k \equiv 2^n - k$

Embedded C

Float Point Number

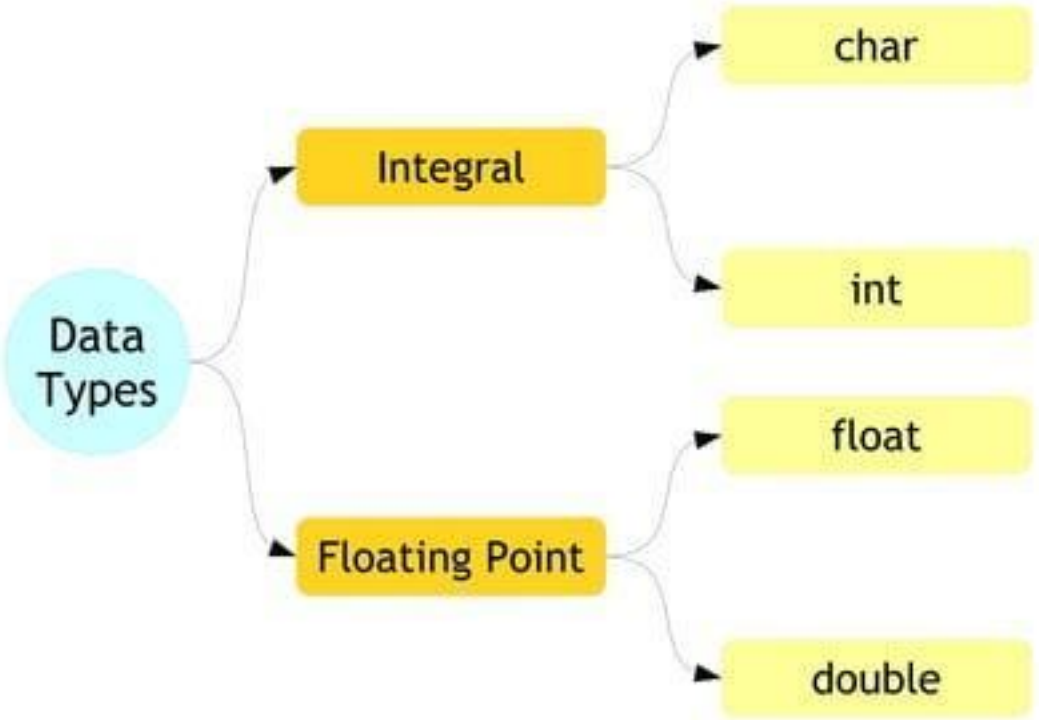
- A formulaic representation which approximates a real number
- Computers are integer machines and are capable of representing real numbers only by using complex codes
- The most popular code for representing real numbers is called the IEEE Floating-Point Standard

	Sign	Exponent	Mantissa
Float (32 bits) Single Precision	1 bit	8 bits	23 bits
Double (64 bits) Double Precision	1 bit	11 bits	52 bits

Basic Data Types

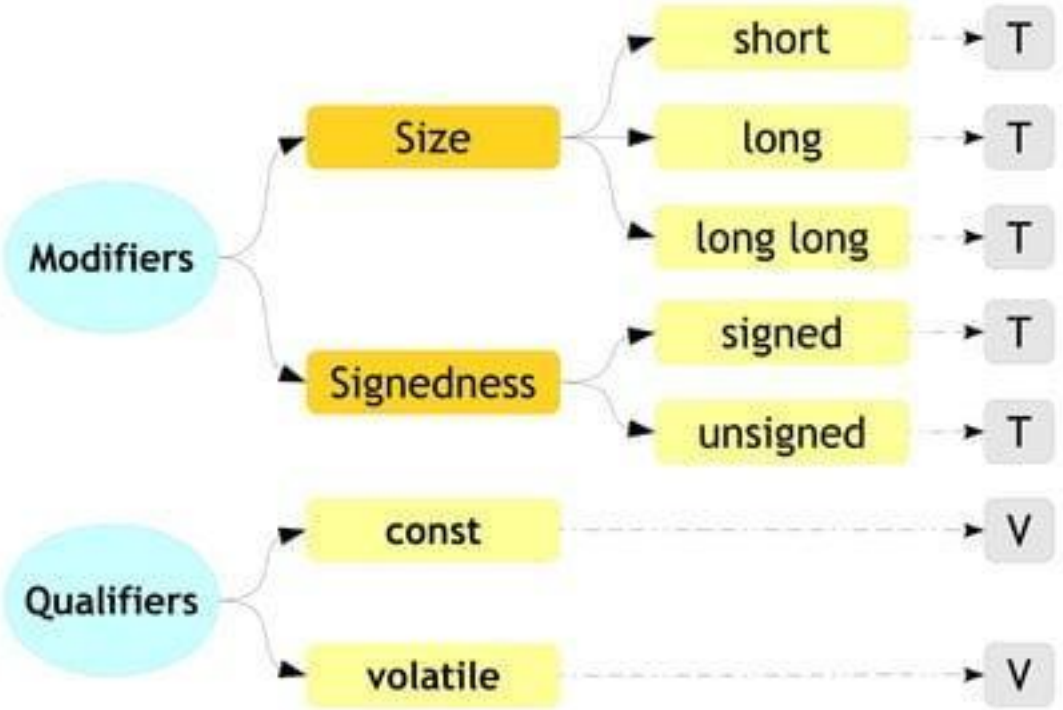
Embedded C

Basic Data Types



Embedded C

Data Type Modifiers and Qualifiers

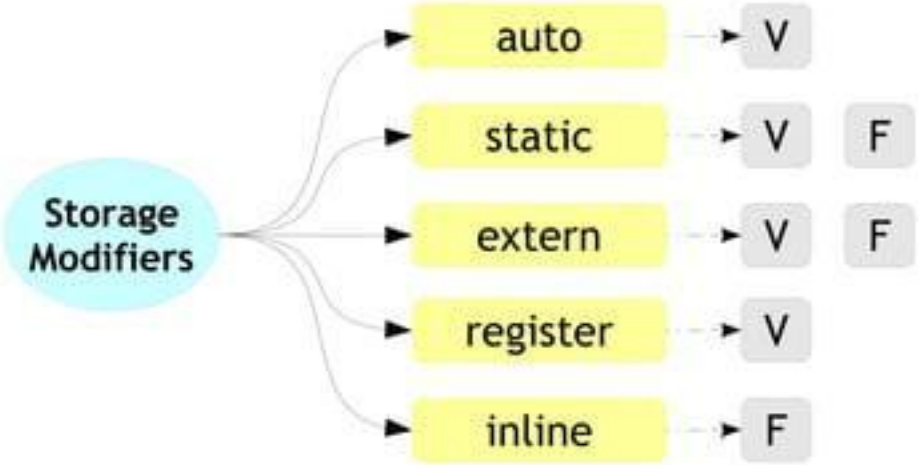


Notes:
- ANSI says, ensure that: **char ≤ short ≤ int ≤ long**
- unsigned float in not supported

V	Variables
T	Data Types
F	Functions

Embedded C

Data Type and Function storage modification



V	Variables
T	Data Types
F	Functions

Conditional Constructs

Embedded C

Code Statements - Simple

```
int main()
{
    number = 5;
    3; +5;
    sum = number + 5;
    4 + 5;
    ;
}
```

Assignment statement

Valid statement, But smart compilers might remove it

Assignment statement. Result of the number + 5 will be assigned to sum

Valid statement, But smart compilers might remove it

This valid too!!

Embedded C

Code Statements - Compound

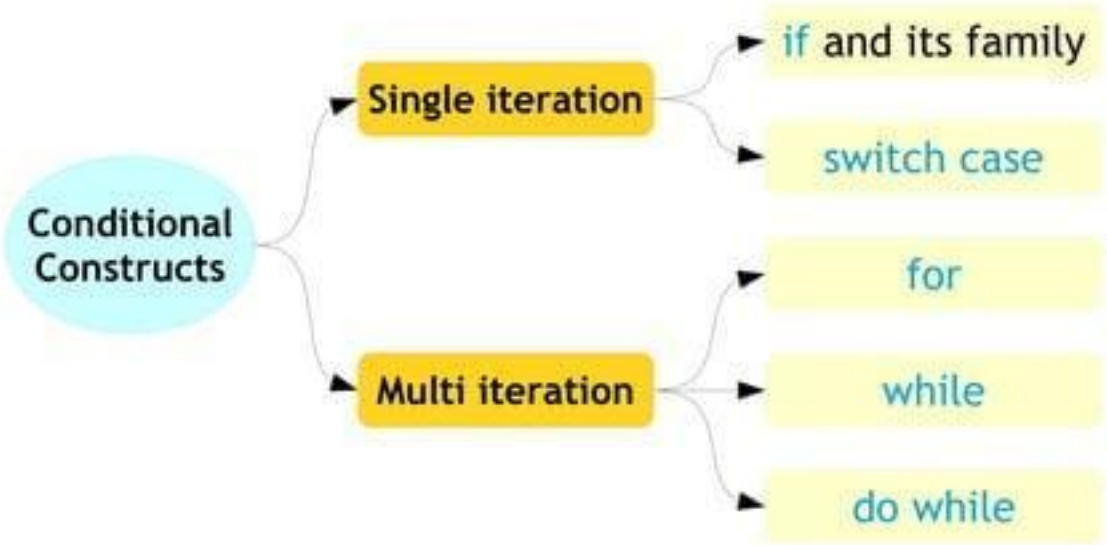
```
int main()
{
    ...
    if (num1 > num2)
    {
        if (num1 > num3)
        {
            printf("Hello");
        }
        else
        {
            printf("World");
        }
    }
    ...
}
```

If conditional statement

Nested if statement

Embedded C

Conditional Constructs



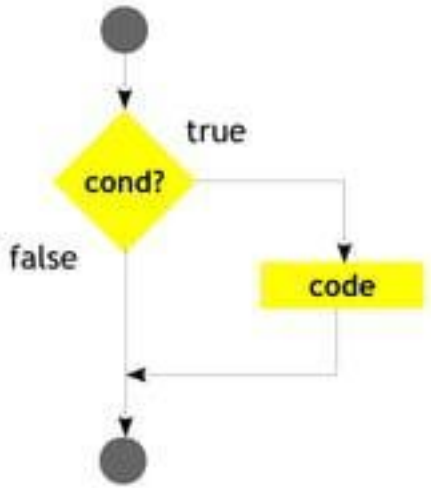
Embedded C

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;
}
```

ΣMERTXE

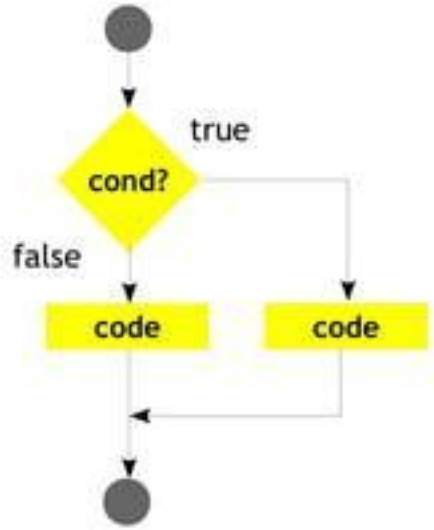
Embedded C

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;
}
```

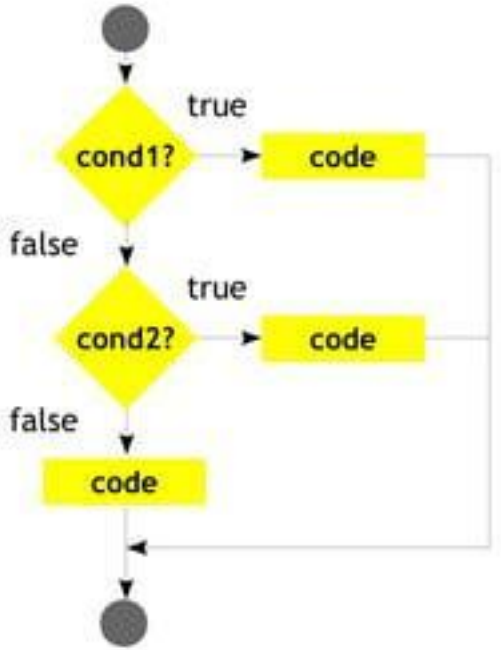
Embedded C

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;
}
```

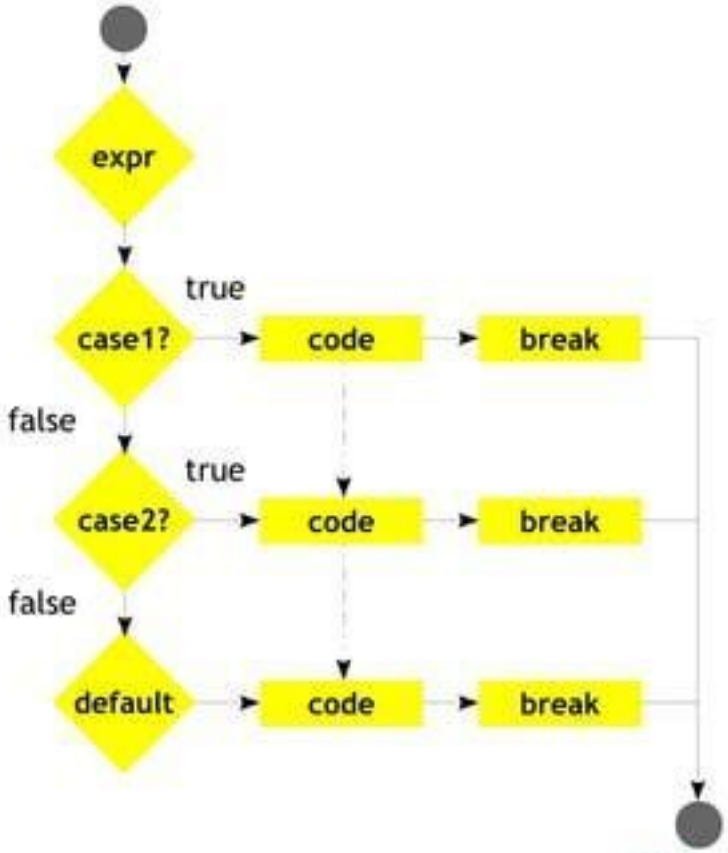
Embedded C

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

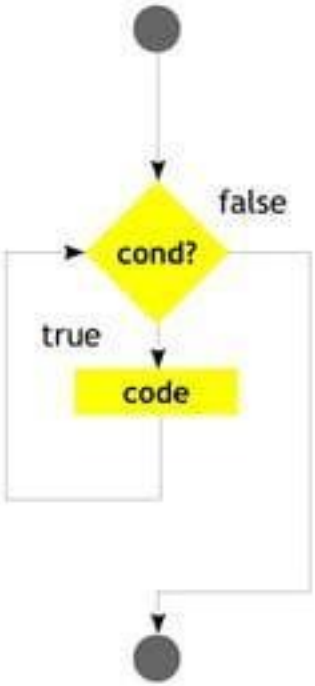
```
while (condition)
{
    statement(s);
}
```

- **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

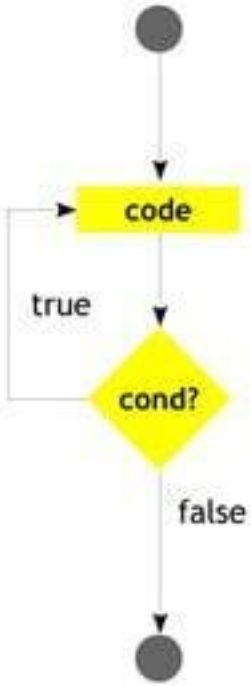
Conditional Constructs - do while

Syntax

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

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

Flow



Example

```
#include <stdio.h>

int main()
{
    int i;

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

    return 0;
}
```

Embedded C

Conditional Constructs - for

Syntax

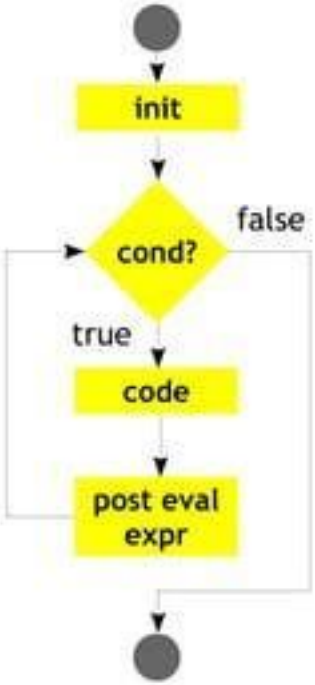
```
for (init; condition; post evaluation expr)
{
    statement(s);
}
```

- **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

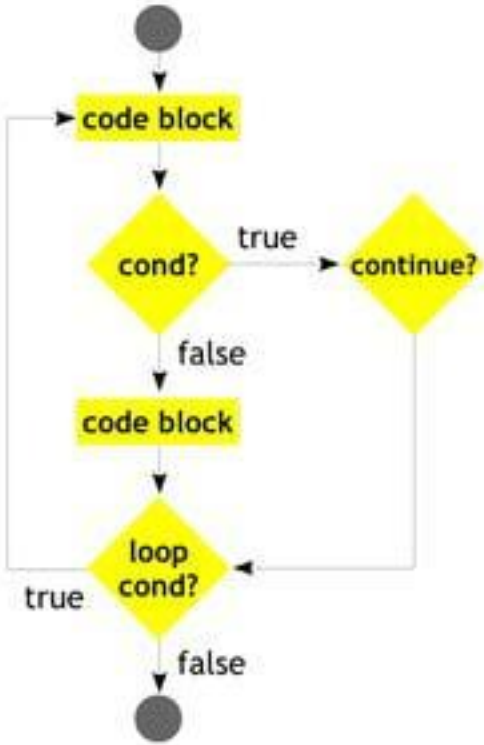
Conditional Constructs - continue

- 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

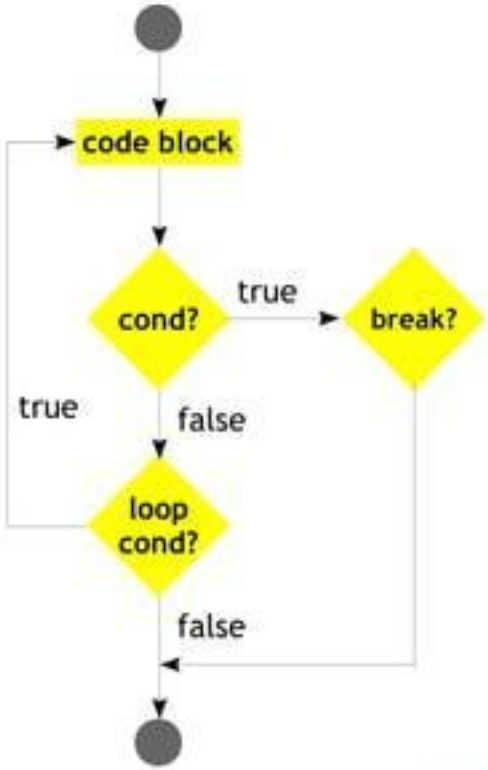
Conditional Constructs - break

- 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

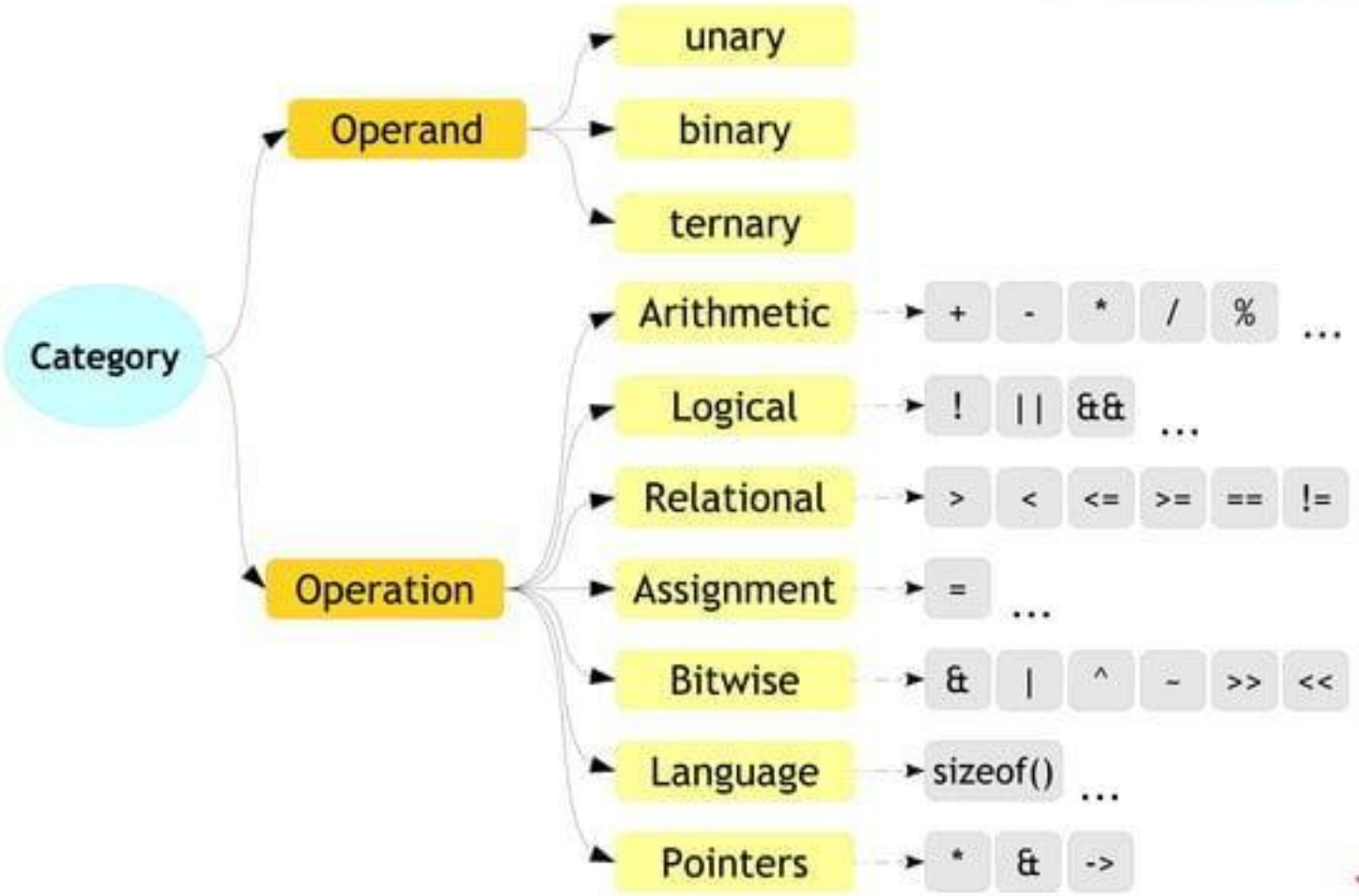
Embedded C

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



Embedded C

Operators - Precedence and Associativity

Operators	Associativity	Precedence
() [] -> .	L - R	HIGH
! ~ ++ -- - + * & (type) sizeof	R - L	
/ % *	L - R	
+ -	L - R	
<< >>	L - R	
< <= > >=	L - R	
== !=	L - R	
&	L - R	
^	L - R	
|	L - R	
&&	L - R	
||	L - R	
?:	R - L	
= += -= *= /= %= &= ^= |= <<= >>=	R - L	
,	L - R	LOW

Note:

post ++ and -- operators have higher precedence than pre ++ and -- operators

(Rel-99 spec)

Embedded C

Operators - Arithmetic

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

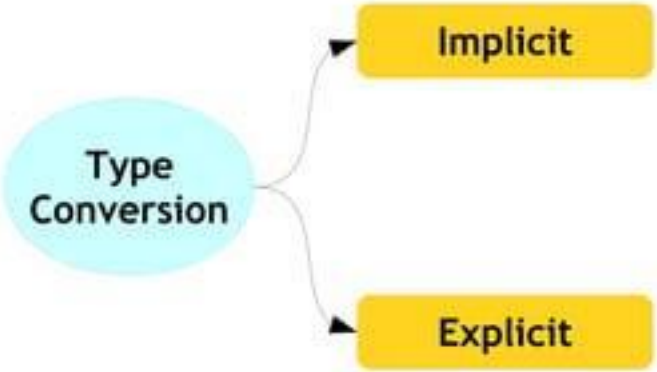
Example

```c
#include <stdio.h>

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

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

    return 0;
}
```

What will be the output?

Embedded C

Type Conversion



Embedded C

Type Conversion Hierarchy

Embedded C

Type Conversion - Implicit

- Automatic Unary conversions
  - The result of + and - are promoted to int if operands are char and short
  - The result of ~ and ! is integer
- Automatic Binary conversions
  - If one operand is of LOWER RANK (LR) data type & other is of HIGHER RANK (HR) data type then LOWER RANK will be converted to HIGHER RANK while evaluating the expression.
  - Example: LR + HR → LR converted to HR

Embedded C

Type Conversion - Implicit

- Type promotion
  - LHS type is HR and RHS type is LR → int = char → LR is promoted to HR while assigning
- Type demotion
  - LHS is LR and RHS is HR → int = float → HR rank will be demoted to LR. Truncated

Embedded C

Type Conversion - Explicit (Type Casting)

Syntax

```
(data type) expression
```

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 5, num2 = 3;

    float num3 = (float) num1 / num2;

    printf("num3 is %f\n", num3);

    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

```c
#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

```c
#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

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

>>	**Right Shift**	Shift all the bits right n times by introducing zeros left

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

<<	**Left Shift**	Shift all the bits left n times by introducing zeros right

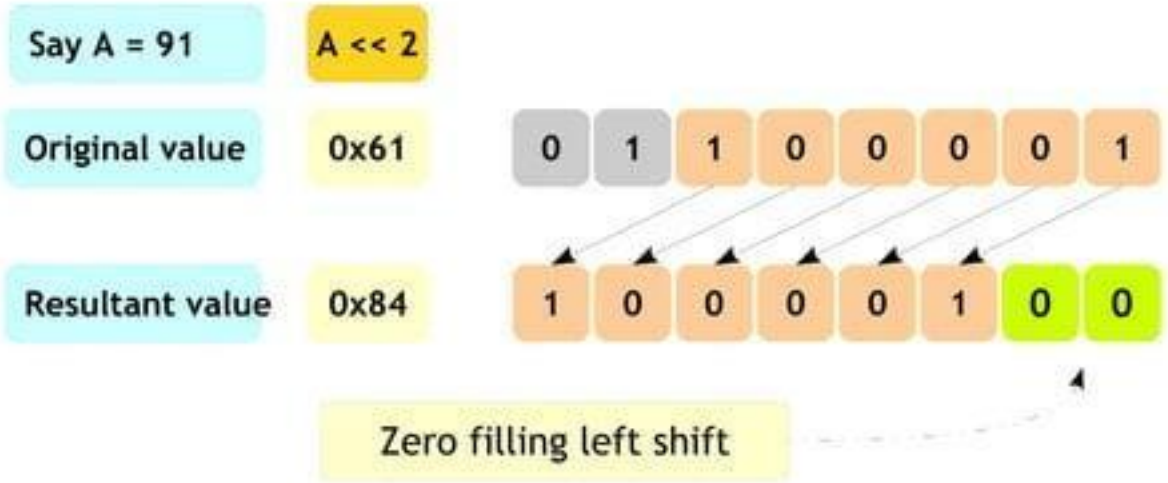
Operand	Value	
A	0x61	0 1 1 0 0 0 0 1
A << 2	0x84	1 0 0 0 0 1 0 0

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

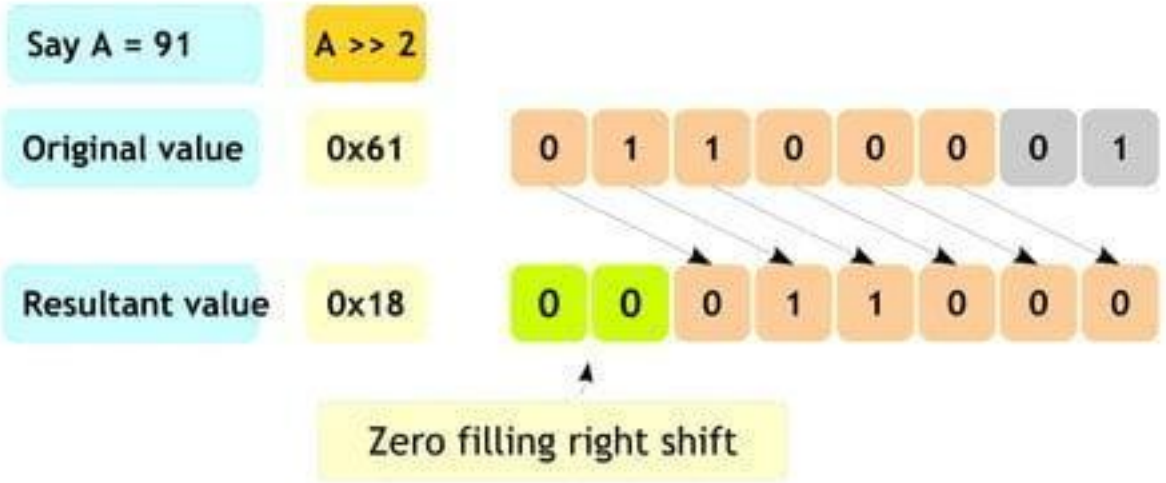


Embedded C

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

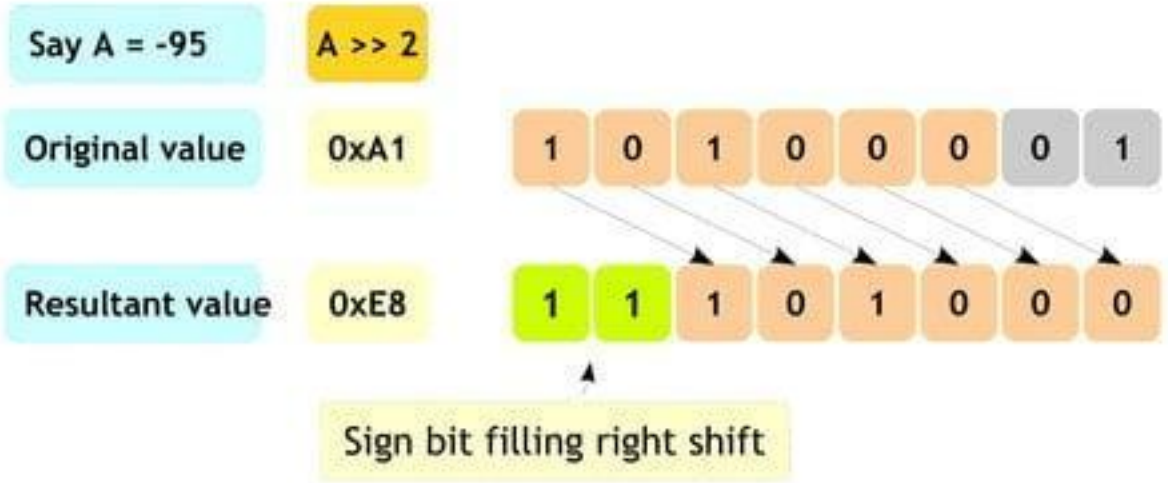


Embedded C

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 - Bitwise

Example

```c
#include <stdio.h>

int main()
{
    int count;
    unsigned char num = 0xFF;

    for (count = 0; num != 0; num >>= 1)
    {
        if (num & 01)
        {
            count++;
        }
    }

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

    return 0;
}
```

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 - Language - sizeof()

- 3 reasons for why sizeof is not a function
  - Any type of operands,
  - Type as an operand,
  - No brackets needed across operands

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;
}
```

EMERTXE

Embedded C

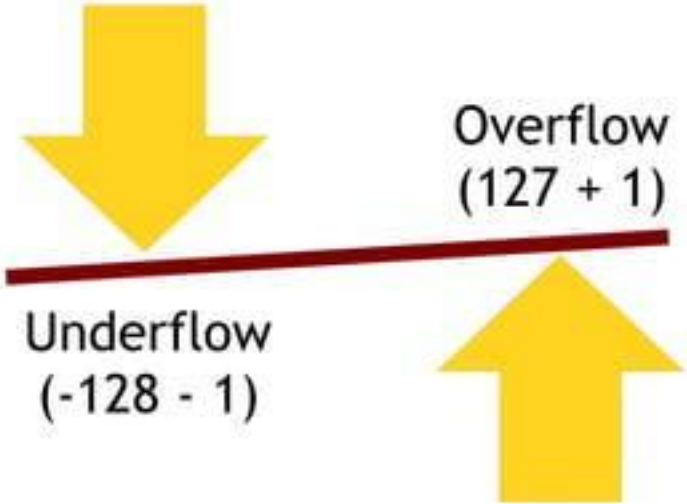
Operators - Comma

- The left operand of a comma operator is evaluated as a void expression: Then the right operand is evaluated, the result has its type and value
- Comma acts as separator (not an operator) in following cases
  - Arguments to functions
  - Lists of initializers (variable declarations)
- But, can be used with parentheses as function arguments such as -
  - `foo ((x = 2, x + 3)); // final value of argument is 5`

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

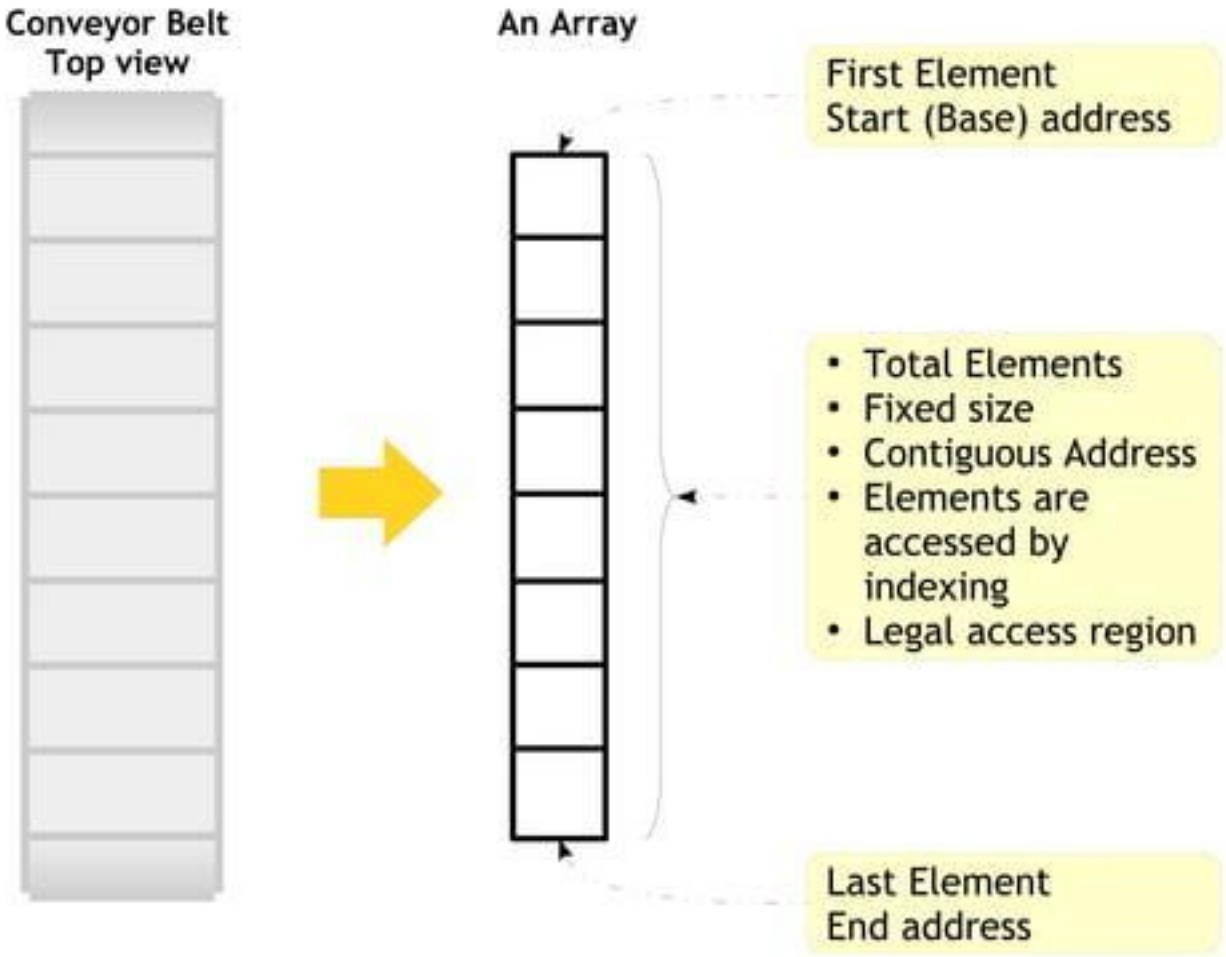
Array

Embedded C

Arrays - Know the Concept

Embedded C

Arrays - Know the Concept



Embedded C

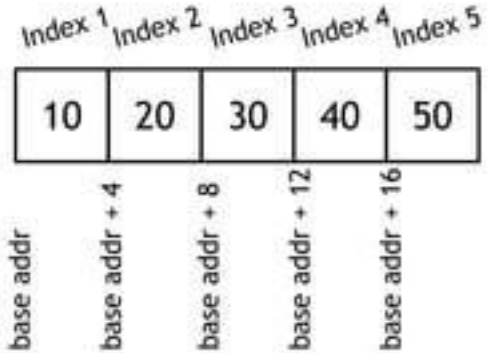
Arrays

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};
```



EMERTXE

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;
}
```

Embedded C

Arrays - Copying

- No!! its not so simple to copy two arrays as put in the previous slide. C doesn't support it!
- Then how to copy an array?
- It has to be copied element by element

Embedded C

Arrays - Oops!! what is this now?

Pointers

Embedded C

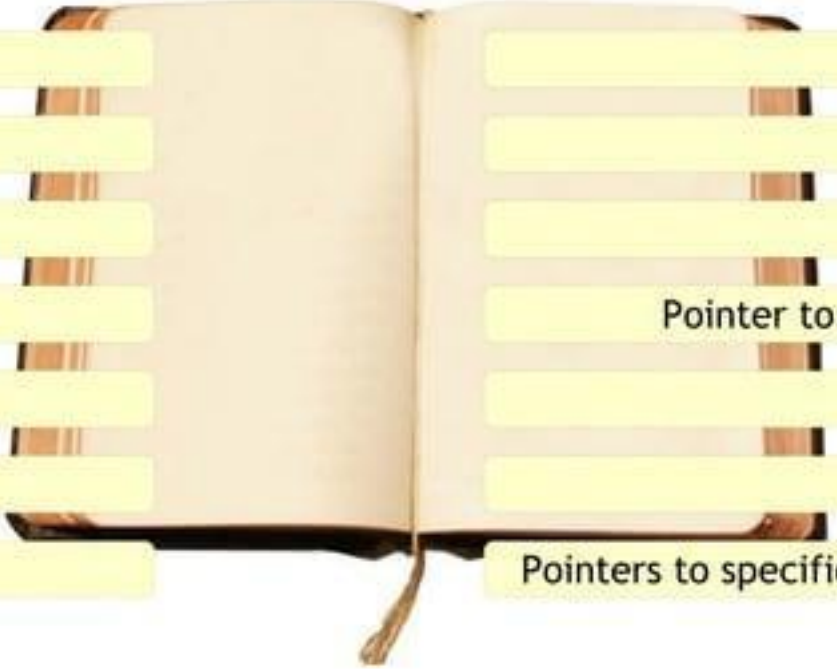
Pointers - Jargon

- What's a Jargon?
  - Jargon may refer to terminology used in a certain profession, such as computer jargon, or it may refer to any nonsensical language that is not understood by most people.
  - Speech or writing having unusual or pretentious vocabulary, convoluted phrasing, and vague meaning.
- Pointer are perceived difficult
  - Because of jargonification
- So, let's dejargonify & understand them

Embedded C

Pointers - Analogy with Book

A Book	System Memory
Contains pages	Memory Pages
Front matter	Boot Code
Table of Contents	Pointer to different programs
Chapters	Code Segments
Notes Section	Data & Stack
Indexes	Pointers to specific memory locations



Embedded C

Pointers - Computers

- Just like a book analogy, Computers contains different different sections (Code) in the memory
- All sections have different purposes
- Every section has a address and we need to point to them whenever required
- In fact everything (Instructions and Data) in a particular section has a address!!
- So the pointer concept plays a big role here

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
- To have the dynamic allocation mechanism

Embedded C

Pointers - The 7 Rules

- Rule 1 - Pointer is an Integer
- Rule 2 - Referencing and De-referencing
- Rule 3 - Pointer means Containing
- Rule 4 - Pointer Type
- Rule 5 - Pointer Arithmetic
- Rule 6 - Pointing to Nothing
- Rule 7 - Static vs Dynamic Allocation

Embedded C

Pointers - The 7 Rules - Rule 1

Integer i;
Pointer p;
Say:
    i = 6;
    p = 6;

Embedded C

Pointers - The 7 Rules - Rule 1

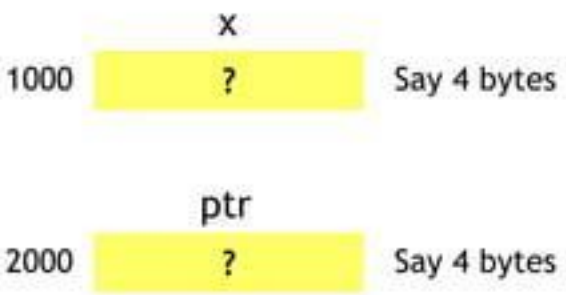
- Whatever we put in data bus is Integer
- Whatever we put in address bus is Pointer
- So, at concept level both are just numbers. May be of different sized buses
- Rule: "Pointer is an Integer"
- Exceptions:
  - May not be address and data bus of same size
  - Rule 2 (Will see why? while discussing it)

Embedded C

Pointers - Rule 1 in detail

Example

```c
#include <stdio.h>

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

    x = 5;
    ptr = 5;

    return 0;
}
```

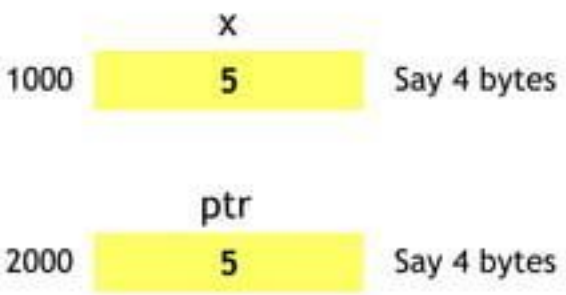


Embedded C

Pointers - Rule 1 in detail

Example

```c
#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”

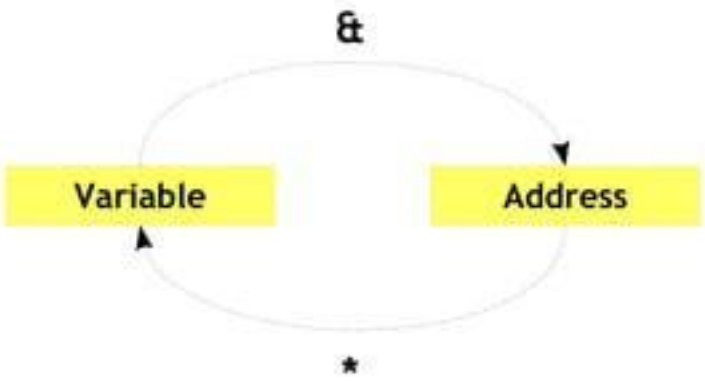
  32 bit system = 4 Bytes

  64 bit system = 8 Bytes

Embedded C

Pointers - The 7 Rules - Rule 2

- Rule : "Referencing and Dereferencing"

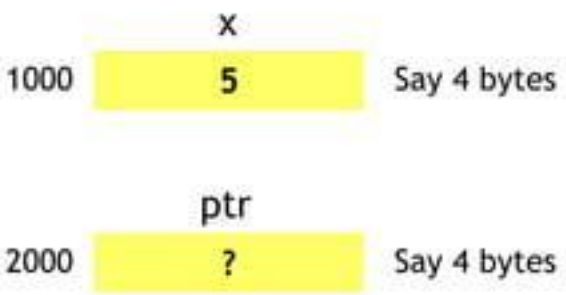


Embedded C

Pointers - Rule 2 in detail

Example

```c
#include <stdio.h>

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

    x = 5;

    return 0;
}
```



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

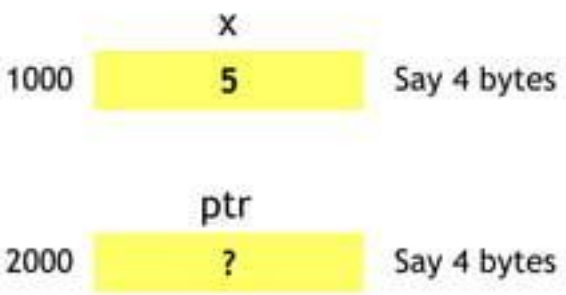
  * 1000

Embedded C

Pointers - Rule 2 in detail

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

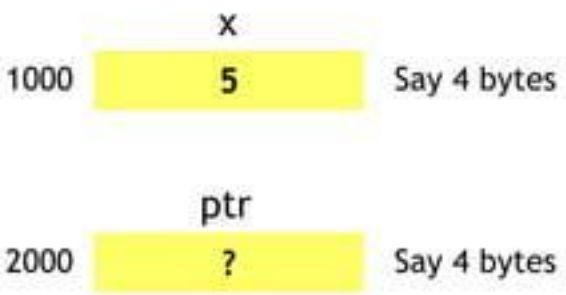
  * 1000 → 5

Embedded C

Pointers - Rule 2 in detail

Example

```c
#include <stdio.h>

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

    x = 5;
    ptr = &x;

    return 0;
}
```



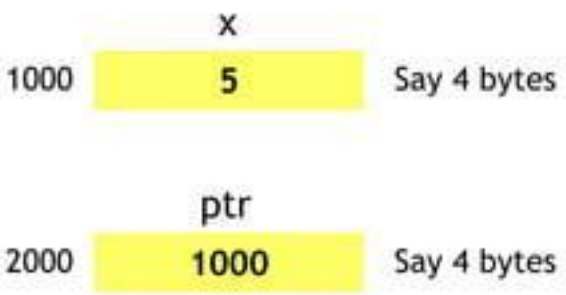
- What should be the change in the above diagram for the above code?

Embedded C

Pointers - Rule 2 in detail

Example

```c
#include <stdio.h>

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

    x = 5;
    ptr = &x;

    return 0;
}
```



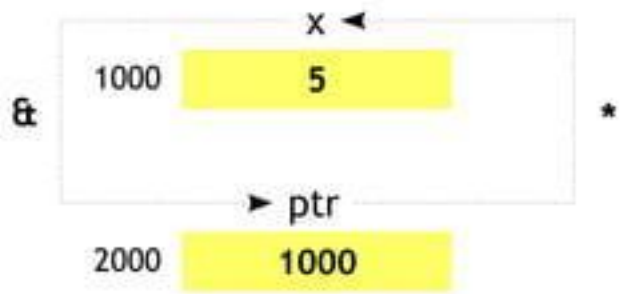
- So pointer should contain the address of a variable
- It should be a valid address

Embedded C

Pointers - Rule 2 in detail

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 - Rule 2 in detail

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 - Rule 2 in detail

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 - Rule 2 in detail

Example

```c
#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"

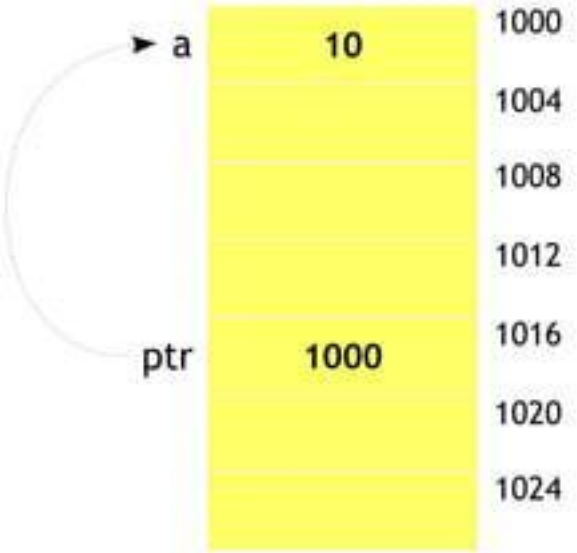
Embedded C

Pointers - The 7 Rules - Rule 3

- 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 - The 7 Rules - Rule 4

- Types to the pointers
- What??, why do we need types attached to pointers?

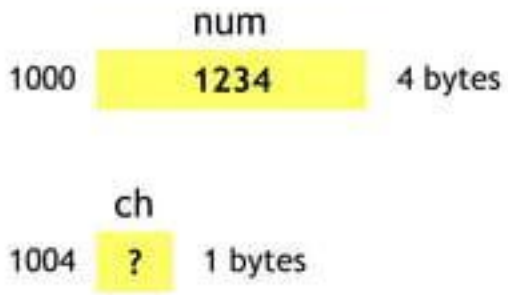
Embedded C

Pointers - Rule 4 in detail

- The question is, does address has a type?

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    return 0;
}
```

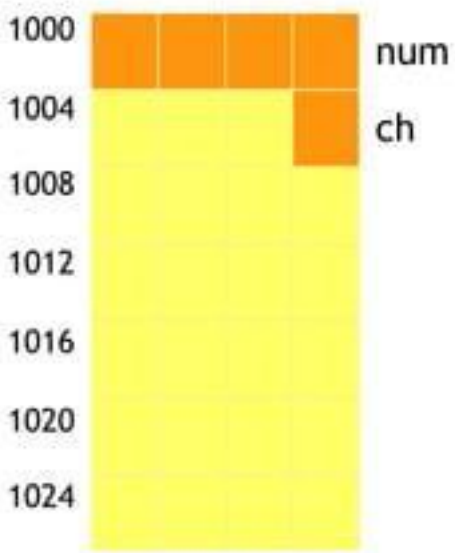


- So from the above above diagram can we say &num → 4 bytes and &ch → 1 byte?

Embedded C

Pointers - Rule 4 in detail

- The answer is no!!, it does not depend on the type of the variable
- The size of address remains the same, and it depends on the system we use
- Then a simple questions arises is why types to pointers?

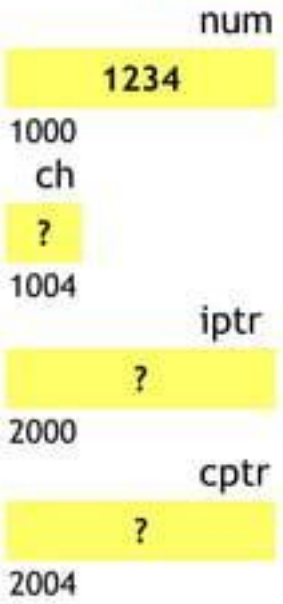


Embedded C

Pointers - Rule 4 in detail

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr;
    char *cptr;

    return 0;
}
```



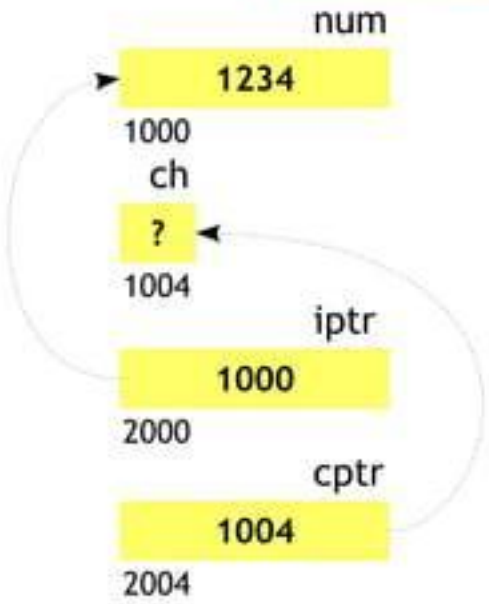
- Lets consider the above examples to understand it
- Say we have a integer and a character pointer

Embedded C

Pointers - Rule 4 in detail

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr = &num;
    char *cptr = &ch;

    return 0;
}
```

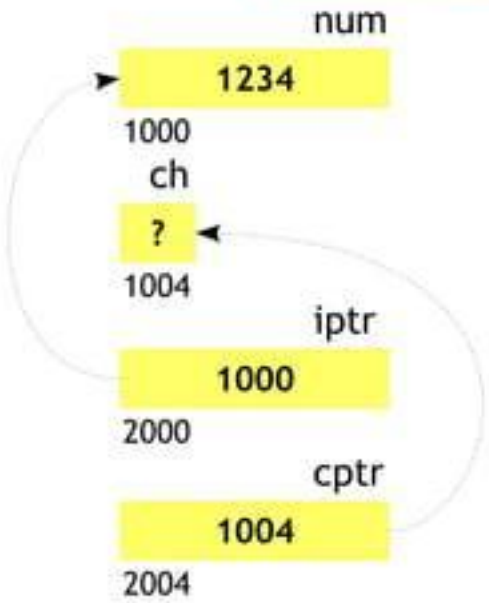


- Lets consider the above examples to understand it
- Say we have a integer and a character pointer

Embedded C

Pointers - Rule 4 in detail

- With just the address, can know what data is stored?
- How would we know how much data to fetch for the address it is pointing to?
- Eventually the answer would be NO!!
- So the type of the pointer is required while
  - Dereferencing it
  - Doing pointer arithmetic



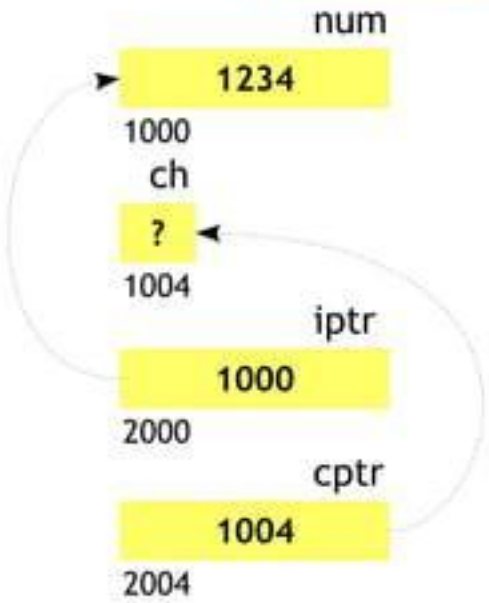
Embedded C

Pointers - Rule 4 in detail

- When we say while dereferencing, how does the pointer know how much data it should fetch at a time
- From the diagram right side we can say

  *cptr fetches a single byte

  *iptr fetches 4 consecutive bytes
- So as conclusion we can say



type * → fetch sizeof(type) bytes

Embedded C

Pointers - Rule 4 in detail - Endianness

- 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 - Rule 4 in detail - Endianness

- LSB
  - The byte of a multi byte number with the least importance
  - The change in it would have least effect on complete number
- MSB
  - The byte of a multi byte number with the most importance
  - The change in it would have more effect on complete change number

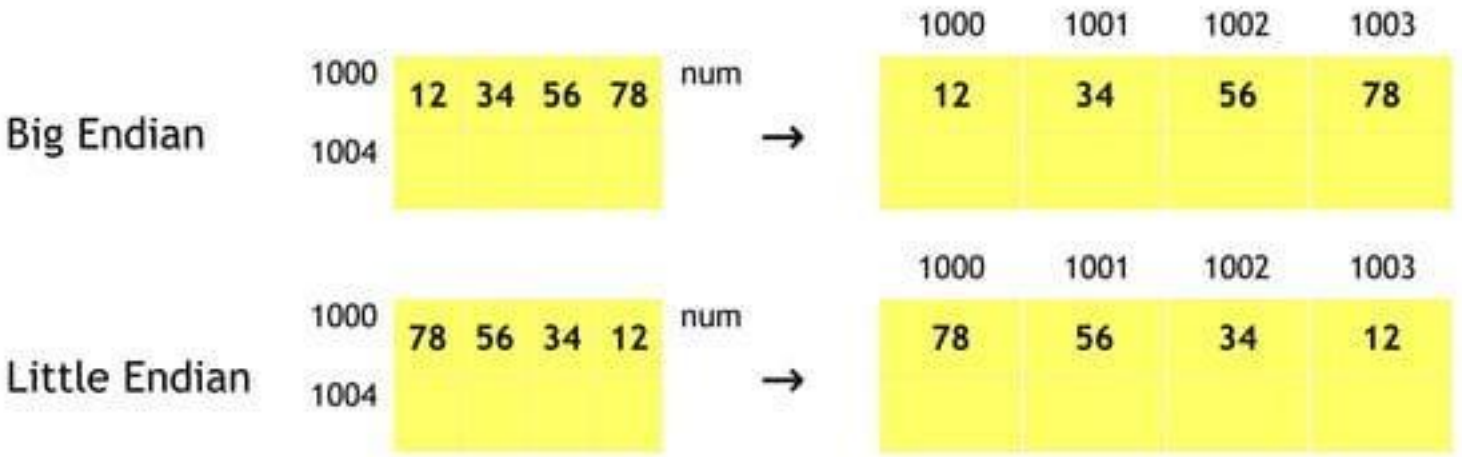
Embedded C

Pointers - Rule 4 in detail - Endianness

Example

```c
#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 - Rule 4 in detail - Endianness

- OK Fine. What now? How is it going affect to fetch and modification?
- Let us consider the same example put in the previous slide

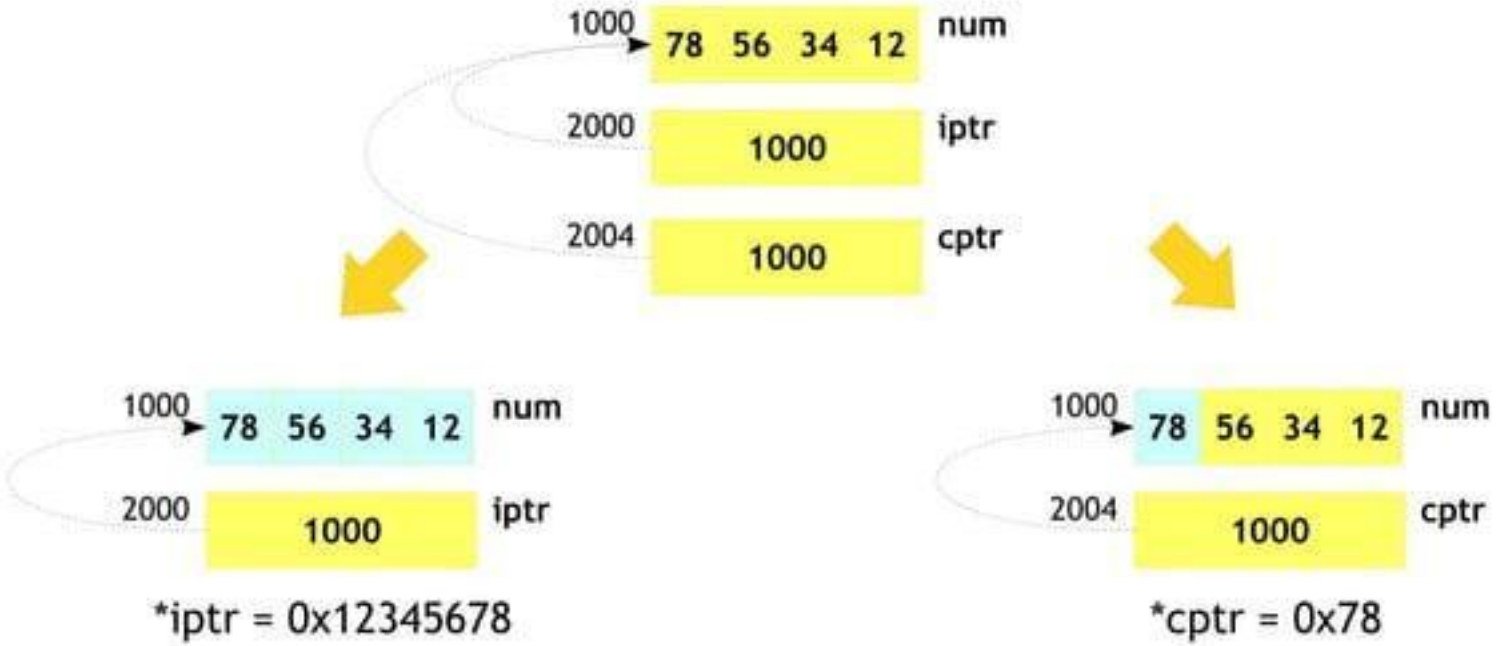
Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;
    int *iptr, char *cptr;

    iptr = &num;
    cptr = &num;

    return 0;
}
```

- First of all is it possible to access a integer with character pointer?
- If yes, what should be the effect on access?
- Let us assume a Litte Endian system

Embedded C

Pointers - Rule 4 in detail - Endianness



- So from the above diagram it should be clear that when we do cross type accessing, the endianness should be considered

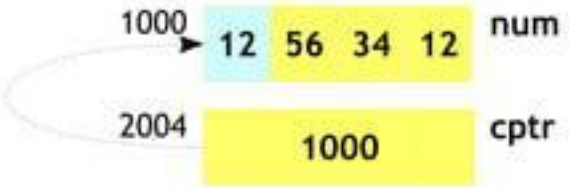
Embedded C

Pointers - The 7 Rules - Rule 4

Example

```c
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    int *iptr = &num;
    char *cptr = &num;

    *cptr = 0x12;

    return 0;
}
```

- So changing *cptr will change only the byte its pointing to



*cptr = 0x12

- So *iptr would contain 0x12345612 now!!

Embedded C

Pointers - The 7 Rules - Rule 4

- 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

```c
#include <stdio.h>

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

    return 0;
}
```

Embedded C

Pointers - The 7 Rules - Rule 4

- Summarized as the following rule:

  Rule: "Pointer of type t = t Pointer = (t *) = A variable, which contains an address, which when dereferenced returns a variable of type t, starting from that address"

Embedded C

Pointers - The 7 Rules - Rule 5

- Pointer Arithmetic

  Rule: “Value(p + i) = Value(p) + i * sizeof(*p)”

Embedded C

Pointers - The Rule 5 in detail

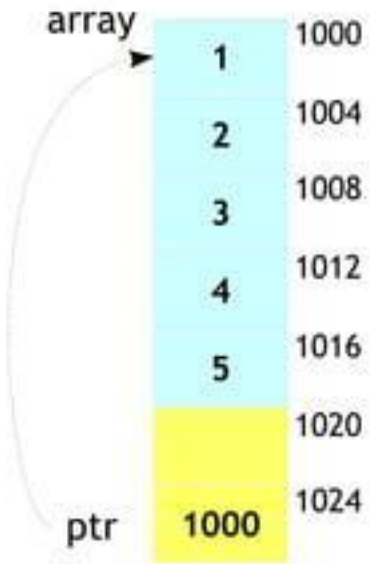
- Before proceeding further let us understand an array interpretation
  - Original Big Variable
  - Constant Pointer to the 1st Small Variable in the Big Variable
  - When first interpretation fails than second interpretation comes to picture.
  - The following are the case when first interpretation fails:
    - When we pass array variable as function argument
    - When we assign a array variable to pointer variable

Embedded C

Pointers - The Rule 5 in detail

Example

```c
#include <stdio.h>

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

    return 0;
}
```



- So,

  Address of array = 1000

  Base address = 1000

  &array[0] = 1 → 1000

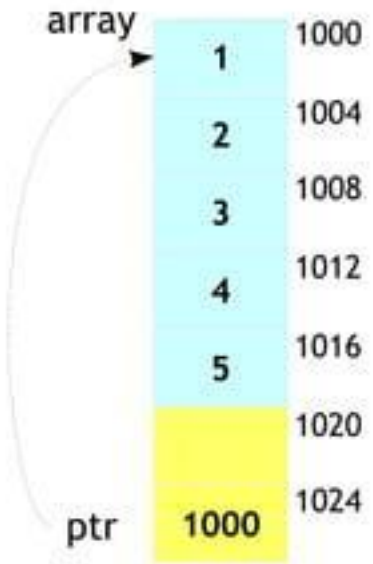
  &array[1] = 2 → 1004

Embedded C

Pointers - The Rule 5 in detail

Example

```c
#include <stdio.h>

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

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

    return 0;
}
```



- This code should print 1 as output since its points to the base address
- Now, what should happen if we do

  ptr = ptr + 1;

Embedded C

Pointers - The Rule 5 in detail

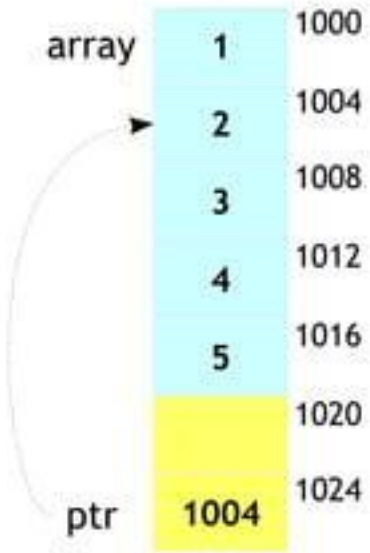
- ptr = ptr + 1;
- The above line can be discribed as follows
- ptr = ptr + 1 * sizeof(data type)
- In this example we have a integer array, so
- ptr = ptr + 1 * sizeof(int)

  = ptr + 1 * 4

  = ptr + 4
- Here ptr = 1000 so

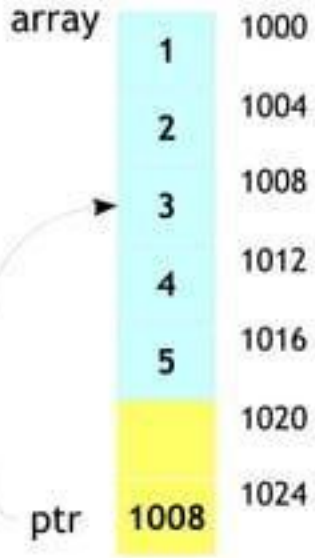
  = 1000 + 4

  = 1004

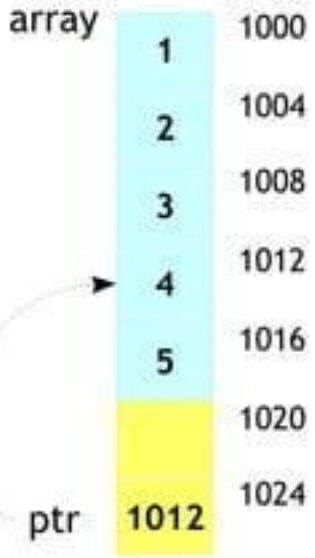


Embedded C

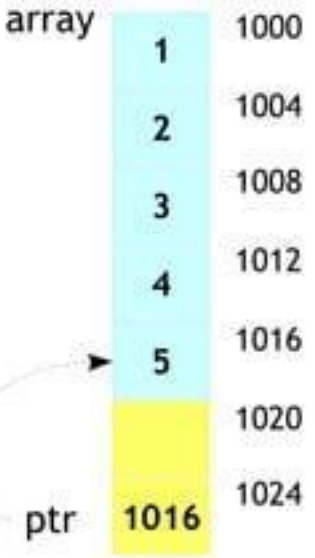
Pointers - The Rule 5 in detail



ptr = ptr + 2;



ptr = ptr + 3;

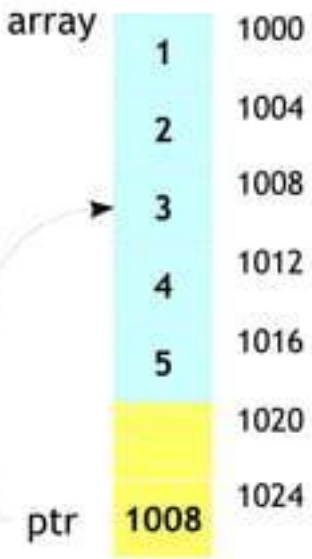


ptr = ptr + 4;

- Why does the compiler does this?. Just for convenience

Embedded C

Pointers - The Rule 5 in detail



ptr = ptr + 2;

- Relation with array can be explained as

  ptr + 2

  ptr + 2 * sizeof(int)

  1000 + 2 * 4

  1008 → &array[2]

- So,

  ptr + 2 → 1008 → &array[2]

  *(ptr + 2) → *(1008) → array[2]

Embedded C

Pointers - The Rule 5 in detail

- So to access a array element using a pointer would be

*(ptr + i) → array[i]

- This can be written as following too!!

array[i] → *(array + i)

- Which results to

ptr = array

- So as summary the below line also becomes valid because of second array interpretation

int *ptr = array;

Embedded C

Pointers - The Rule 5 in detail

- Wait can I write

*(ptr + i) → *(i + ptr)

- Yes. So than can I write

array[i] → i[array]

- Yes. You can index the element in both the ways

Embedded C

Pointers - The 7 Rules - Rule 5 - Size of void

- On gcc size of void is 1
- Hence pointer arithmetic can be performed on void pointer
- Its compiler dependent!

Note: To make standard compliant, compile using gcc -pedantic-errors

Embedded C

Pointers - The 7 Rules - Rule 6

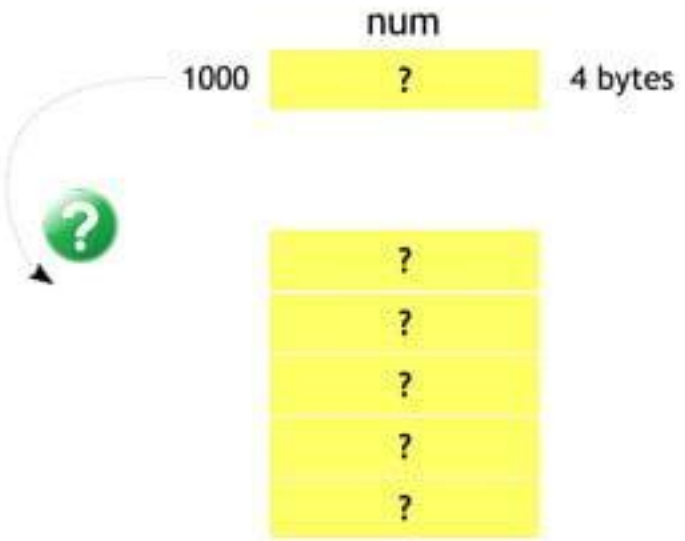
- Rule: "Pointer value of NULL or Zero = Null Addr = Null Pointer = Pointing to Nothing"

Embedded C

Pointers - Rule 6 in detail - NULL Pointer

Example

```c
#include <stdio.h>

int main()
{
    int *num;

    return 0;
}
```



Embedded C

Pointers - Rule 6 in detail - NULL Pointer

- 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 - Rule 6 in detail - NULL Pointer

- 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 - Rule 6 in detail - NULL Pointer

- 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!

Embedded C

Pointers - Rule 6 in detail - NULL Pointer

- Need for Pointing to 'Nothing'
  - Terminating Linked Lists
  - Indicating Failure by malloc, ...
- Solution
  - Need to reserve one valid value
  - Which valid value could be most useless?
  - In wake of OSes sitting from the start of memory, 0 is a good choice
  - As discussed in previous sides it is implementation dependent

Embedded C

Pointers - Rule 6 in detail - NULL Pointer

Example

```c
#include <stdio.h>

int main()
{
    int *num;

    num = NULL;

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    int *num = NULL;

    return 0;
}
```

Embedded C

Pointers - The 7 Rules - Rule 7

- Rule: "Static Allocation vs Dynamic Allocation"

Example

```c
#include <stdio.h>

int main()
{
    int num1;
    static int num2;
    char *ptr;
    char array[5];

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```

Embedded C

Pointers - Rule 7 in detail

- Unnamed vs named Allocation = Unnamed/named Houses

Ok, House 1, I should go??? Oops

Ok, House 1, I should go that side ←

Embedded C

Pointers - Rule 7 in detail

- Managed by Compiler vs User
- Compiler
  - The compiler will allocate the required memory internally
  - This is done at the time of definition of variables
- User
  - The user has to allocate the memory whenever required and deallocate whenever required
  - This done by using malloc and free

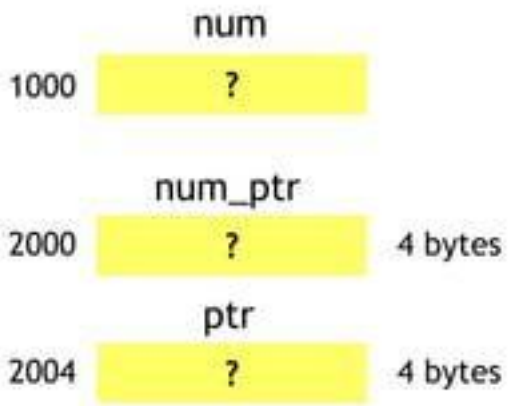
Embedded C

Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



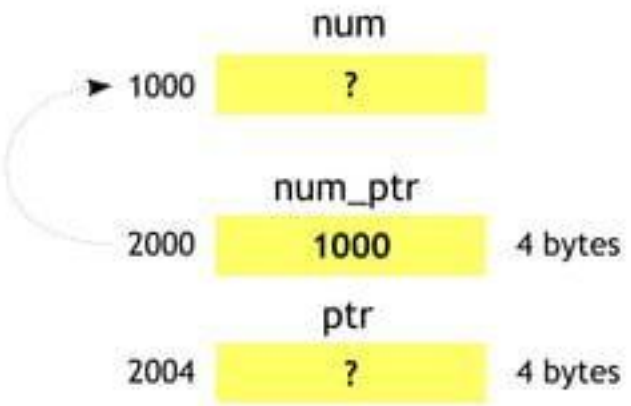
Embedded C

Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



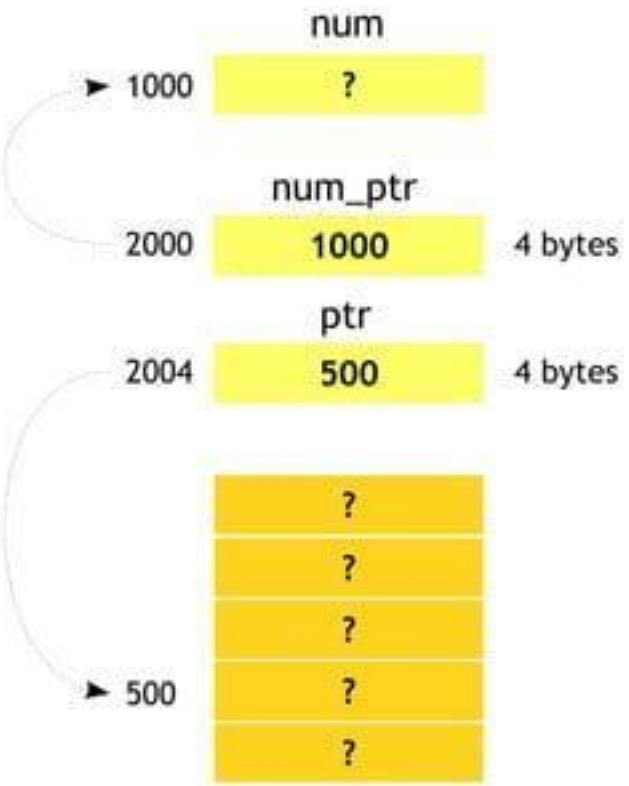
Embedded C

Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(1);

    return 0;
}
```



Embedded C

Pointers - Rule 7 in detail - Dynamic Allocation

- The need
  - You can decide size of the memory at run time
  - You can resize it whenever required
  - You can decide when to create and destroy it.

Embedded C

Pointers - Rule 7 - Dynamic Allocation - malloc

Prototype

```
void *malloc(size_t size);
```

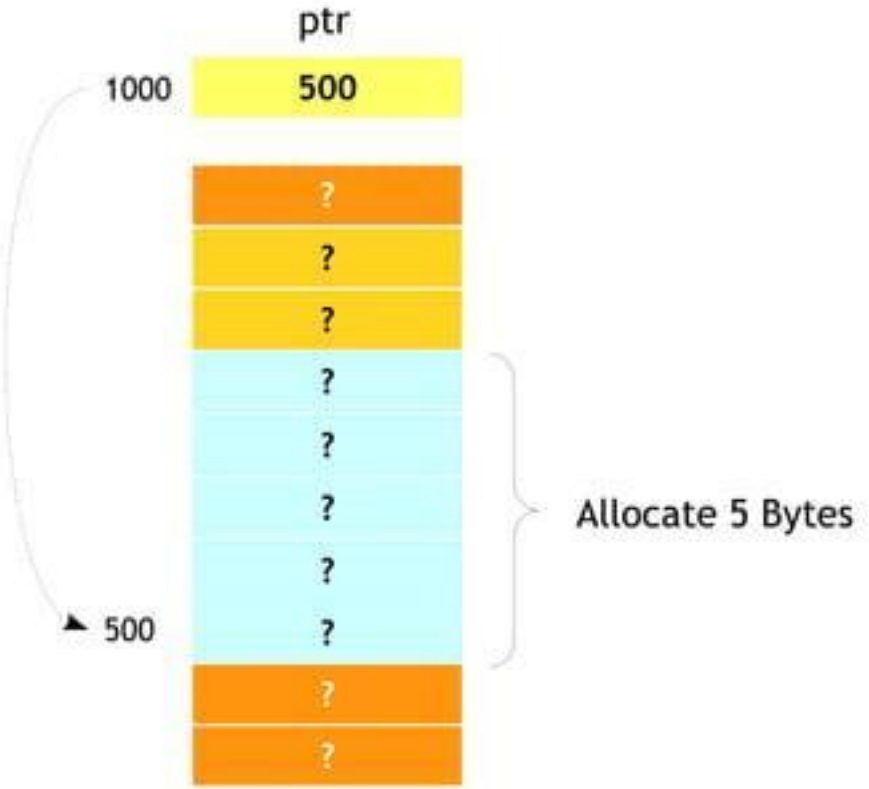
- Allocates the requested size of memory from the heap
- The size is in bytes
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Embedded C

Pointers - Rule 7 - Dynamic Allocation - malloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```

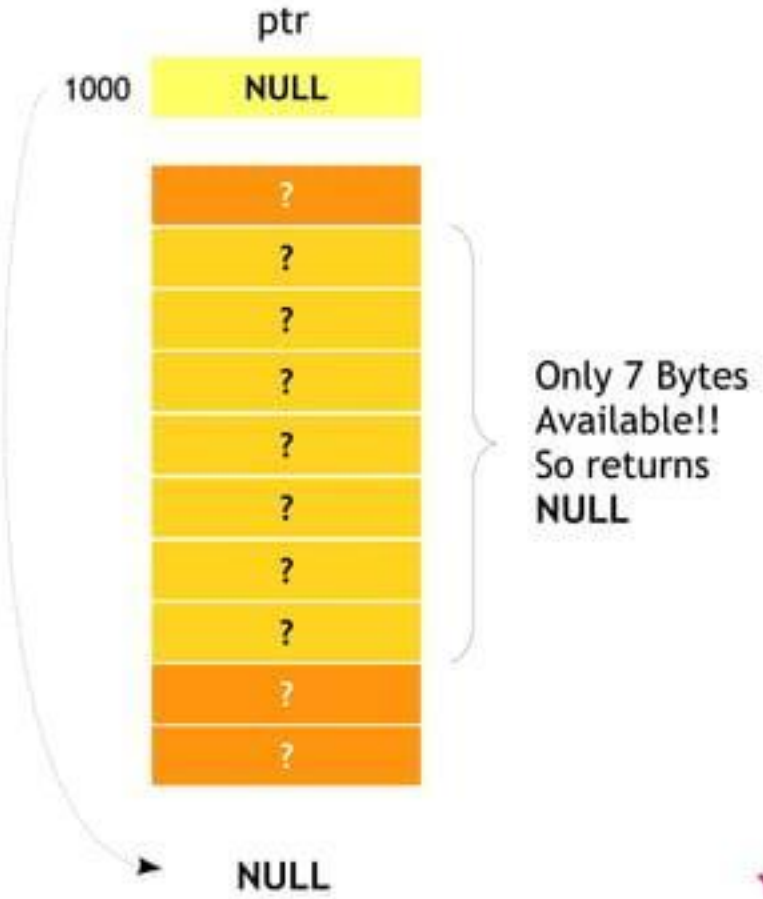


Embedded C

Pointers - Rule 7 - Dynamic Allocation - malloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(10);

    return 0;
}
```



Embedded C

Pointers - Rule 7 - Dynamic Allocation - calloc

Prototype

```
void *calloc(size_t nmemb, size_t size);
```

- Allocates memory blocks large enough to hold "n elements" of "size" bytes each, from the heap
- The allocated memory is set with 0's
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Embedded C

Pointers - Rule 7 - Dynamic Allocation - calloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = calloc(5, 1);

    return 0;
}
```

Embedded C

Pointers - Rule 7 - Dynamic Allocation - realloc

Prototype

```
void *realloc(void *ptr, size_t size);
```

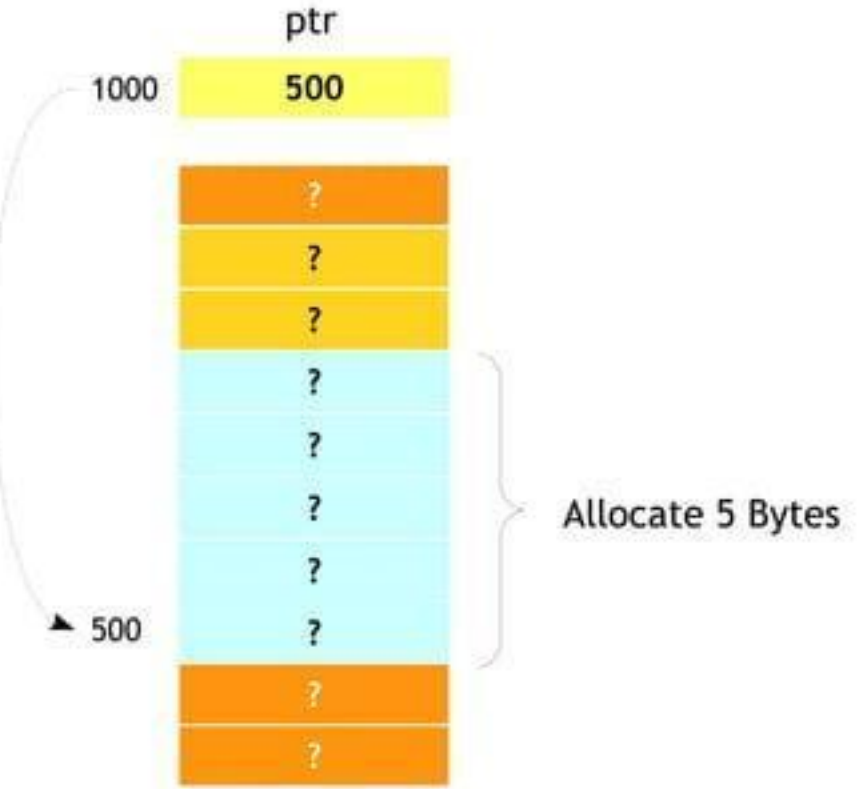
- Changes the size of the already allocated memory by malloc or calloc.
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Embedded C

Pointers - Rule 7 - Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```

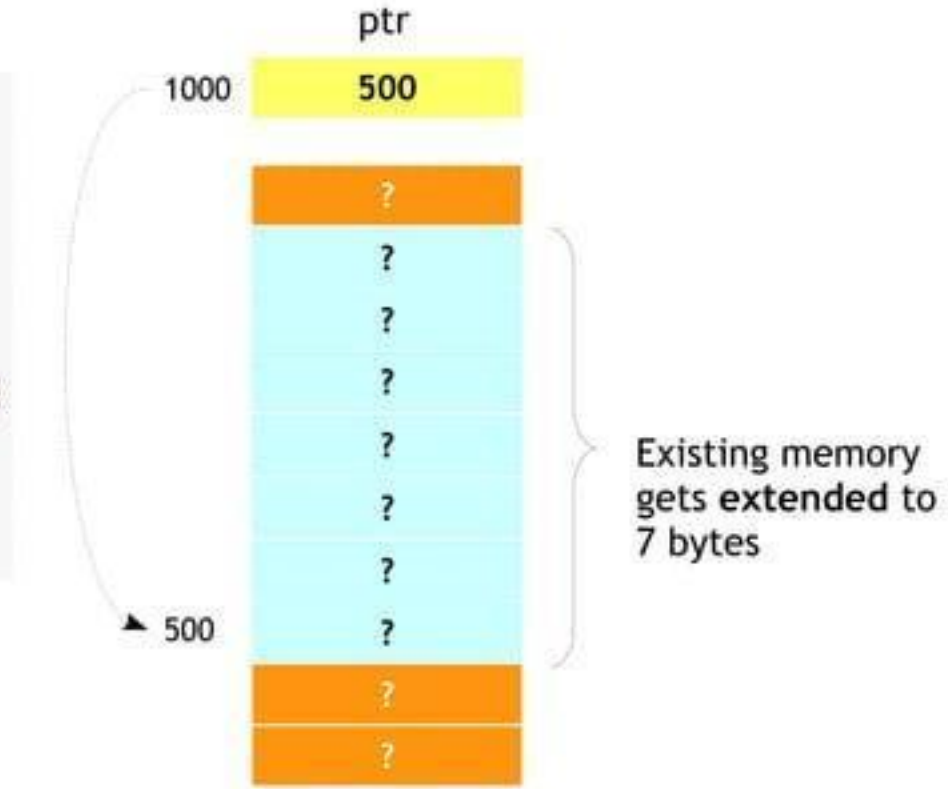


Embedded C

Pointers - Rule 7 - Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```

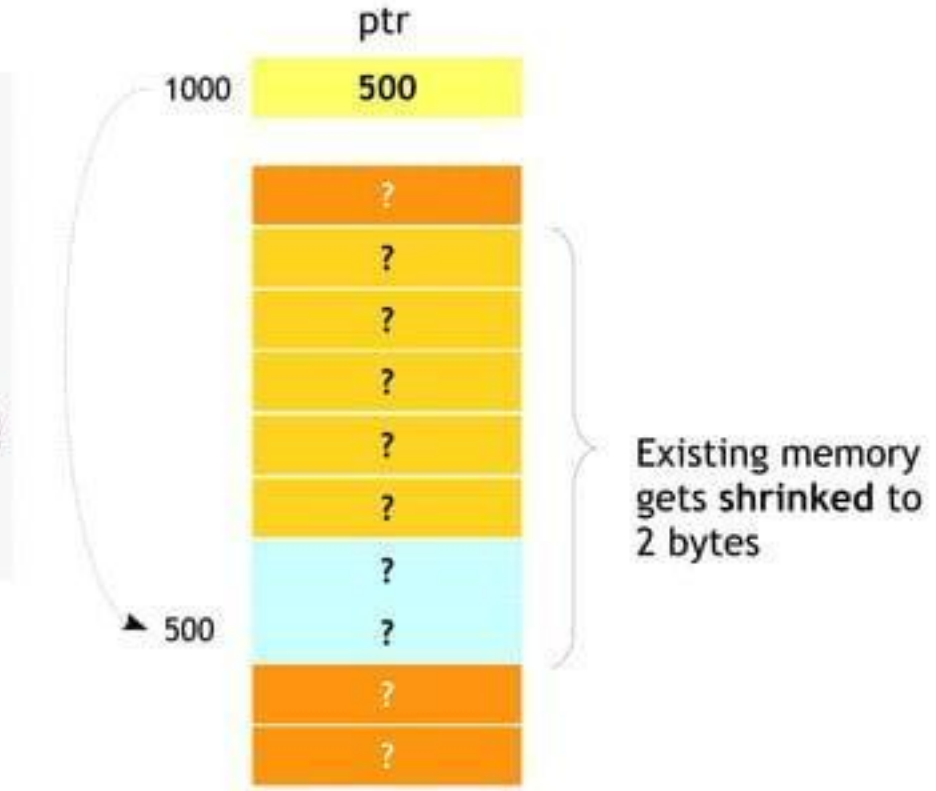


Embedded C

Pointers - Rule 7 - Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    ptr = realloc(ptr, 7);
    ptr = realloc(ptr, 2);

    return 0;
}
```



Embedded C

Pointers - Rule 7 - Dynamic Allocation - realloc

- Points to be noted
  - Reallocating existing memory will be like deallocating the allocated memory
  - If the requested chunk of memory cannot be extended in the existing block, it would allocate in a new free block starting from different memory!
    - So its always a good idea to store a reallocated block in pointer, so that we can free the old pointer.

Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

Prototype

```
void free(void *ptr);
```

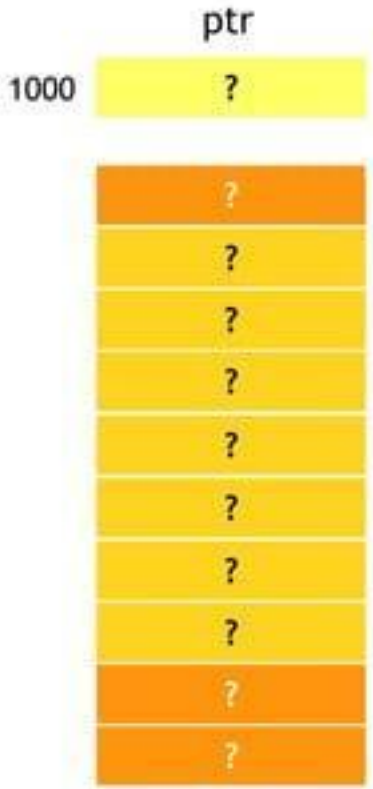
- Frees the allocated memory, which must have been returned by a previous call to malloc(), calloc() or realloc()
- Freeing an already freed block or any other block, would lead to undefined behaviour
- Freeing NULL pointer has no effect.
- If free() is called with invalid argument, might collapse the memory management mechanism
- If free is not called after dynamic memory allocation, will lead to memory leak

Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int i;

    ptr = malloc(5);

    for (i = 0; i < 5; i++)
    {
        ptr[i] = 'A' + i;
    }

    free(ptr);

    return 0;
}
```

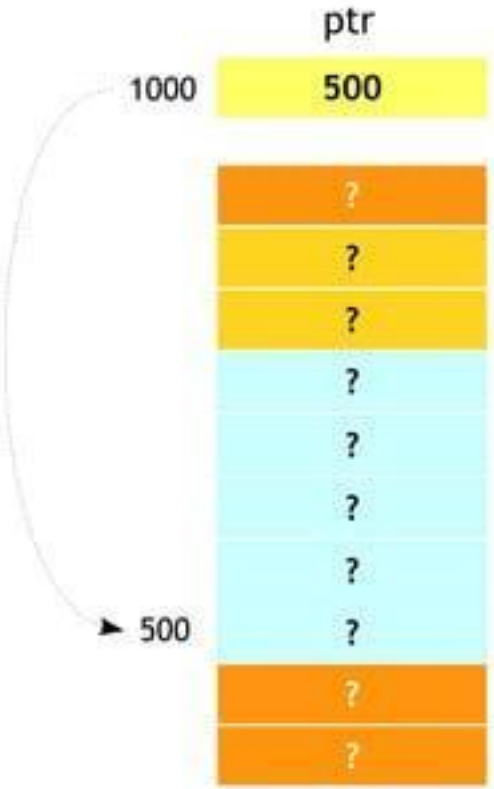


Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int i;

    ptr = malloc(5);

    for (i = 0; i < 5; i++)
    {
        ptr[i] = 'A' + i;
    }

    free(ptr);

    return 0;
}
```

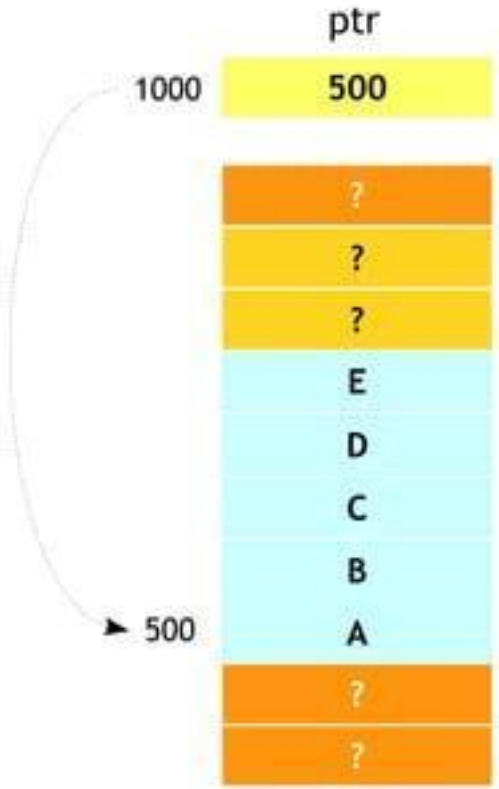


Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int i;

    ptr = malloc(5);

    for (i = 0; i < 5; i++)
    {
        ptr[i] = 'A' + i;
    }

    free(ptr);

    return 0;
}
```

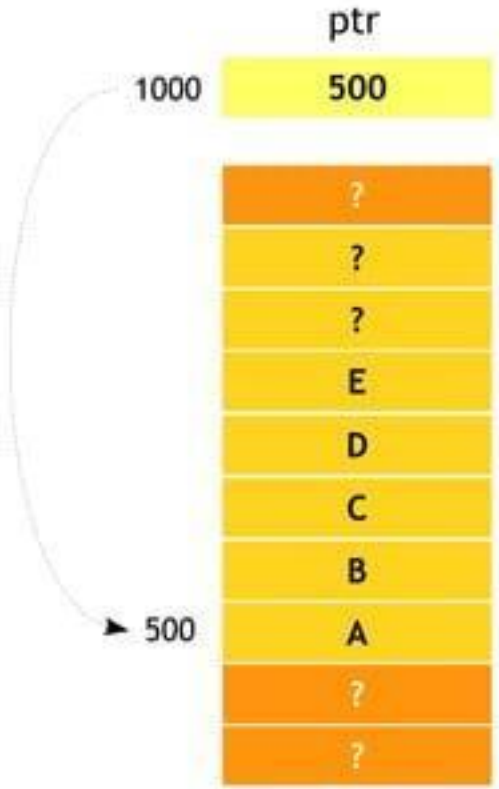


Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;
    int i;

    ptr = malloc(5);

    for (i = 0; i < 5; i++)
    {
        ptr[i] = 'A' + i;
    }

    free(ptr);

    return 0;
}
```



Embedded C

Pointers - Rule 7 - Dynamic Deallocation - free

- Points to be noted
  - Free releases the allocated block, but the pointer would still be pointing to the same block!!, So accessing the freed block will have undefined behaviour.
  - This type of pointer which are pointing to freed locations are called as **Dangling Pointers**
  - Doesn't clear the memory after freeing

Will meet again