

# Basic Refresher

## Number System

- In the mathematics there are mainly 4 type of number system
- Decimal (Base 10, 0 to 9)
- Binary (Base 2, 0 and 1)
- Octal (Base 8, 0 to 7)
- Hexadecimal (Base 16, 0 to 16)

What is the purpose of learning here? When we type any letter or word, the computer translates them into numbers since machines can only understand the binary numbers. To understand this conversion better we will explore the number system here.

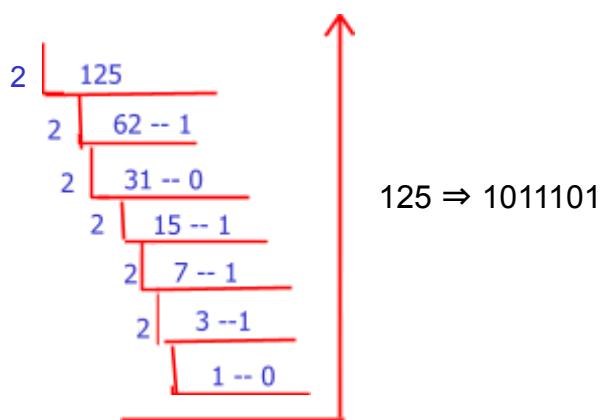
## Decimal to Binary Conversion

- Whenever a decimal number has to be converted to binary we will always use the LCM method by dividing the number by 2.

E.g., number = 125, then binary will be

Decimal to binary

125 to binary



## Binary to decimal

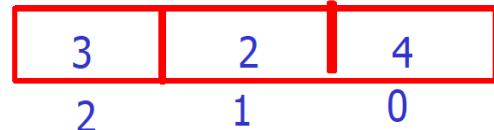
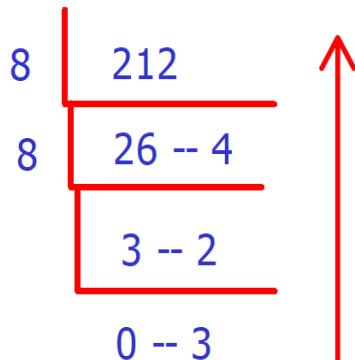
$$\Rightarrow 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 + 1 * 2^4 + 1 * 2^5 + 1 * 2^6 + 0 * 2^7$$

$$\Rightarrow 1 + 0 + 4 + 8 + 16 + 32 + 64 + 0$$

$$\Rightarrow 125$$

### Decimal to octal

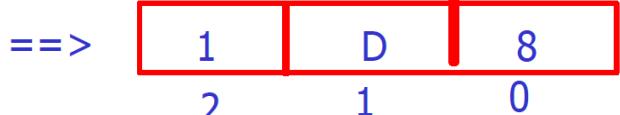
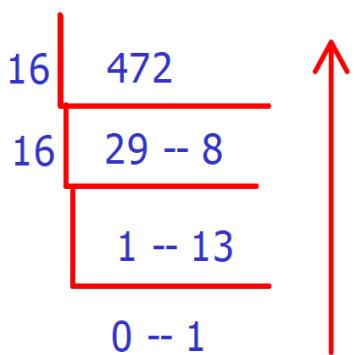
212 to octal ==> 0324



$$\begin{aligned} & \Rightarrow 3*8^2 + 2*8^1 + 4*8^0 \\ & \Rightarrow 192 + 16 + 4 \\ & \Rightarrow 212 \end{aligned}$$

### Decimal to hexadecimal

472 to hexadecimal ==> 1D8

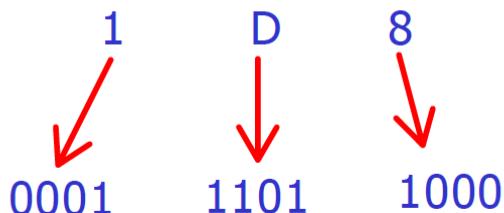


$$\begin{aligned} & 1 * 16^2 + D * 16^1 + 8 * 16^0 \\ & 256 + 208 + 8 \\ & 472 \end{aligned}$$

### Hexadecimal to binary :

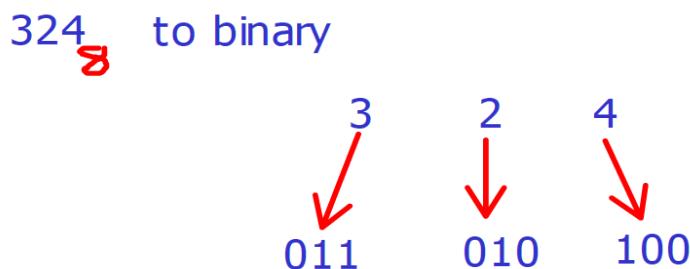
- The maximum value of hexa is F which is 15, so to represent 15 4 bits are required.
- Therefore whenever hexa is to be converted to binary then each digit should be converted to 4bit binary value like below:

1D8 to binary  
16



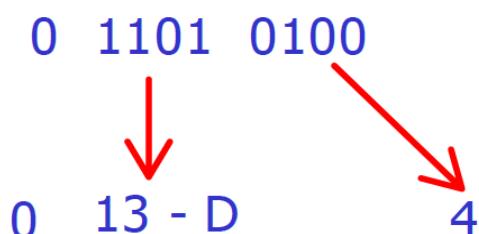
### Octal to binary :

- The maximum value of octal is 7, so to represent 7, 3 bits are required.
- Therefore whenever octal is to be converted to binary then each digit should be converted to 3bit binary value like below:



### Octal to hexadecimal:

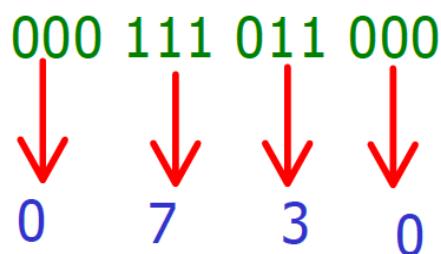
- To convert octal number to hexadecimal
  - Convert octal number to binary
  - group binary number into 4 bits each
  - take the equivalent 4-bit value of hexa
  - E.g., 324 octal to hexa
  - 324 in binary is 011 010 100
  - Group above binary to 4 bits then it will be:



- So, 324 octal == D4 in hexa

### Hexadecimal to octal:

e.g, 1D8 in octal



## Data Representation

- In a computer system different data is represented in a specific way. Here, we will try to explore different ways of representing data in computer
- First, is Bit representation, it's the only language which is understood by the machine.
  - Bit is derived from the word binary digit.
  - It has 2 possible states: true or false or 0 or 1.
- Byte is a collection of 8-bit.
  - its a smallest unit of information in the computer memory.
- Character is a 1byte integer which is represented differently in different languages with the help of character code like, ASCII, EBCD, UTF etc.
  - In C, character is represented with the ASCII code.
  - Characters are represented with ‘ ’ and a character will be 1byte information
  - For example if you use any character internally these are represented as follows:

'0' → 48(ASCII code) → 0011 0000

'A' → 65 → 0100 0001

- Next, word is the capacity of a processor, which defines how much information it can fetch or process at a time.
  - For example, 8-bit MC → MC can fetch 8-bit of information at one time/one cycle 32 --> fetching 32-bits of information at a time.
- Integer numbers are represented as a direct binary conversion
  - for example, positive 13 will be in 32 bits  
0000 0000 0000 0000 0000 0000 0000 1101
  - Negative number -13 will be stored as 2s complement

1's complement + 1

0000 0000 0000 0000 0000 0000 0000 1101

1111 1111 1111 1111 1111 1111 1111 0010  
+0000 0000000 0000 0000 00000000 0001  
1111 1111 1111 1111 1111 1111 1111 0011

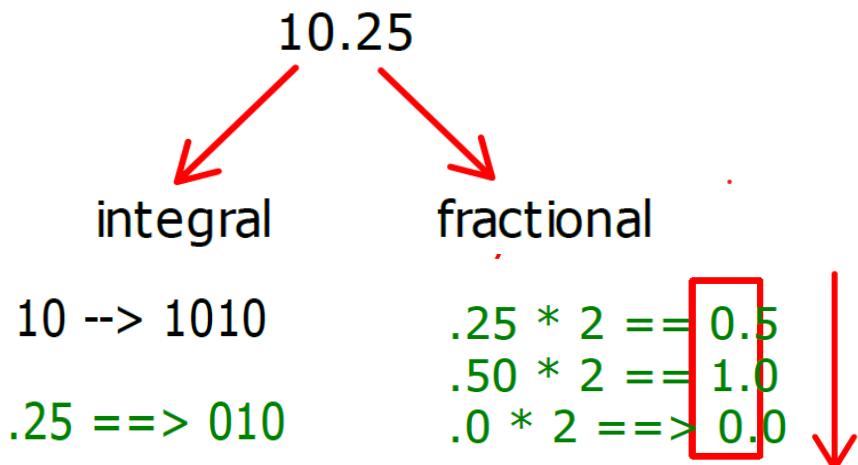
- In other word -13 in 2's complement can be in 8-bit system

==> 256 -13 ==> 255

# Float Representation

- Real values in C are represented as a IEEE 754 standard format which follows following steps

step 1: Convert fractional and integral value to binary



SO, 10.25 ⇒ 1010.010

Step 2 : Convert the result of step1 to standard exponent formula

1.  $\text{Xe}^{+/-y}$

1010.010  
101.0010  
10.10010  
1.010010

y --> number of shift you will do to get the format  
y=3  
if shift is towards left then value is +ve  
or if towards right side value is -ve

X --> mantissa --> 0100100000000000000000000 (23 bits in float)

sign bit - 0

**step 3: Find the exponent part**

--> add or subtract v with the standard bias number

--> for floating point standard bias number is --> 127

--> if v is positive add else if negative subtract

$$\text{exponent} = 127 + 3 == 130 \Rightarrow$$

- If negative only is given then only sign bit will be 1 other rest of the steps are same as previous
  - If the value is double then step 1 and 2 remain the same. In step 3 the standard bias number will be 1023 for double.

2. 1.7 --> never ending number

$\Rightarrow 1$

1.10110011001100110.....

$$.7 * 2 = 1.4$$

$$.4 * 2 = 0.8$$

$$.8 * 2 = 1.6$$

$$.6 * 2 = 1.2$$

$$.2 * 2 = 0.4$$

$$.4 * 2 = 0.8$$

$$A.8 * 2 = 1.$$

$$.6 * 2 = 1.2$$

$$2 * 2 = 0.4$$

step 2: no need to convert so

$$\rightarrow y=0$$

X==> 1 0110 0110 0110 0110 0110 01

**sign = 0**

### step 3: exponent

$127 + 0 == 127 \Rightarrow 0111\ 1111$

0	0	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

For double

--> All the steps are same only change is representation of exponent and mantissa and standard bias number is 1023

--> exponent --> 11

--> mantissa --> 52 1. 2.5

10.10

$$2. y = 1$$

1.010e

sign = 0

### 3. exponent -->

$1023 \Rightarrow 1023 + 1 \Rightarrow 1024 \Rightarrow 1000000000$

0 01000000000 0100

## Data types

- Data Type specifies which type of value a variable has and how much memory can be allocated to each.
  - In C data types are divided into primitive/basic and non primitive(user defined) data types.
  - In basic data types following are the division:
    - Integral
      - int : integer type and memory allocation is compiler implementation dependent. If turboc then sizeof(int) is 2 bytes, If gcc then the sizeof(int) is 4 bytes.
      - char: Character type is a 1byte integer data. All the characters in C are converted to ASCII equivalent code.
    - Real
      - Float: 4bytes real value which is a single precision data type
      - Double: 4byte real value which is a double precision datatype.
  - Size of datatype can be obtained with the help of sizeof operator.
    - sizeof(variable\_name);
    - sizeof(data type);
    - sizeof(constant);
    - sizeof datatype;
    - sizeof returns the value in unsigned int(%u) if 32 bit system else unsigned long int(%lu) in 64 bit system.
    - The format specifier is %zu which is a portable one.

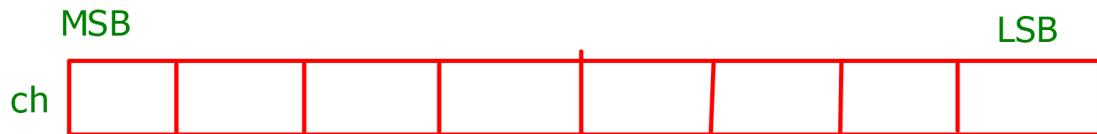
## Modifiers and Qualifiers

- Modifiers and qualifiers are the special keywords which will modify the width, accessibility and memory scope of the variables.
- Modifiers are divided as follows:
  - size modifier: short, long, long long

short	long	long long
<ul style="list-style-type: none"> <li>• short modifiers can be used with only int datatype</li> <li>• It will modify the width of int to 2bytes in gcc.</li> <li>• short int num; short num;</li> <li>• format specifiers: int - %hd hexa - %hx/%hX octal - %ho</li> </ul>	<ul style="list-style-type: none"> <li>• long modifiers can be used either with int or double data type</li> <li>• It will modify the width of int and double which is compiler implementation dependant - for int 4/8, for double - 12/16.</li> <li>• long int num; long num;//int long double num;</li> <li>• format specifiers: int - %ld hexa - %lx/%lX octal - %lo double - %Lf</li> </ul>	<ul style="list-style-type: none"> <li>• long long modifiers can be used with only int datatype</li> <li>• It will modify the width of int depending on the compiler. Sizeof long long will be 8byte/16 bytes</li> <li>• long long int num; long long num;</li> <li>• format specifiers: int - %lld hexa - %llx/%llx octal - %llo</li> </ul>

- sign modifiers: signed and unsigned
  - signed is a default modifier for the integral data type. Signed variables are capable of holding both positive and negative values.
  - In signed, MSB is dedicated to the sign bit.
  - So, the range of the signed variable can be calculated as,
    - $-2^{(n-1)}$  to  $+2^{(n-1)}-1$
    - if character then the range is:  $-2^{(8-1)}$  to  $+2^{(8-1)}-1 = -128$  to  $+127$
    - The values outside the specified range then it will result in overflow and underflow(discussed later)

example: signed char ch;



- Here, MSB(Most Significant Bit is dedicated for sign bit), if 0 its a positive number, if 1 then negative number,
- So,

--> **0** 000 0000      **for positive**

**0** 111 1111      **0 to 127**

**for negative**

--> **1** 000 0000      0111 1111  
                        1  
                        1000 0000      -128  
**1** 111 1111      0000 0000  
                        +1  
                        0000 0001 --> -1

**-128 to -1**

- The range will be, -128 to +127\
- Example,
  - signed char ch = 123; //binary is: 0111 1011
  - char ch = 133;
    - Here, binary of 133 = 1000 0101, MSB bit is 1
    - So, compiler will take the 2s complement of the number

0111 1010

0000 0001

0111 1011 → -123

## Unsigned Variables

- Only positive values are stored in unsigned types.
- No special dedication for MSB bit.
- So, range = 0 to  $2^n - 1$ , so for unsigned char range = 0 to 255
- Example,

unsigned char ch = -13;

- Here, first negative value will be converted to 2s complement

• 0000 1101

1111 0010

00000001

1111 0011

- When you try to print the value then the output will be positive equivalent of 1111 0011 which is, 243

- Now, variables can be declared with both the signedness and size modifiers like,
  - unsigned long int;
  - long long int; // by default its signed
  - short unsigned int;

# Statements

## Conditional Constructs

- In C, conditional constructs are used to construct the statements depending on the condition whether it's true or false. Its divided into 2 groups:
  - Single iterative : Here, statements will be executed only else depending on the condition.
  - Multiple iterative: Here, statements are executed repeatedly until the certain condition is true. Once the condition becomes false it will stop the loop.

### Single Iteration:

- If and its family: If is a single iterative conditional construct which will execute the statements only once.

syntax:

```
if(expression)
```

```
{
```

```
    statements;
```

```
}
```

- Examples:

```
int num = 10;
```

```
if(num == 10)
```

```
{
```

```
    printf("num is equal to 10\n");\n
```

```
}
```

```
if (num < 5);
```

```
printf("If: num < 5\n");
```

```
if (num < 5)
```

```
{
```

```
    //dummy code
```

```
}
```

```
printf("If: num < 5\n")
```

```
if (num < 5);
```

```
{
```

```
    printf("num < 5\n");
```

```
}
```

```
else
```

```
{
```

```
    printf("num == 5");
```

```
}
```

```
if(num<5)
```

```
{
```

```
;
```

```
}
```

```
{
```

```
printf("num < 5\n");
```

```
}
```

```
else
```

```
{
```

```
printf("num == 5");
```

```
}
```

## **switch conditional construct**

- Switch is also a single iterative condition construct which will execute the statements once which is matching with the case label.

syntax:

```
switch(expression)
```

```
{
```

```
    case label:
```

```
        statements;
```

```
        break;
```

```
    case label:
```

```
        statements;
```

```
        break;
```

```
    default:
```

```
        statements
```

```
}
```

- Where, case label has to be integral constant or enum.
- Real constants are not allowed --> 2.5,4.5,7.8.....
- Case label should be unique
- Multiple values are not allowed in case label
- A break statement is used to break the switch cases.
- If break is not used then switch will result in fall through condition like
  - switch(expression)

```
{
```

```
    case 10:
```

```
        printf("10 is selected\n");
```

```
    case 20:
```

```
        printf("20 is selected\n");
```

```
    default:
```

```
        printf("None is matching\n");
```

```
}
```

- In the above example if the expression is having 10, then all the 3 printf is executed.

## **switch vs if else**

- consider the below scenario where  $x = 10$ , then execution time for if is

```
if (x == 100)           1    cycle
{
}
else if(x == 90)        2    cycle      200nsec * 4 == 800nsec
{
}
else if(x == 80)        3    cycle 4th cycle
{
}
else
{
}
```

For switch JUMP default:

```
case 100:
.
.
.
case 90:               default:
.
.
.
case 80:               1cycle == 200nsec
.
.
```

## **Multi Iteration Conditional construct:**

- While loop is known as entry controlled loop where statements are executed repeatedly until the condition is true.
- syntax:

```
while(expression)
{
    statement;
}
```

- Example:

```
int iter = 0;
while(iter < 5)
{
    printf("Looped %d times\n",iter);
    iter++;
}
```

- In the above loop, iter is 0 so the condition is true it will enter the loop.
  - prints, looped 0 times then increments the iter = 1
  - continues till iter = 5
  - Once the condition is false comes out of the loop.

```
1. whileÃ¢int main()
{
int iter;
iter = 0;
while (iter < 5)
printf("Looped %d times\n", iter);
iter++;
return 0;
}
```

Interpretation:

```
int main()
{
int iter;
iter = 0;
while (iter < 5)
{
printf("Looped %d times\n", iter);
}
iter++;

return 0;
}
```

Output: looped 0 times

2.

```
int main()
{
int iter;
iter = 0;
while (iter < 5);
{
    printf("Looped %d times\n", iter);
    iter++;
}
printf("iter: %d\n", iter);

return 0;
}
```

```
int main()
{
int iter;
iter = 0;
while (iter < 5)
{
    ;
}
{
    printf("Looped %d times\n", iter);
    iter++;
}
printf("iter: %d\n", iter);

return 0;
}
```

```

int main()
{
int iter; iter = 6;
while (iter < 5);
{
printf("Looped %d times\n", iter); iter++;
}
printf("iter: %d\n", iter);

return 0;
}

```

### compiler view

```

int main()
{
int iter; iter = 6;
while (iter < 5)
{
;
}
{
printf("Looped %d times\n", iter); iter++;
}
printf("iter: %d\n", iter); return 0;
}

```

## do...while loop

```

iter = 0;
do
{
printf("Looped %d times\n", iter);
iter++;
} while (iter < 5);

```

1. Looped 0 times      3. looped 2 times  
iter = 1                  iter = 3  
1 < 5                  3 < 5
2. looped 1 times      4. looped 3 times  
iter = 2                  iter = 4  
2 < 5                  4 < 5
5. looped 4 times      5 < 5 --> exit the loop  
iter = 5

## While vs do..while

### While

--> entry controlled loop  
if the condition is true then enters loop

--> if condition is true then only statements will be executed

where and when?

number of times/iteration is unknown

### do..while

--> exit controlled loop  
enter the loop then check for the condition

--> atleast for once the statements gets executed

--> Menu-driven application

ATM --> do you want to continue - y/n  
Gaming -->

## For loop:

- For loop is also like while loop, which will execute the statements repeatedly until the condition is true.
- For loop syntax:

```
for(initialisation; condition ; post evaluation)
{
    statements;
}
```
- For loop will be executed as stated below:
  1. Initialization
  2. Condition - if true goto step 3 else goto step 6
  3. Statements
  4. Post evaluation expression
  5. Goto step 2
  6. Exit

1.

```
int main()
{
int iter = 0;

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

```
int main()
{
int iter = 0;
for (iter = 0 ; iter < 10; iter++ )
{
;
}
{
printf("Looped %d times\n", iter);
}
return 0;
}
```

2.

```
while(iter++ < 5)
{
printf("Looped %d times\n", iter);
}
```

1.  $0++ < 5 \Rightarrow 0 < 5$   
 $(\text{iter} = \text{iter}+1)$   
Looped 1 times

2.  $1++ < 5 \Rightarrow 1 < 5$   
 $(\text{iter} = \text{iter}+1)$   
Looped 2 times

3.  $2++ < 5 \Rightarrow 2 < 5$   
 $\text{iter} = \text{iter}+1$   
looped 3 times

Post increment -->  
assign the value/compare the value  
next time use the incremented value

4.  $3++ < 5 \Rightarrow 3 < 5$   
 $\text{iter} = \text{iter}+1$   
Looped 4 times

5.  $4++ < 5 \Rightarrow 4 < 5$   
 $\text{iter} = \text{iter}+1$   
Looped 5 times

6.  $5++ < 5 \Rightarrow 5 < 5 \rightarrow \text{false}$   
 $\text{iter} = \text{iter}+1 = 6$

3.

```
while(++iter < 5)
{
    printf("Looped %d times\n", iter);
}
```

1. ++0 < 5 ==> 1 < 5  
Looped 1 times
2. ++1 < 5 ==> 2 < 5  
looped 2 times
3. ++2 < 5 ==> 3 < 5  
looped 3 times

Pre increment==>  
first increment the value and then assign/  
compare

4. ++3 < 5 ==> 4 < 5  
looped 4 times
5. ++4 < 5 ==> 5 < 5  
exit the loop

### Nested for loop:

#### Nested for loop

```
for(i = 0; i < 3; i++)
{
    for(j = 0 ; j < 2 ; j++)
    {
        printf("Hello\n");
    }
}
```

	col 0	col 1
row 0	hello	hello
row 1	hello	hello
row 2	hello	hello

1. i = 0  
0 < 3  
  1.1 j = 0  
    0 < 2  
    Hello  
  1.2 j = 1  
    1 < 2  
    Hello  
  1.3 j=2  
    2 < 2 --> false  
    --> exit inner loop

2. i = 1  
1 < 3  
  2.1 j = 0  
    0 < 2  
    Hello  
  2.2 j = 1  
    1 < 2  
    Hello  
  2.3 j=2  
    2 < 2 --> false  
    --> exit inner loop

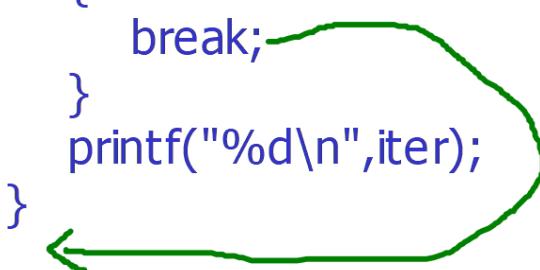
3. i = 2  
2 < 3  
  2.1 j = 0  
    0 < 2  
    Hello  
  2.2 j = 1  
    1 < 2  
    Hello  
  2.3 j=2  
    2 < 2 --> false  
    --> exit inner loop

4. i = 3  
3 < 3 --> false --> exit the loop

## Break:

- used to exit from the loop or switch case depending on the condition
- Break should be within the loop

```
for(iter=0;iter<10;iter++)           1. iter = 0
{
    if(iter == 5)                   2. iter = 1
    {
        break;                    3. iter = 2
    }
    printf("%d\n",iter);          4. iter = 3
}
6. iter = 5                          5. iter = 4
```



continue statement

--> only used in multi iterative conditional construct --> do..while,while,for

--> used to skip that specific iteration of the loop

```
for(iter=0;iter<10;iter++)           1. iter=0
{
    if(iter == 5)                   2. iter=1
    {
        continue;                  3. iter=2
    }
    printf("%d\n",iter);          4. iter=3
}
5. iter=4
6. iter=5
7. iter=6
8. iter=7
9. iter=8
10. iter=9
11. iter=10
```



## Goto Statement--> unconditional jump

```
int iter = 5;  
  
goto lab1;  
printf("Hello\n");  
  
lab1:  
printf("World\n");
```

--> generally it is avoided due to difficulty in tracing the flow

Which one is Efficient?

for(*i* = 0 ; *i* < 10; *i*++)  
  for(*j* = 0 ; *j* < 100 ;*j*++)

1. outer loop:  
  initialization:1  
  post evaluation expression: 10  
  condition: 11  
  total 22 machine cycle
2. inner loop: for single  
  initialisation: 1  
  condition: 101  
  post eval expre:100  
  
  total  $202 * 10 == 2020$

$22+2020 ==> 2042$  Machine cycle

for(*i* = 0 ; *i* < 100; *i*++)  
  for(*j* = 0 ; *j* < 10 ;*j*++)

1. Outer Loop  
 $1 + 101 + 100 == 202$   
  
inner loop  
 $1+11+10 == 22 * 100 == 2200$   
  
total ==>  $202 + 2200$   
          ==> 2402 machine cycle

1st one is efficient

## Operators

- Operator is a special symbol which will perform a specific task on the operand and returns a value.
- Operators are divided on operands as below:
  - Unary : One operand - `++`, `--`, `+`, `-` etc
  - Binary: arithmetic, logical, bitwise etc.
  - Ternary: `?:`

### Unary Operators

--> used with one operand  
--> highest precedence in the table

<code>int x, y=10;</code>	<code>x = -y;</code>	<code>int x, y=-10;</code>
<code>x=y;</code>	<code>x = -(y);</code>	<code>x = -y; // x = -(-10)</code>
<code>x = +y;</code>	<code>x = -10</code>	<code>x = 10</code>

```
int x, y=-10;  
  
x = + - y; //+ - (-10)  
  ^  
x = +10
```

### Increment and Decrement Operator

#### 1. Pre increment

- 1 --> increment first
- 2 --> assign the value

`y = ++x;`      `int y, x = 10`

- |                         |   |
|-------------------------|---|
| <code>1. x = x+1</code> | <code>2. Incremented value will be assigned to y</code> |
| <code>x = 10+1</code>   | <code>y = 11</code>                                     |
| <code>x=11</code>       |   |

## 2. Post increment

- 1 --> assign first/use the original value first
- 2 --> increment the value of variable

`y=x++;`

1. x value will be assigned to y, i.e, 10 will be stored in y  
 $y = 10$

2.  $x = x+1$

$x = 10+1$

$x = 11$

## 3. Pre decrement

--> decrease value by 1

$x-- ==> 9$

$--x ==> 8$

$x=5$

$y = --x;$       4 4

## 4. Post decrement

$y = x--;$        $y = 5 \ x = 4$

5.  $y = y++;$

- These expression will have undefined behaviors because variable y is incrementing assigning at the same time
- So, output cannot be predicted.

```
6. x =0;  
printf("%d %d %d %d\n",++x, x++, x++, ++x);  
• This also results in undefined behavior
```

1.  $\text{++}x ==> 1$

2.  $x++ ==> 1$   
 $x = 2$

3.  $x++ ==> 2$   
 $x=3$

4  $\text{++}x ==> 4$



4 2 1 1

- Because of post increment and pre increment in the same line
- I am supposed to get 4 2 11 but output will be different

### sizeof() --> function or operator?

--> for sizeof u can pass any type of data

--> can also pass data type - char int float

--> sizeof(), sizeof var

### Type Conversion

- Type conversion is a process of converting one type to another whenever required.

#### Type conversion

1. int x=10, y=4, z;	$\Rightarrow$	10/4
$z = x / y;$		int = int / int ==> integer result ==> 2
$z = 2$		
2. int x = 10, y = 4;	$\Rightarrow$	$x / y ==> \text{int} / \text{int} = 10 / 4 = 2$
float z;		$z = 2.000000$
$z = x / y;$		

3. int x= 10;  
float y = 4, z;  
z = x / y;

x / y ==> int / float  
==> float / float  
10 / 4 ==> 10.000000 / 4.000000

==> implicit type conversion

==> int is the lower rank type, so it will be converted to higher rank type float

int x = 10, y=4;  
float z;

z = (float) x / y; // explicit type conversion

float / int ==> float / float ==> 10.000000 / 4.000000 ==> 2.500000

## Unary Conversion

--> if char and short both will be converted to integer

char x = 12, y=20, z;  
z = x + y;

==> x + y ==> char + char ==> int + int

char x = 12;  
short int y = 20, z;  
z = x + y;

==> short + char ==> int + int

## Logical Operator – AND, OR, NOT

### 1. AND operator --> &&

A	B	Output	
0	0	0	0 --> false
0	1	0	1 --> True
1	0	0	
1	1	1	

==> For AND if any 1 input is false then output will be false  
if both input is true then only output is true

### 2. OR Operator --> ||

A	B	Output	
0	0	0	if any one input is true then output will be true
0	1	1	
1	0	1	
1	1	1	

### 3. NOT Operator --> !

A	Output
0	1
1	0

Examples:

1. int a = 10,b=20;  
a && b ==> 10 && 20 ==> true && true ==> true ==> 1 1

All the values are true except 0, NULL

## Circuit Logic:

- Circuit logic is the way of evaluation used by compiler to optimize code
- For example if OR is used, num1 = 1, num2 = 0, then

```
if (++num1 || num2++)  
{  
    //statement  
}
```

Here, ++num1  $\Rightarrow$  ++1 is true value so second is not evaluated

- Similarly if(num2++ && num1++)

Here, the first expression is false so the second is not evaluated.

## Examples:

1.

a = 50  
b = -50  
c = 0  
d = 10

1

a || b && c && d  
a || (b && c) && d  
a || ((b && c) && d)  
50 || ((-50 && 0) && 10)

2.

a && b && c || d

$\Rightarrow$  (a && b) && c || d  
 $\Rightarrow$  ((a && b) && c) || d  
 $\Rightarrow$  ((50 && -50) && 0) || 10  
 $\Rightarrow$  ((1) && 0) || 10  
 $\Rightarrow$  0 || 1  
 $\Rightarrow$  1

a && b && c && d

0

### **Compound Statements:**

`num1 += num2 += num3 += num4;`

**1. num3 += num4;**  
`num3 = num3 + num4;`  
`= 1.7 + 1.5`  
`= 3.2`

**2. num2 += num3**

`num2 = num2 + num3;`  
`= 1 + 3.2`  
`= 4.2`  
`num2 = 4.2`  
`num2 = 4`

**3. num1 += num2**  
`num1 = num1 + num2`  
`= 1 + 4`  
`= 5`

### **Bitwise Operators**

- Bitwise operators work on bits of the given values.
- Bitwise operators are used whenever a particular bit/s has to be set, clear or toggle.

#### **1. Bitwise AND - &**

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

#### **2. Bitwise OR -- |**

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

### 3. Bitwise XOR -- ^

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

**Examples:**

1. char x = 0x61 y = 0x31

x & y

x = 0110 0001

y = 0011 0001

    
0010 0001 ==> 0x21 ==> 33

2. x = 0xAA y = 0x57

x & y

1010 1010

0101 0111

0000 0010 ==> 0x02

3. int x = 10 y = 15

0000 0000 0000 0000 0000 0000 0000 1010

0000 0000 0000 0000 0000 0000 0000 1111

0000 0000 0000 0000 0000 0000 0000 1010

4. OR -->  $x = 0x61$     $y = 0x13$

$x \mid y$

0110 0001

0001 0011

0111 0011             $0x73 ==> 115$

5.  $x = 0xAA$     $y = 0x53$

$x \mid y$

1010 1010  
0101 0011

1111 1011 --> 0xfb --> 251

XOR -->  $x = 0xBC$     $y = 0x35$

$x \wedge y ==> 1011\ 1100$   
 $0011\ 0101$   
 $1000\ 1001 ==> 0x89 ==> 137$

Bitwise Complement --> ( $\sim$ )

--> 1's compliment of a given number

A	Output
---	--------

0	1
1	0

1.  $x = 0xBC$

$\sim x ==> 1011\ 1100$

0100 0011 ==> 0x43

## Shift Operators

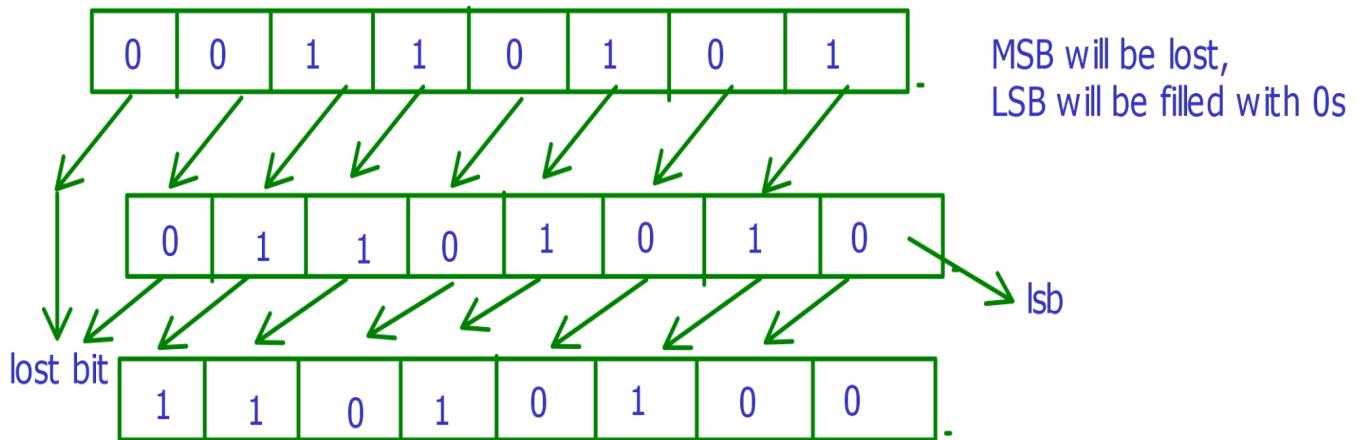
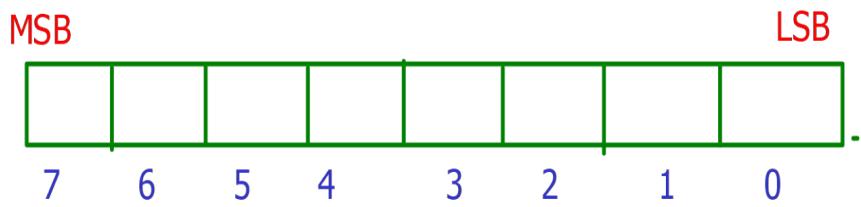
1. Left Shift --> bits will be shifted to left side depending on the number of shift
2. right shift --> bits will be shifted to right side

1. Left shift: <<

value << no\_of\_bits (no\_of\_shift)

char a = 0x35

a << 2



=> 1101 0100 => 0xD4

0x35 --> 53

1. first shift --> 0110 1010 --> 0x6A --> 106

2. second shift --> 0xD4 --> 212

=>  $53 * 2^0$

=>  $53 * 2^1$

=>  $53 * 2^2$

--> efficient way of multiplying 2 to the power values

output of shift

resultant = value << no\_of\_bits ==> resultant \*  $2^{no\_of\_bits}$

2.

2.  $0x16 \ll 3 ==>$

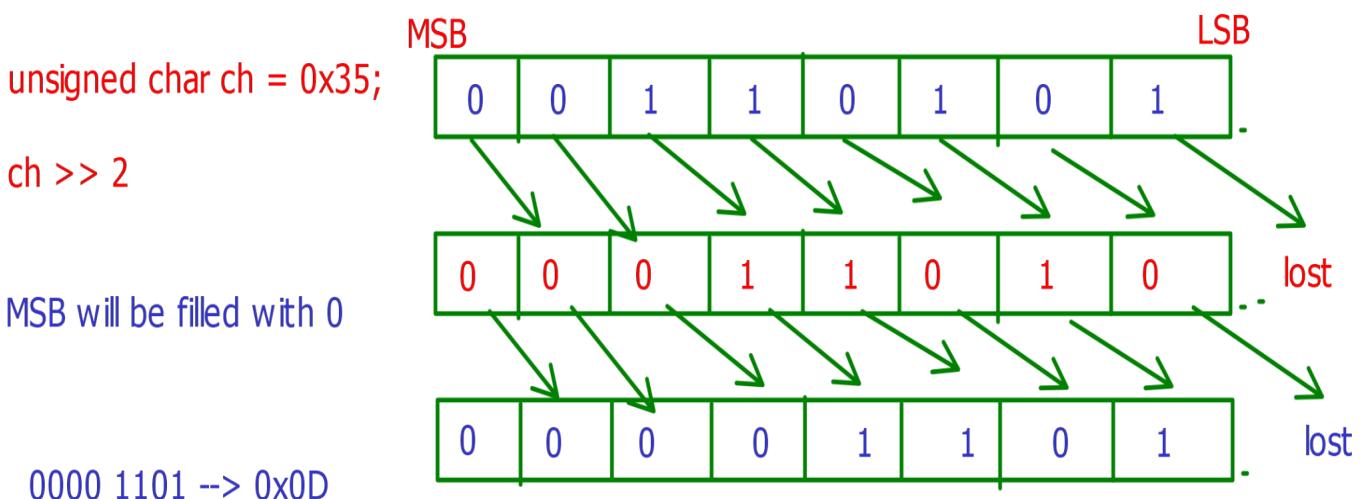
1st shift --> 22 --> 0001 0110 --> 0010 1100 --> 44 --> 2C

2nd shift --> 0101 1000 == 88 == 58

3rd shift --> 1011 0000 == 176 == B0

$22 * 2^3 ==> 22 * 8 ==> 176 ==> B0$

**Right Shift: Unsigned right shift**



--> unsigned right shift efficient way of dividing number by 2 power values

$0x35 \rightarrow 53 ==> 53 / 2^0 ==> 53$

$0x1A \rightarrow 26 ==> 53 / 2^1 ==> 26$

$0xD \rightarrow 13 ==> 53 / 2^2 ==> 13$

value >> no\_of\_bits

resultant = value /  $2^{no\_of\_bits}$

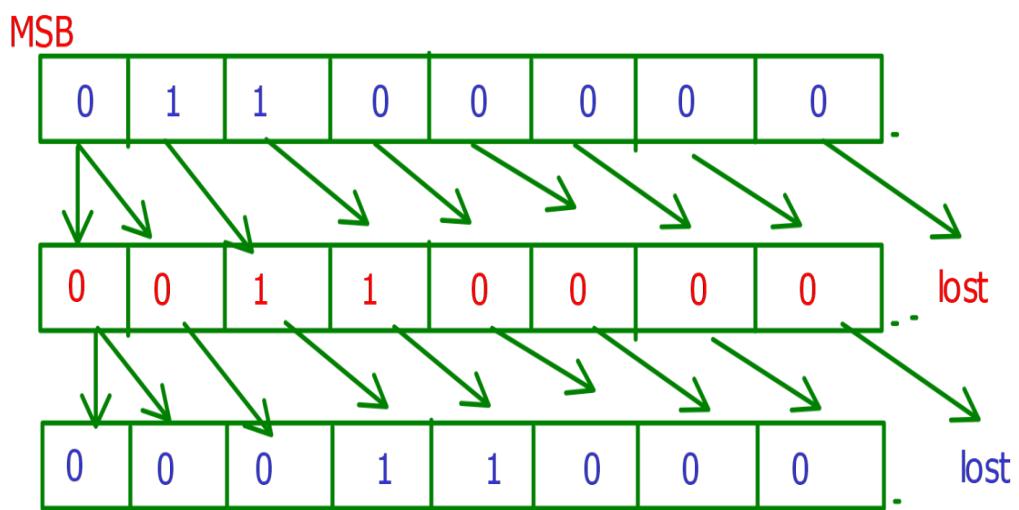
## Signed right shift:

signed right shift

signed char ch = 96;

ch >> 2

previous MSB will be filled with current shift of MSB



$$96 \Rightarrow 96 / 2^0 \Rightarrow 96$$

$$96 \Rightarrow 96 / 2^1 \Rightarrow 48$$

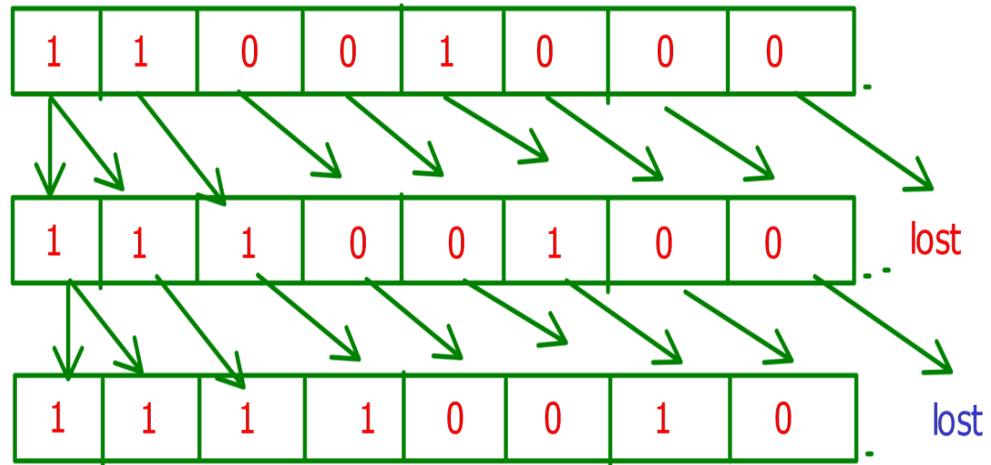
$$96 / 2^2 \Rightarrow 24$$

signed right shift - negative number

signed char ch = -56

ch >> 2

$$\begin{array}{r} 00111000 \\ 11000111 \\ +1 \\ \hline 11001000 \end{array}$$



$$1\text{st shift} \rightarrow 1110\ 0100$$

$$\begin{array}{r} 0001\ 1011 \\ +1 \\ \hline \end{array}$$

$$\begin{array}{r} 0001\ 1100 \\ ==> -28 \end{array}$$

$$-56 / 2^1 = -28$$

$$2\text{nd shift} \rightarrow 1111\ 0010 ==> -14$$

$$-56 / 4 ==> -14$$

### Points to remember:

- ❖ Number of shift should be positive value, negative shift value will result in undefined behavior (output cannot be predicted)
- ❖ right operand (number of bits/shift) has to be within the range of left operand  
for example,

    unsigned char ch = 90;

- so valid right operand values are 0 to 7
- in case of integer --> 0 to 31
- if right operand value is more than the range of bits of left operand then output will be undefined behavior

- ❖ After shifting, if the result is not within the range of datatype then undefined behavior

e.g, signed char ch = 96;

    ch << 4 ==> 1536 //the value is out of range

### Applications of Bitwise operators:

Set bit --> Whatever the bit might be, make it as 1 ( either it can be 1 or 0 )

value = 0xAA

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0

.

set the bit present at the position 4 --> 1011 1010

--> chose any bitwise operator, perform the operation with mask

1010 1010	
0001 0000	0x10 ==> mask
1011 1010	

set the bit present in 6th position

1010 1010	
0100 0000	0x40 ==> mask
1110 1010	

--> whenever it is a set bit, you have to use bitwise OR operator

--> 0xAA | mask == result

--> 1 << 4

0000 0001
0000 0010
0000 0100
0000 1000
0001 0000

1010 1010
0001 0000
1011 1010

1011 1010
-----------

1 << 6

0100 0000
-----------

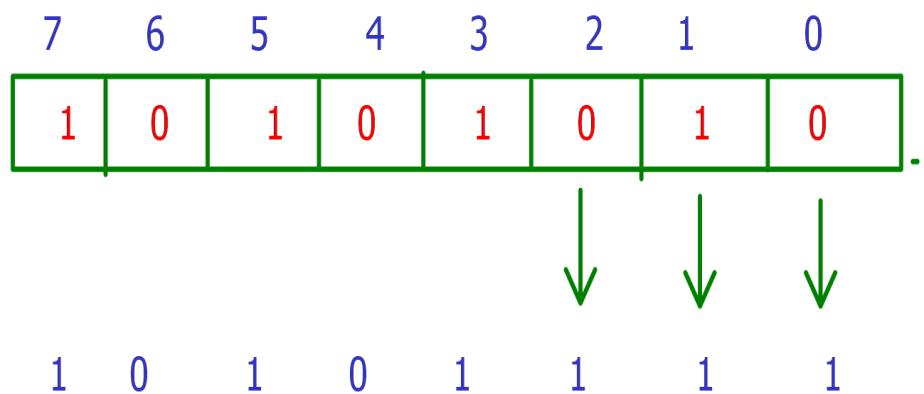
1010 1010
0100 0000
1110 1010

mask = 1 << position

result = value | mask

set 3-bits from lsb of the given value

value = 0xAA



--> 1010 1010

1010 1010
-----------

0000 0101
-----------

0000 0111
-----------

1010 1111
-----------

1010 1111
-----------

Generic mask --> only 3-bits has to unprotected

mask = (1 << no\_of\_bits) - 1

0000 1000 - 1 ==> 7

Clear bit --> whatever the bit may be, you have to make it as 0

clear the 3rd bit of given value  
bitwise AND - &

## Generic mask

1010 1010                    $\sim(1 \ll pos)$   
                                 $\sim(1 \ll 3) ==> 0000\ 1000 => 1111\ 0111$   
1111 0111  
  
1010 0010

$\Rightarrow$  whenever clear bit is asked, you should use bitwise AND operator with desired mask

clear 4 bits from LSB of the given value

**1010 1010  
1111 0000**

1010 0000

generic mask -->	$\sim((1 << 4) - 1)$	0000 1111 1111 0000
mask = $\sim ((1 << \text{no\_of\_bits}) - 1)$	$\sim((16) - 1)$ $\sim 15$	

## Toggle bit --> 0 to 1 and 1 to 0

0xAA ==> 10101010                    1010 0010

11110111  
10100010

1010 1010 ==> 1011 1010  
                  0001 1000

==> whenever toggle bit is asked, use XOR (^)

## Generic Mask

mask = 1 << pos

mask = 1 << 3

1010 1010

res = value ^ mask;

^ 0000 1000  
1010 0010

## Ternary Operator

- works with 3 operands → ?:
- syntax: condition ? expression\_true : expression\_false
- For some scenarios or requirement ternary is the optimization for if..else
- Example:

max = num1 > num2 ? num1 : num2;

num1 > num2 > printf("Num1 is max\n") : printf("Num2 is max\n");

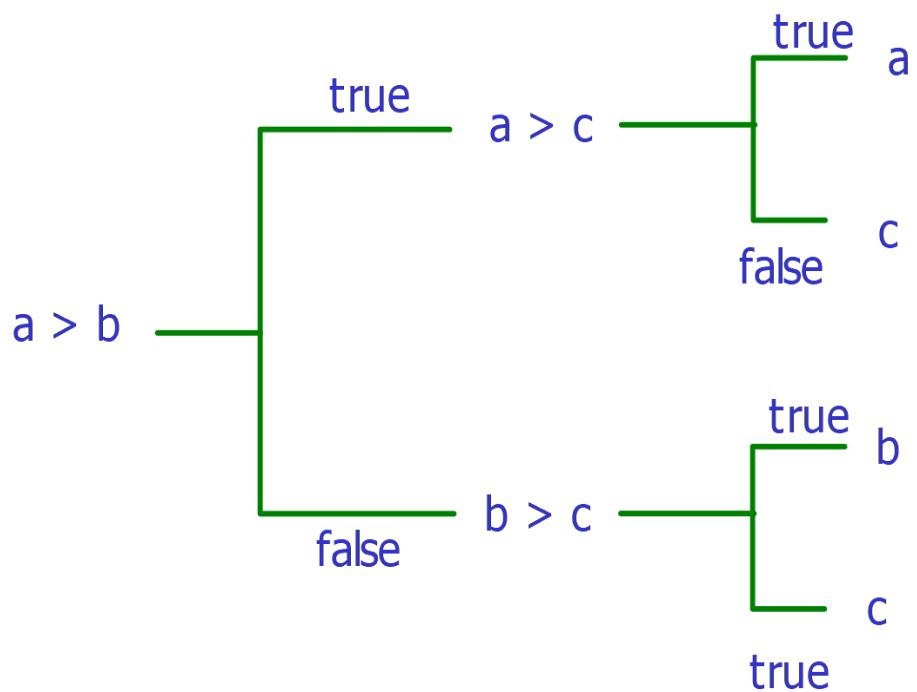
- Can also have nested ternary like,

```

if(a > b)
{
    if(a > c)
    {
        pf(a);
    }
    else
    {
        pf(c);
    }
}
else
{
    if(b > c)
    {
        pf(b);
    }
    else
    {
        pf(c);
    }
}

```

1. `res = a > b ? a > c ? a : c : b > c ? b : c;`



Comma operator:

- Comma operator have certain usage with for loop and in some expression, or function calling
- It will evaluate the expression but consider or return the right most value to the function or expression.

`int num1,num2,num3;`

`num1 = (1, 2, 3)`

Here, `num1=3`

`num1 = (x = 1 + 2, y = 2 + 2, z = 3 + 3);`    `num1 = 6, x = 3, y = 4, z = 6`

`x = 5, 10, 5;`

`x = (y=100, z = y + 10)`

Examples:

j = i++ ? i++, ++j : ++j, i++;

= 0++ ? i++, ++j : ++j, i++;      ==> i = 1

? : ,

(i++ ? (i++, ++j) : ++j), i++;

0++? not eval : ++j, i++;      ==> i = 1, j = 1      i = 2

```
for(i = 0, j = 0;i < 5, j < 10;i++, j++)  
{  
}
```

1. i = 0 , j=0

2. i = 1, j = 1

3. i = 2, j = 2

0 < 5, 0 < 10

1 < 5, 1 < 10

4. i = 3, j = 3

6. i = 5, j = 5

7. i = 6, j = 6  
6 < 5, 6 < 10

5 < 5, 5 < 10

8. i = 7, j = 7

9. i = 8 , j = 8

10. i = 9, j = 9

11. i = 10, j = 10

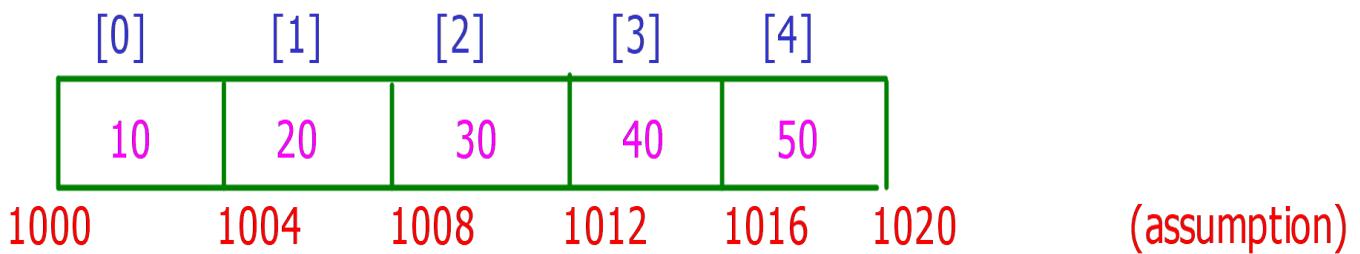
10 < 10

## Arrays

- Array is a collection of the same type of data.
- huge data type which can store more than one value
- used to organize the data

syntax:

```
datatype array_name[SIZE]; e.g,  
int arr[5]; //integer array of size 5  
int arr[5] = {10,20,30,40,50};
```

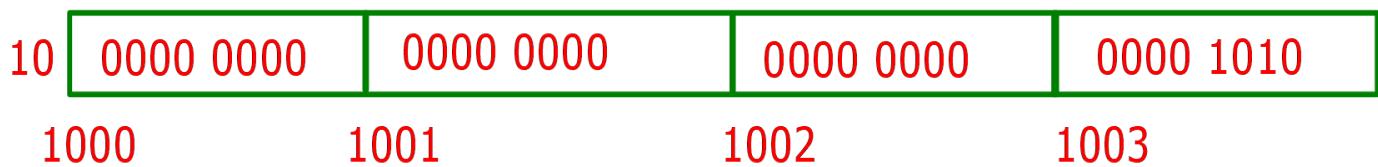


$$\text{Total Memory allocated} = \text{sizeof\_datatype} * \text{sizeof\_array}$$

$$\begin{aligned} &= 4 * 5 \\ &= 20 \end{aligned}$$

$$\begin{array}{lll} \text{arr[0]} ==> 10 & \text{arr[2]} ==> 30 & \text{arr[4]} ==> 50 \\ \text{arr[1]} ==> 20 & \text{arr[3]} ==> 40 & \end{array}$$

how each element is represented?



How to print an array?

- Arrays are printed with the help of loops
- Example:

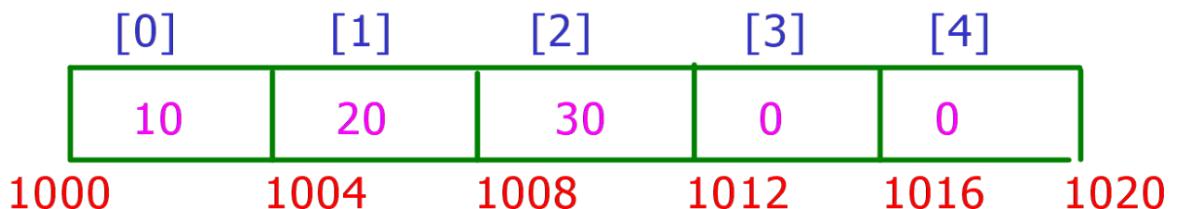
```
for(index = 0; index < 5 ; index++)  
    printf("%d\n",arr[index]);
```

## Different ways of declaring an array

1. `int arr[5] = {10, 20, 30, 40, 50};`

2. Partial initialisation

```
int arr[5] = {10, 20, 30};
```



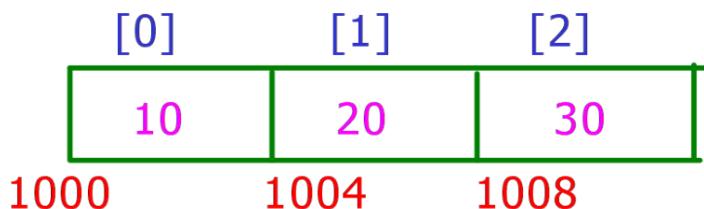
-- rest of the elements will be initialised 0

3. `int arr[] = {10, 20, 30};`

memory allocated ==> sizeof\_datatype \* number\_of\_elements

$$4 * 3$$

12bytes



4. `int arr[];`

// error, invalid

5. `int arr[] = {10, 20, ,30, 40};`

// invalid, error

6. `int arr[5];`

--> by default array elements will have garbage values

## **Reading array from user:**

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

```
for(index=0;index < 5;index++)
{
    printf("%d ",arr[index]);
}
```

# Problem Solving



# Advanced C

## Problem Solving - What?

- An approach which could be taken to reach to a solution
- The approach could be ad hoc or generic with a proper order
- Sometimes it requires a creative and out of the box thinking to reach to perfect solution

# Advanced C

## Problem Solving



- Introduction to SDLC
- Polya's Rules
- Algorithm Design Methods



# Advanced C

## Problem Solving - SDLC - A Quick Introduction



- Never jump to implementation. Why?
  - You might not have the clarity of the application
  - You might have some loose ends in the requirements
  - Complete picture of the application could be missing and many more...



# Advanced C

## Problem Solving - SDLC - A Quick Introduction



Requirement

Design

Code

Test

- Understand the requirement properly
- Consider all the possible cases like inputs and outputs
- Know the boundary conditions
- Get it verified

# Advanced C

## Problem Solving - SDLC - A Quick Introduction



Requirement

Design

Code

Test

- Have a proper design plan
- Use some algorithm for the requirement
  - Use paper and pen method
- Use a flow chart if required
- Make sure all the case are considered

# Advanced C

## Problem Solving - SDLC - A Quick Introduction



Requirement

Design

Code

Test

- Implement the code based on the derived algorithm
- Try to have modular structure where ever possible
- Practice neat implementation habits like
  - Indentation
  - Commenting
  - Good variable and function naming's
  - Neat file and function headers

# Advanced C

## Problem Solving - SDLC - A Quick Introduction



Requirement

Design

Code

Test

- Test the implementation thoroughly
- Capture all possible cases like
  - Negative and Positive case
- Have neat output presentation
- Let the output be as per the user requirement

# Advanced C

## Problem Solving - How?



- Polya's rule
  - Understand the problem
  - Devise a plan
  - Carryout the Plan
  - Look back



# Advanced C

## Problem Solving - Algorithm - What?

- A procedure or formula for solving a problem
- A sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

# Advanced C

## Problem Solving - Algorithm - Need?

- Algorithms is needed to generate correct output in finite time in a given constrained environment
  - Correctness of output
  - Finite time
  - Better Prediction

# Advanced C

Problem Solving - Algorithm - How?

- Natural Language
- Pseudo Codes
- Flowcharts etc.,

# Advanced C

## Problem Solving - Algorithm - Daily Life Example

- Let's consider a problem of reaching this room
- The different possible approach could be thought of
  - Take a Walk
  - Take a Bus
  - Take a Car
  - Let's Pool
- Lets discuss the above approaches in bit detail

# Advanced C

Algorithm - Reaching this Room - Take a Walk



The steps could be like

1. Start at 8 AM
2. Walk through street X for 500 Mts
3. Take a left on main road and walk for 2 KM
4. Take a left again and walk 200 Mts to reach



# Advanced C

## Algorithm - Reaching this Room - Take a Walk



- Pros
  - You might say walking is a good exercise :)
  - Might have good time prediction
  - Save some penny
- Cons
  - Depends on where you stay (you would choose if you stay closer)
  - Should start early
  - Would get tired
  - Freshness would have gone



# Advanced C

Algorithm - Reaching this Room - Take a Bus



The steps could be like

1. Start at 8.30 AM
2. Walk through street X for 500 Mts
3. Take a left on main road and walk for 100 Mts to bus stop
4. Take Bus No 111 and get down at stop X and walk for 100 Mts
5. Take a left again and walk 200 Mts to reach



# Advanced C

Algorithm - Reaching this Room - Take a Bus



- Pros
  - You might save some time
  - Less tiredness comparatively
- Cons
  - Have to walk to the bus stop
  - Have to wait for the right bus (No prediction of time)
  - Might not be comfortable on rush hours



# Advanced C

Algorithm - Reaching this Room - Take a Car



The steps could be like

1. Start at 9 AM
2. Drive through street X for 500 Mts
3. Take a left on main road and drive 2 KM
4. Take a left again and drive 200 Mts to reach



# Advanced C

Algorithm - Reaching this Room - Take a Car



- Pros
  - Proper control of time and most comfortable
  - Less tiresome
- Cons
  - Could have issues on traffic congestions
  - Will be costly



# Advanced C

Algorithm - Reaching this Room - Let's Pool



The steps could be like

1. Start at 8.45 AM
2. Walk through street X for 500 Mts
3. Reach the main road wait for your partner
4. Drive for 2 KM on the main road
5. Take a left again and drive 200 Mts to reach



# Advanced C

Algorithm - Reaching this Room - Let's Pool



- Pros
  - You might save some time
  - Less costly comparatively
- Cons
  - Have to wait for partner to reach
  - Could have issues on traffic congestions



# Advanced C

## Algorithm - Daily Life Example - Conclusion

- All the above solution eventually will lead you to this room
- Every approach some pros and cons
- It would be our duty as a designer to take the best approach for the given problem

# Advanced C

## Algorithm - A Computer Example1



- Let's consider a problem of adding two numbers
- The steps involved :

Start

Read the value of A and B

Add A and B and store in SUM

Display SUM

Stop

- The above 5 steps would eventually will give us the expected result



# Advanced C

## Algorithm - A Computer Example1 - Pseudo Code

- Let's consider a problem of adding two numbers
- The steps involved :

BEGIN

    Read A, B

    SUM = A + B

    Print SUM

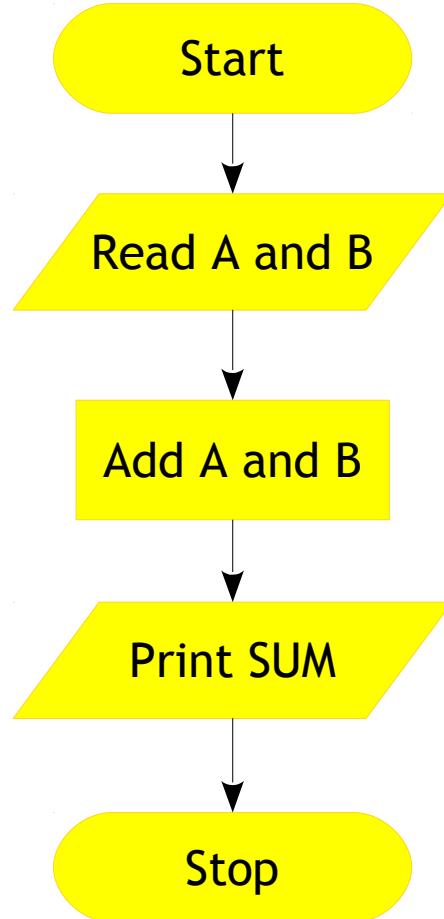
END

- The above 5 steps would eventually will give us the expected result

# Advanced C

## Algorithm - A Computer Example1 - A Flow Chart

- Let's consider a problem of adding two numbers



# Advanced C

## Algorithm - DIY - Pattern

- Write an algorithm to print the below pattern

```
*****  
* *  
* *  
* *  
* *  
* *  
* *  
*****
```

# Advanced C

## Algorithm - DIY - Pattern



- Write an algorithm to print number pyramid

```
1234554321
1234__4321
123____321
12_____21
1_____1
```



# Advanced C

## Algorithm - DIY

- Finding largest of 2 numbers
- Find the largest member of an array

# Advanced C

## Algorithm - Home Work

- Count the number of vowels
- Count the number of occurrences of each vowel
- To find the sum of n - natural numbers
- Convert a number from base 10 to base N

# Introduction to C Programming



# Advanced C



Have you ever pondered how

- powerful it is?
- efficient it is?
- flexible it is?
- deep you can explore your system?

if [ NO ]

Wait!! get some concepts right before you dive into it  
else

You shouldn't be here!!



# Advanced C

Introduction - 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!!



# Advanced C

## Introduction - Language - What?



- A stylized communication technique
- Language has collection of words called “Vocabulary”
  - Rich vocabulary helps us to be more expressive
- Language has finite rules called “Grammar”
  - Grammar helps us to form infinite number of sentences
- The components of grammar :
  - The ***syntax*** governs the structure of sentences
  - The ***semantics*** governs the meanings of words and sentences



# Advanced C

## Introduction - Language - What?



- A stylized communication technique
- It has set of words called “keywords”
- Finite rules (Grammar) to form sentences (often called expressions)
  - Expressions govern the behavior of machine (often a computer)
- Like natural languages, programming languages too have :
  - Syntactic rules (to form expressions)
  - Semantic rules (to govern meaning of expressions)



# Advanced C

## Introduction - Brief History



- Prior to C, most of the computer languages (such as Algol) were academic oriented, unrealistic and were generally defined by committees.
- Since such languages were designed having application domain in mind, they could not take the advantages of the underlying hardware and if done, were not portable or efficient under other systems.
- It was thought that a high-level language could never achieve the efficiency of assembly language

Portable, efficient and easy to use language was a dream.



# Advanced C

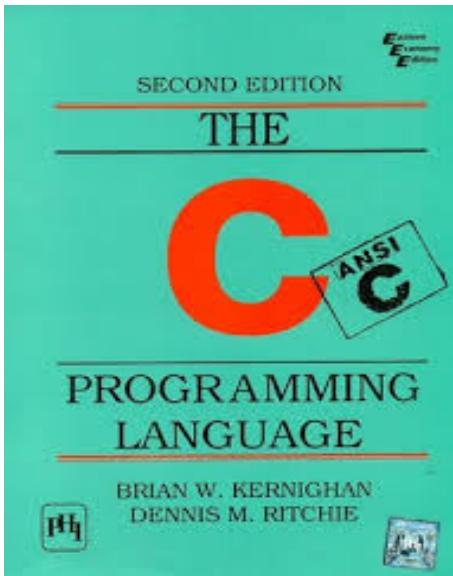
## Introduction - Brief History



- It was a revolutionary language and shook the computer world with its might. With just 32 keywords, C established itself in a very wide base of applications.
- It has lineage starting from CPL, ([Combined Programming Language](#)) a never implemented language
- Martin Richards implemented BCPL as a modified version of CPL. Ken Thompson further refined BCPL to a language named as B
- Later Dennis M. Ritchie added types to B and created a language, what we have as C, for rewriting the UNIX operating system

# Advanced C

## Introduction - 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

# Advanced C

## Introduction - Important Characteristics



- Considered as a middle level language
- Can be considered as a pragmatic language
- It is intended 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 compact code
- It is widely available in various platforms from mainframes to palmtops and is known for its wide availability

# Advanced C

## Introduction - 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

# Advanced C

## Introduction - 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



# Advanced C

## Introduction - Keywords - Categories

| Type       | Keyword                             | Type          | Keyword                                 |
|------------|-------------------------------------|---------------|-----------------------------------------|
| Data Types | char<br>int<br>float<br>double      | Decision      | if<br>else<br>switch<br>case<br>default |
| Modifiers  | signed<br>unsigned<br>short<br>long | Storage Class | auto<br>register<br>static<br>extern    |
| Qualifiers | const<br>volatile                   | Derived       | struct<br>unions                        |
| Loops      | for<br>while<br>do                  | User defined  | enums<br>typedefs                       |
| Jump       | goto<br>break<br>continue           | Others        | void<br>return<br>sizeof                |

# Advanced C

## Introduction - 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

# Advanced C

## Introduction - Anatomy of a Simple C Code



```
/* My first C code */
```

File Header

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

Preprocessor Directive

```
int main()
```

The start of program

```
{
```

```
/* To display Hello world */
```

Comment

```
printf("Hello world\n");
```

Statement

```
return 0;
```

Program Termination

```
}
```



# Advanced C

## Introduction - 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>**



# Advanced C

## Introduction - Execution



- To execute your code you shall try  
`$ ./a.out`
- If you have named your output file as your <file\_name> then  
`$ ./<file_name>`
- This should be the expected result on your system



# Basic Refreshers



# Number System



# Advanced C

## Number Systems

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

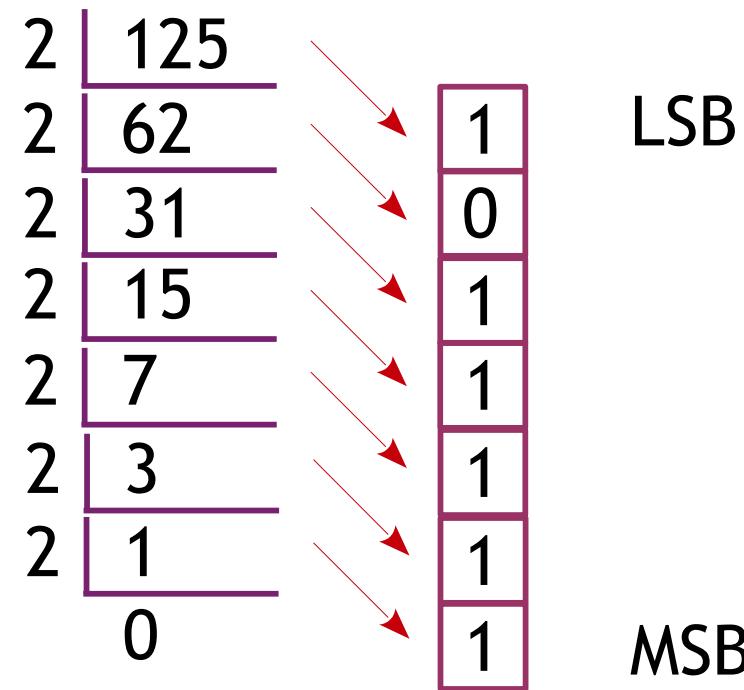
| Type        | Range (8 Bits)          |
|-------------|-------------------------|
| Decimal     | 0 - 255                 |
| Octal       | 000 - 0377              |
| Hexadecimal | 0x00 - 0xFF             |
| Binary      | 0b00000000 - 0b11111111 |

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

# Advanced C

# Number Systems - Decimal to Binary

- $125_{10}$  to Binary

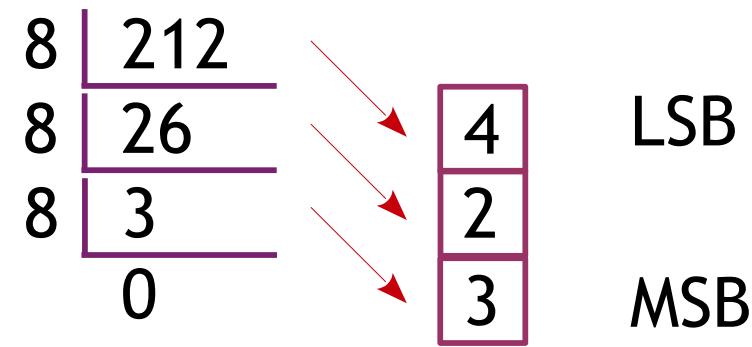


- So  $125_{10}$  is  $1111101_2$

# Advanced C

## Number Systems - Decimal to Octal

- $212_{10}$  to Octal

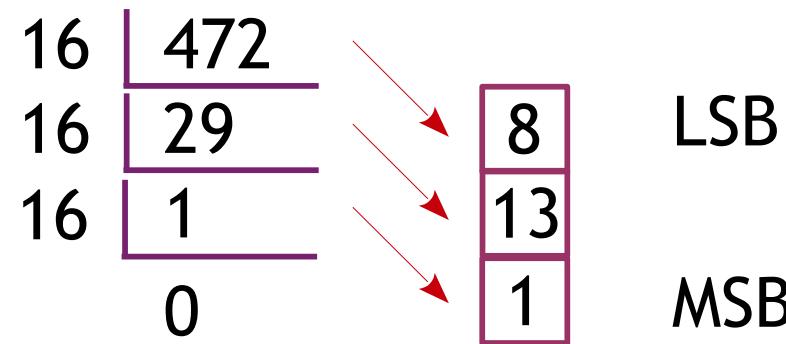


- So  $212_{10}$  is  $324_8$

# Advanced C

## Number Systems - Decimal to Hexadecimal

- $472_{10}$  to Hexadecimal



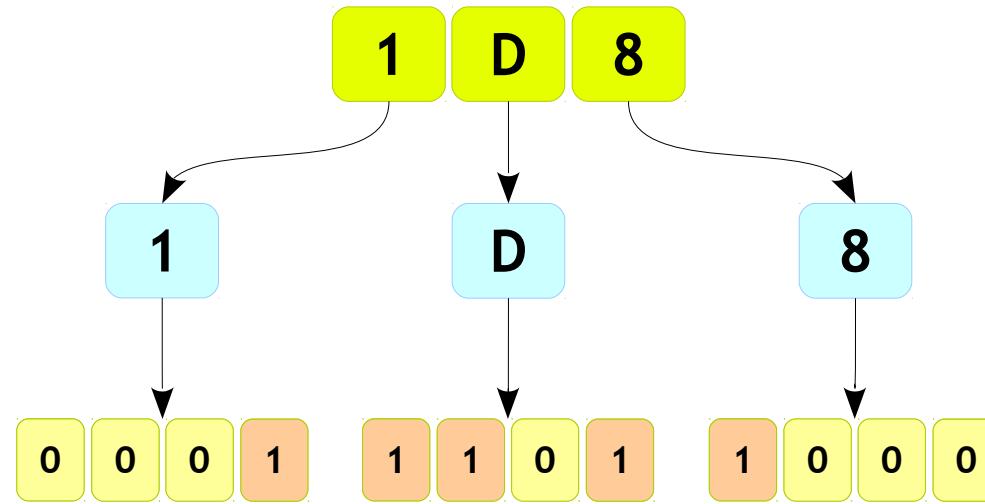
| Representation | Substitutes                           |
|----------------|---------------------------------------|
| Dec            | 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 |
| Hex            | 0 1 2 3 4 5 6 7 8 9 A B C D E F       |

- So  $472_{10}$  is  $1D8_{16}$

# Advanced C

## Number Systems - Hexadecimal to Binary

- $1D8_{16}$  to Binary

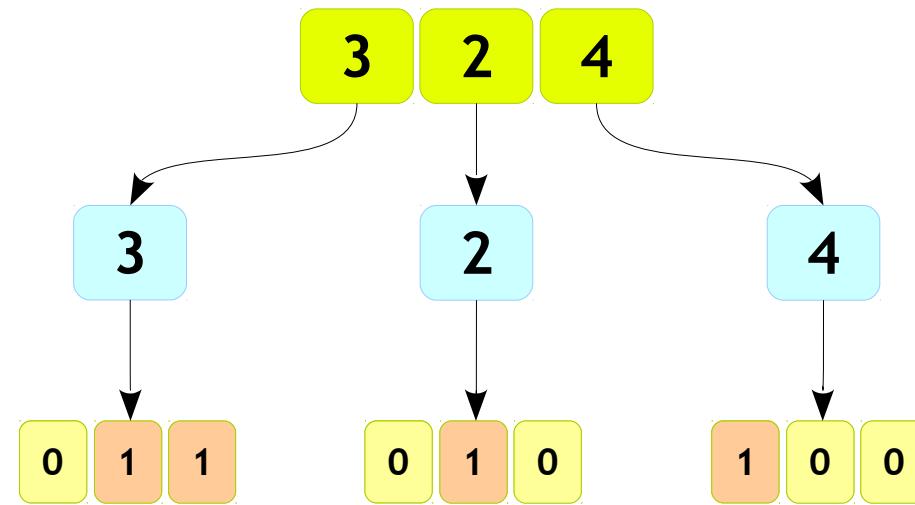


- So  $1D8_{16}$  is  $000111011000_2$  which is nothing but  $111011000_2$

# Advanced C

## Number Systems - Octal to Binary

- $324_8$  to Binary

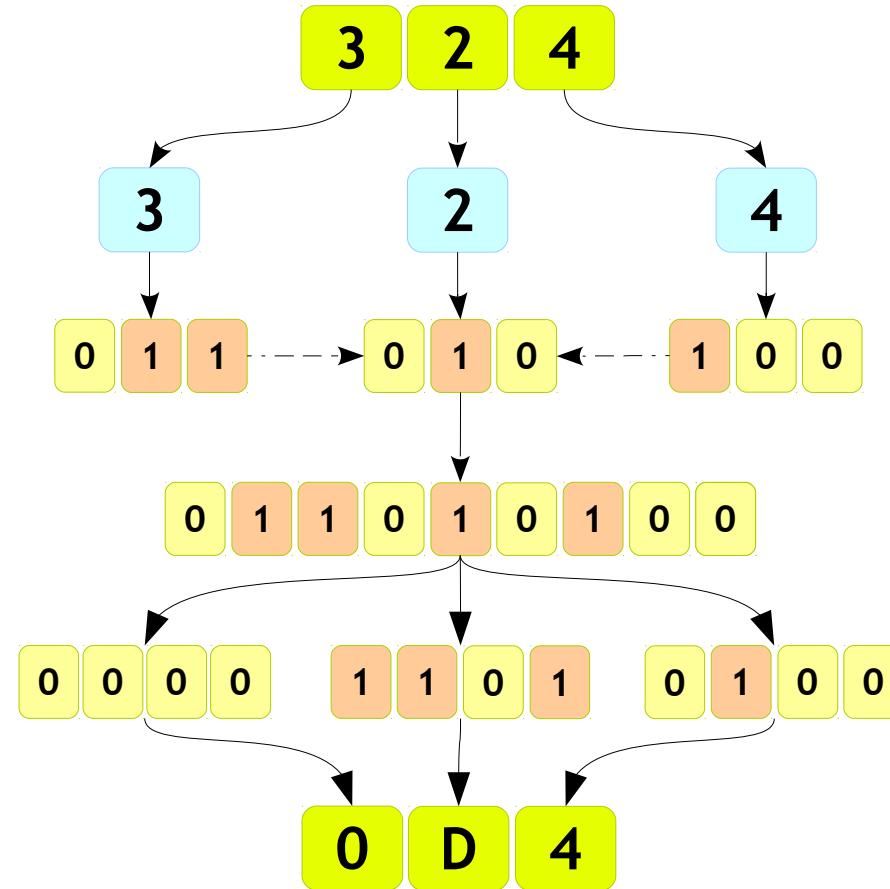


- So  $324_8$  is  $011010100_2$  which is nothing but  $11010100_2$

# Advanced C

## Number Systems - Octal to Hexadecimal

- $324_8$  to Hexadecimal

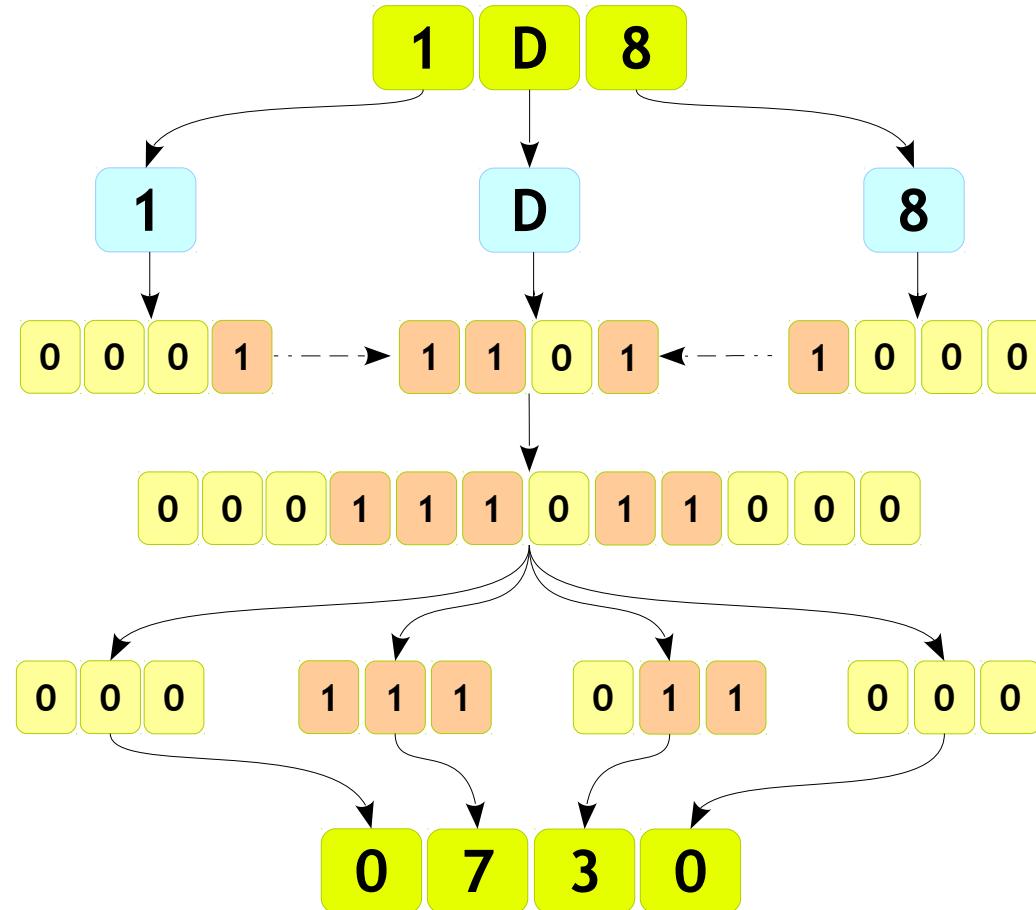


- So  $324_8$  is  $0D4_{16}$  which is nothing but  $D4_{16}$

# Advanced C

## Number Systems - Hexadecimal to Octal

- $1D8_{16}$  to Octal

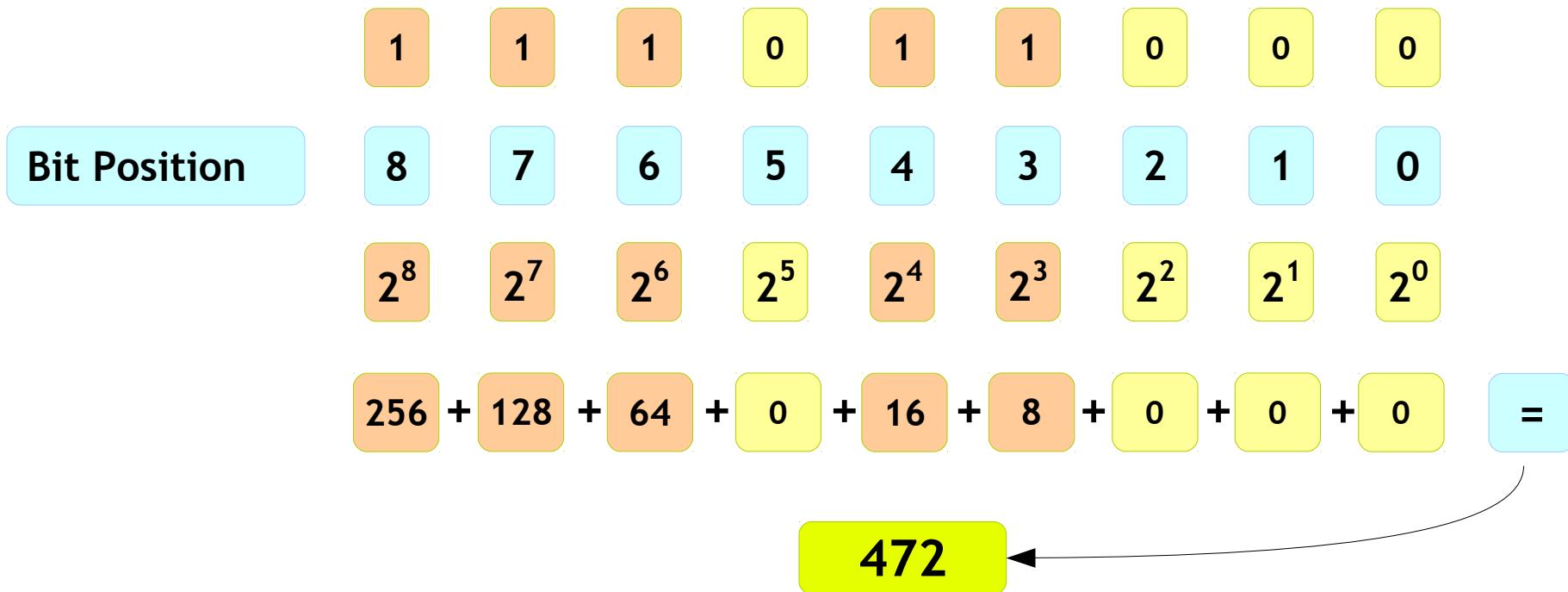


- So  $1D8_{16}$  is  $0730_8$  which is nothing but  $730_8$

# Advanced C

## Number Systems - Binary to Decimal

- $111011000_2$  to Decimal



- So  $111011000_2$  is  $472_{10}$

# Data Representations



# Advanced 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          |

# Advanced 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 |

# Advanced C

## Data Representation - Character

- One byte represents one unique character like 'A', 'b', '1', '\$' ...
- It's 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

# Advanced 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 |

# Advanced 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 |

# Advanced 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 0 0 1 1 0 1                   |

# Advanced 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 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 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 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 1 1 0 0 1 1                   |

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

# Advanced 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 |
|----------------------------------------------|-------|----------|----------|
| <b>Float (32 bits)<br/>Single Precision</b>  | 1 bit | 8 bits   | 23 bits  |
| <b>Double (64 bits)<br/>Double Precision</b> | 1 bit | 11 bits  | 52 bits  |

# Advanced C

## Float Point Number - Conversion Procedure



- **STEP 1:** Convert the absolute value of the number to binary, perhaps with a fractional part after the binary point. This can be done by -
  - Converting the integral part into binary format.
  - Converting the fractional part into binary format.

The integral part is converted with the techniques examined previously.

The fractional part can be converted by multiplying it with 2.

- **STEP 2:** Normalize the number. Move the binary point so that it is one bit from the left. Adjust the exponent of two so that the value does not change.

$$\text{Float} : V = (-1)^s * 2^{(E-127)} * 1.F$$

$$\text{Double} : V = (-1)^s * 2^{(E-1023)} * 1.F$$



# Advanced C

# Float Point Number - Conversion - Example 1

# Convert 2.5 to IEEE 32-bit floating point format

## Step 1:

## Step 2:

$$0.5 \times 2 = 1.0$$

$$2.5_{10} = 10.1_2$$

Normalize:  $10.1_2 = 1.01_2 \times 2^1$

Mantissa is 010000000000000000000000  
Exponent is  $1 + 127 = 128 = 1000\ 0000_2$   
Sign bit is 0



# Advanced C

# Float Point Number - Conversion - Example 2

## Convert 0.625 to IEEE 32-bit floating point format

## Step 1:

|       |            |      |   |
|-------|------------|------|---|
| 0.625 | $\times 2$ | 1.25 | 1 |
| 0.25  | $\times 2$ | 0.5  | 0 |
| 0.5   | $\times 2$ | 1.0  | 1 |

$$0.625_{10} = 0.101_2$$

## Step 2:

**Normalize:**  $0.101_2 = 1.01_2 \times 2^{-1}$

Mantissa is 010000000000000000000000  
Exponent is  $-1 + 127 = 126 = 01111110_2$   
Sign bit is 0



# Advanced C

## Float Point Number - Conversion - Example 3

Convert 39887.5625 to IEEE 32-bit floating point format

Step 1:

|        |              |       |   |
|--------|--------------|-------|---|
| 0.5625 | $\times 2 =$ | 1.125 | 1 |
| 0.125  | $\times 2 =$ | 0.25  | 0 |
| 0.25   | $\times 2 =$ | 0.5   | 0 |
| 0.5    | $\times 2 =$ | 1.0   | 1 |

$$39887.5625_{10} = \\ 1001101111001111.1001_2$$

Step 2:

Normalize:

$$1001101111001111.1001_2 = \\ 1.001101111001111001_2 \times 2^{15}$$

Mantissa is 00110111100111110010000  
Exponent is  $15 + 127 = 142 = 10001110_2$   
Sign bit is 0

| Bit      | S  | Exponent             | Mantissa                                                      |
|----------|----|----------------------|---------------------------------------------------------------|
| 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  | 1 0 0 0 1 1 1        | 0 0 0 1 1 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0 0             |

# Advanced C

# Float Point Number - Conversion - Example 5

## Convert -13.3125 to IEEE 32-bit floating point format

## Step 1:

|        |              |       |   |
|--------|--------------|-------|---|
| 0.3125 | $\times 2 =$ | 0.625 | 0 |
| 0.625  | $\times 2 =$ | 1.25  | 1 |
| 0.25   | $\times 2 =$ | 0.5   | 0 |
| 0.5    | $\times 2 =$ | 1.0   | 1 |

$$13.3125_{10} = 1101.0101_2$$

## Step 2:

## Normalize:

$$1101.0101_2 = 1.1010101_2 \times 2^3$$

**Mantissa is 101010100000000000000000**  
**Exponent is  $3 + 127 = 130 = 1000\ 0010_2$**   
**Sign bit is 1**

# Advanced C

## Float Point Number - Conversion - Example 6

Convert 1.7 to IEEE 32-bit floating point format

Step 1:

|     |              |     |   |
|-----|--------------|-----|---|
| 0.7 | $\times 2 =$ | 1.4 | 1 |
| 0.4 | $\times 2 =$ | 0.8 | 0 |
| 0.8 | $\times 2 =$ | 1.6 | 1 |
| 0.6 | $\times 2 =$ | 1.2 | 1 |
| 0.2 | $\times 2 =$ | 0.4 | 0 |
| 0.4 | $\times 2 =$ | 0.8 | 0 |
| 0.8 | $\times 2 =$ | 1.6 | 1 |
| 0.6 | $\times 2 =$ | 1.2 | 1 |

Step 2:

Normalize:

$$1.10110011001100110011001_2 =$$

$$1.10110011001100110011001_2 \times 2^0$$

Mantissa is 10110011001100110011001

Exponent is  $0 + 127 = 127 = 0111111_2$

Sign bit is 1

$$1.7_{10} = 1.10110011001100110011001_2$$

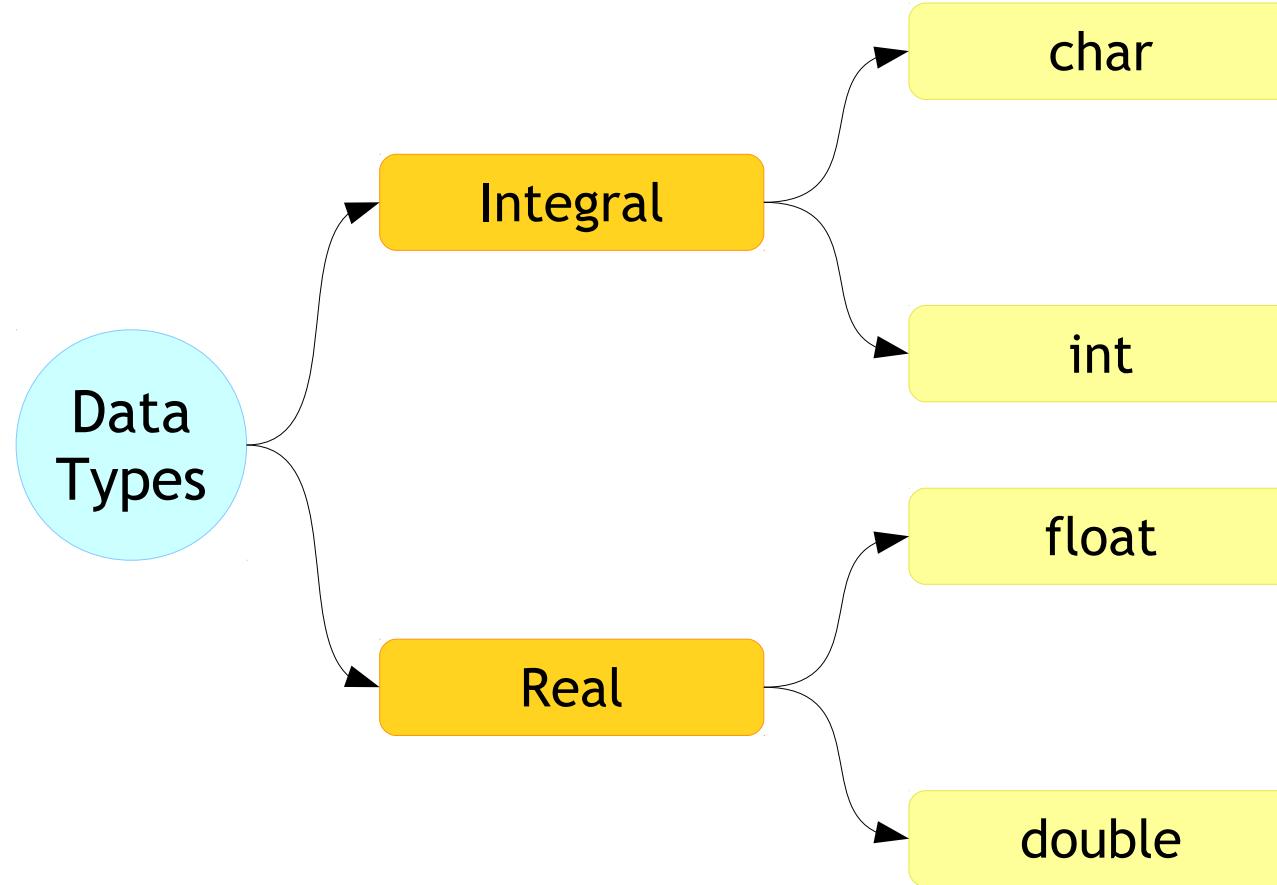
| Bit      | S  | Exponent                | Mantissa                                                   |
|----------|----|-------------------------|------------------------------------------------------------|
| Position | 31 | 30 29 28 27 26 25 24 23 | 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| Value    | 0  | 0 1 1 1 1 1 1 1         | 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1    |

# Data Types



# Advanced C

## Data Types - Categories



# Advanced C

## Data Types - Usage



### Syntax

```
data_type name_of_the_variable;
```

### Example

```
char option;
int age;
float height;
```



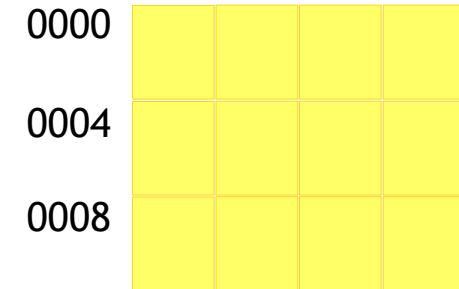
# Advanced C

## Data Types - Storage

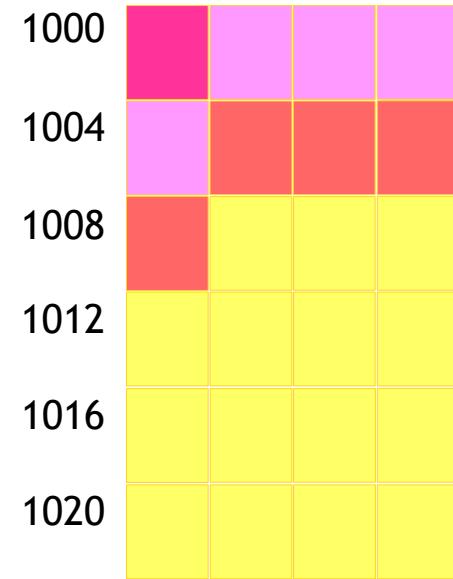


### Example

```
char option;  
int age;  
float height;
```



•  
•  
•



# Advanced C

## Data Types - Printing

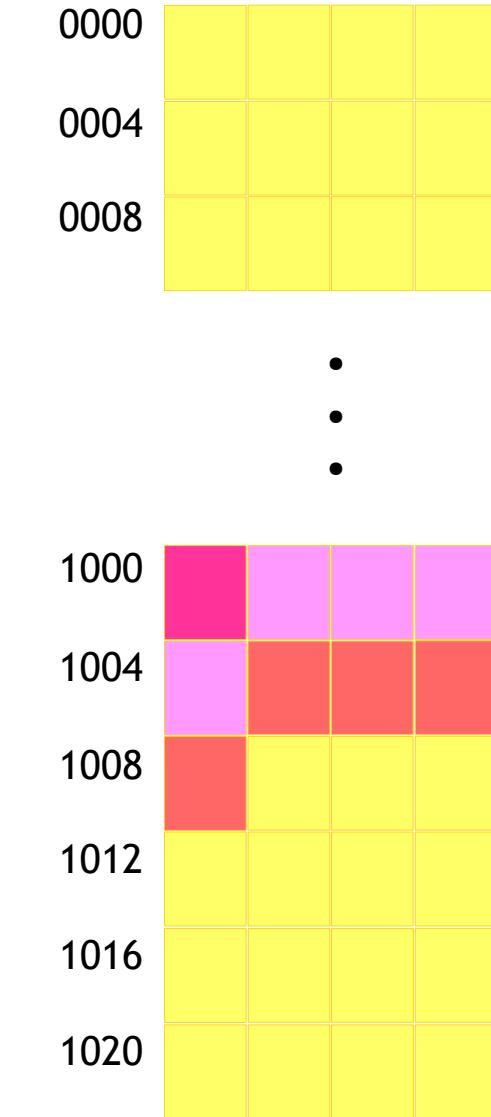
001\_example.c

```
#include <stdio.h>

int main()
{
    char option;
    int age;
    float height;

    printf("The character is %c\n", option);
    printf("The integer is %d\n", age);
    printf("The float is %f\n", height);

    return 0;
}
```



# Advanced C

## Data Types - Scaning

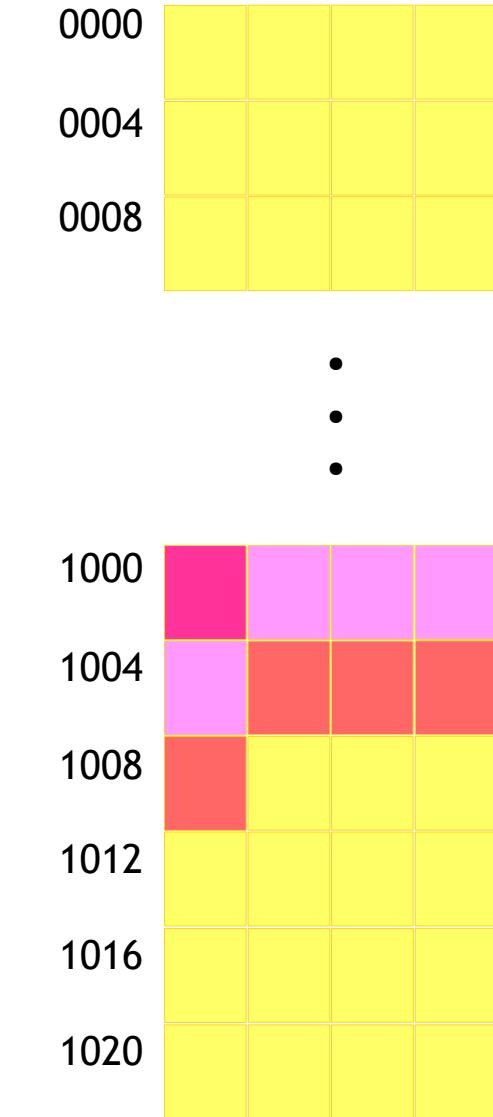
002\_example.c

```
#include <stdio.h>

int main()
{
    char option;
    int age;
    float height;

    scanf("%c", &option);
    printf("The character is %c\n", option);
    scanf("%d", &age);
    printf("The integer is %d\n", age);
    scanf("%f", &height);
    printf("The float is %f\n", height);

    return 0;
}
```



# Advanced C

## Data Types - Size

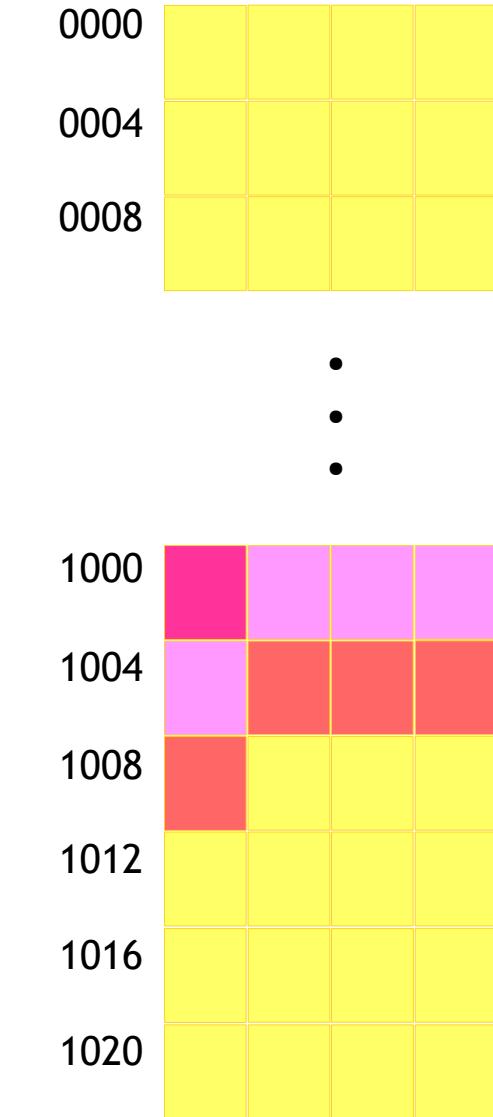
### 003\_example.c

```
#include <stdio.h>

int main()
{
    char option;
    int age;
    float height;

    printf("The size of char is %u\n", sizeof(char));
    printf("The size of int is %u\n", sizeof(int));
    printf("The float is %u\n", sizeof(float));

    return 0;
}
```



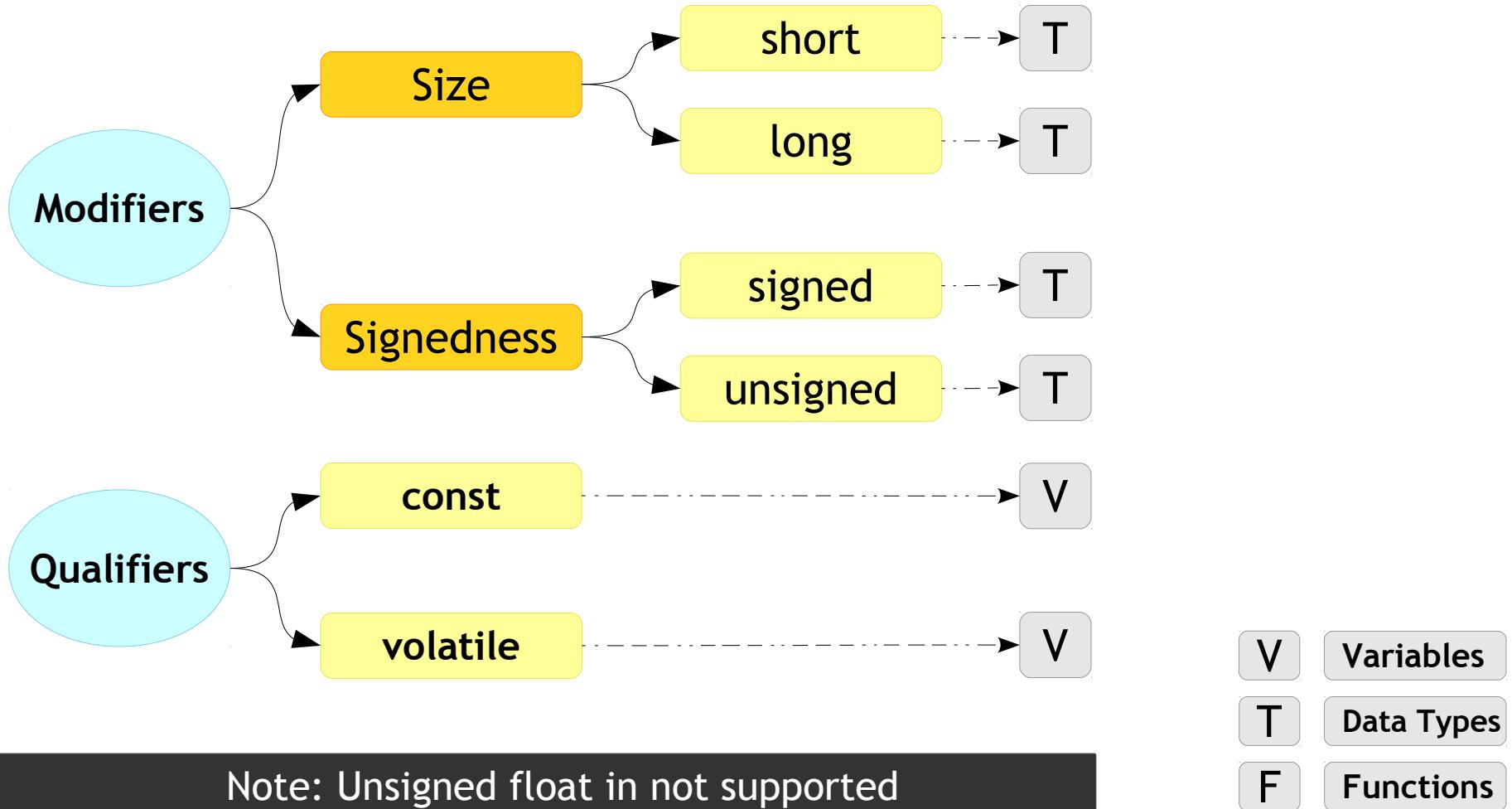
# Advanced C

## Data Types - Modifiers and Qualifiers

- These are some keywords which is used to tune the property of the data type like
  - Its width
  - Its sign
  - Its storage location in memory
  - Its access property
- The K & R C book mentions only two types of qualifiers (refer the next slide). The rest are sometimes interchangably called as specifiers and modifiers and some people even call all as qualifiers!

# Advanced C

## Data Types - Modifiers and Qualifiers



Note: Unsigned float is not supported

# Advanced C

## Data Types - Modifiers and Qualifiers - Usage



### Syntax

```
<modifier / qualifier> <data_type> name_of_the_variable;
```

### Example

```
short int count1;
long int count2;
const int flag;
```



# Advanced C

## Data Types - Modifiers - Size

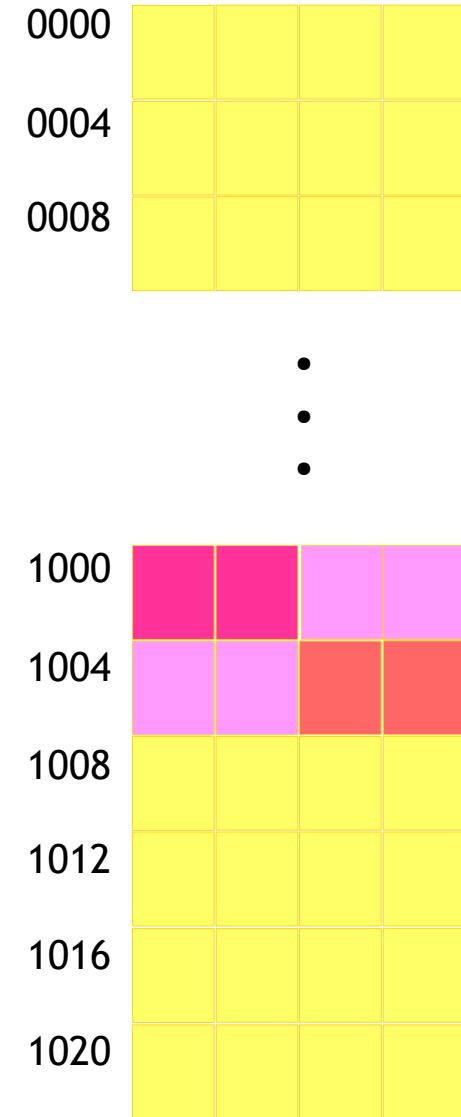
### 004\_example.c

```
#include <stdio.h>

int main()
{
    short int count1;
    int long count2;
    short count3;

    printf("short is %u bytes\n", sizeof(short int));
    printf("long int is %u bytes\n", sizeof(int long));
    printf("short is %u bytes\n", sizeof(short));

    return 0;
}
```



# Advanced C

## Data Types - Modifiers - Sign

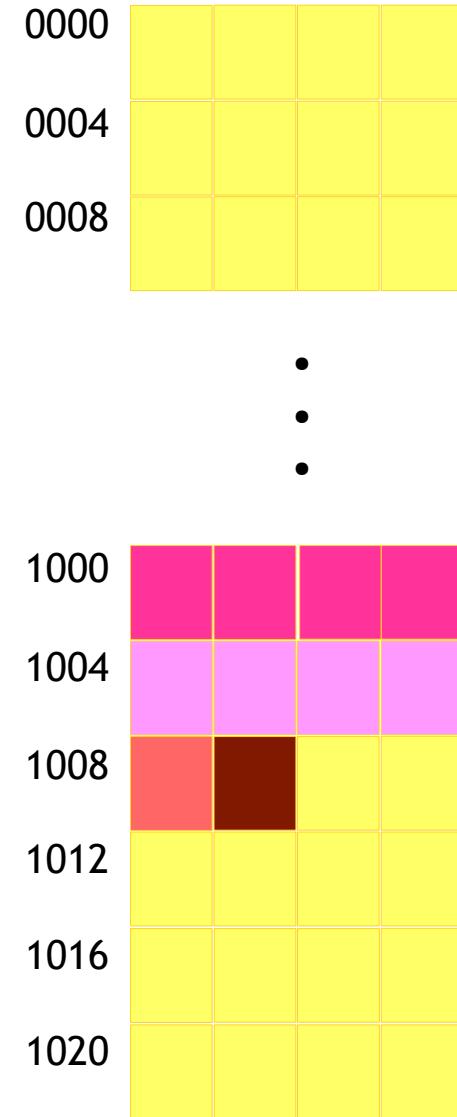
### 005\_example.c

```
#include <stdio.h>

int main()
{
    unsigned int count1;
    signed int count2;
    unsigned char count3;
    signed char count4;

    printf("count1 is %u bytes\n", sizeof(unsigned int));
    printf("count2 is %u bytes\n", sizeof(signed int));
    printf("count3 is %u bytes\n", sizeof(unsigned char));
    printf("count3 is %u bytes\n", sizeof(signed char));

    return 0;
}
```



# Advanced C

## Data Types - Modifiers - Sign and Size

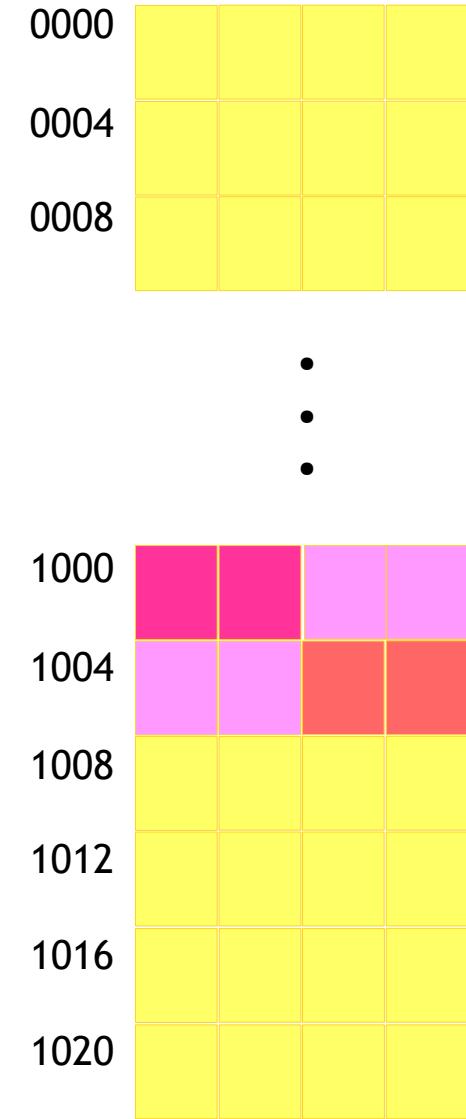
006\_example.c

```
#include <stdio.h>

int main()
{
    unsigned short count1;
    signed long count2;
    short signed count3;

    printf("count1 is %u bytes\n", sizeof(count1));
    printf("count2 is %u bytes\n", sizeof(count2));
    printf("count3 is %u bytes\n", sizeof(count3));

    return 0;
}
```



# Advanced C

## Data Types - Modifiers - Sign and Size

### 007\_example.c

```
#include <stdio.h>

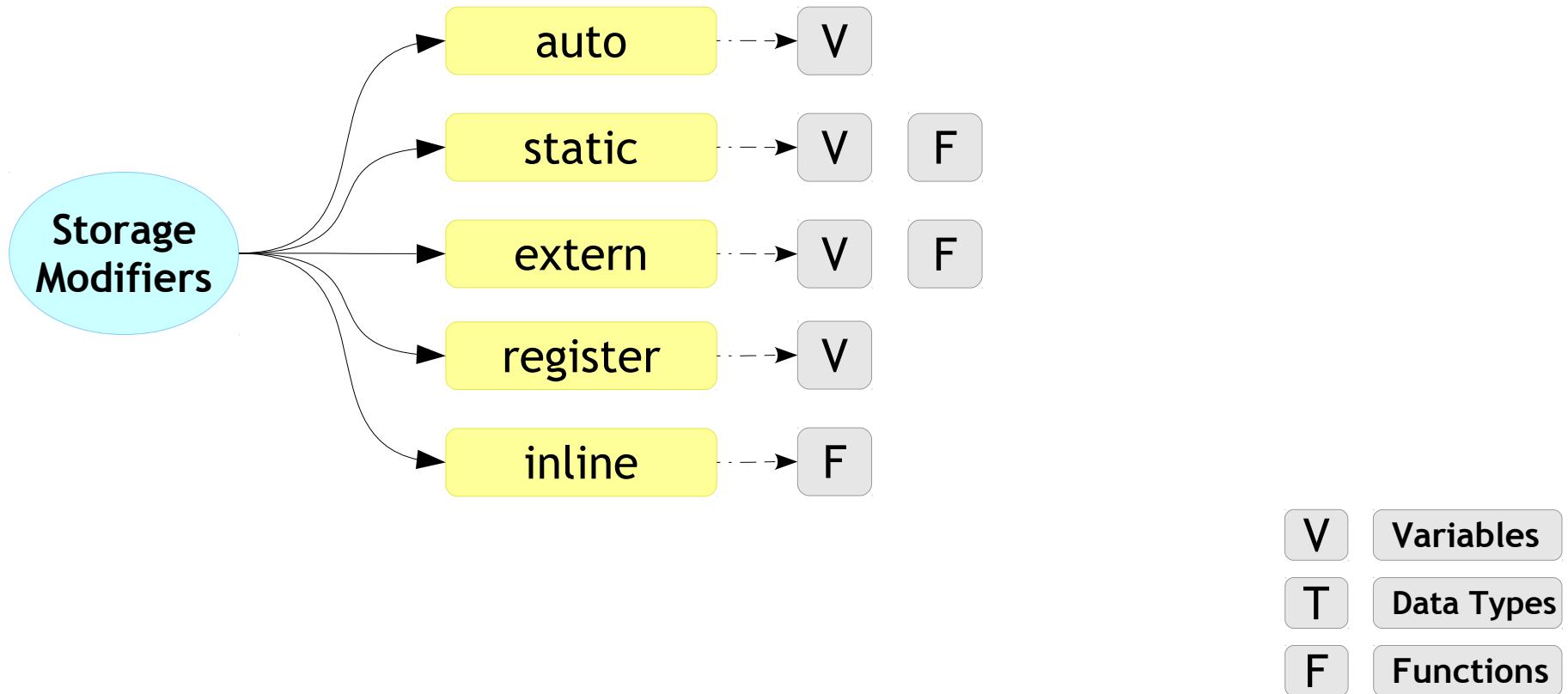
int main()
{
    unsigned int count1 = 10;
    signed int count2 = -1;

    if (count1 > count2)
    {
        printf("Yes\n");
    }
    else
    {
        printf("No\n");
    }

    return 0;
}
```

# Advanced C

## Data Types and Function - Storage Modifiers



# Statements



# Advanced 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!!

# Advanced C

## Code Statements - Compound

```
int main()
{
    ...
    if (num1 > num2)           ← - - - - - If conditional statement
    {
        if (num1 > num3)       ← - - - - - Nested if statement
        {
            printf("Hello");
        }
        else
        {
            printf("World");
        }
    }
    ...
}
```

If conditional statement

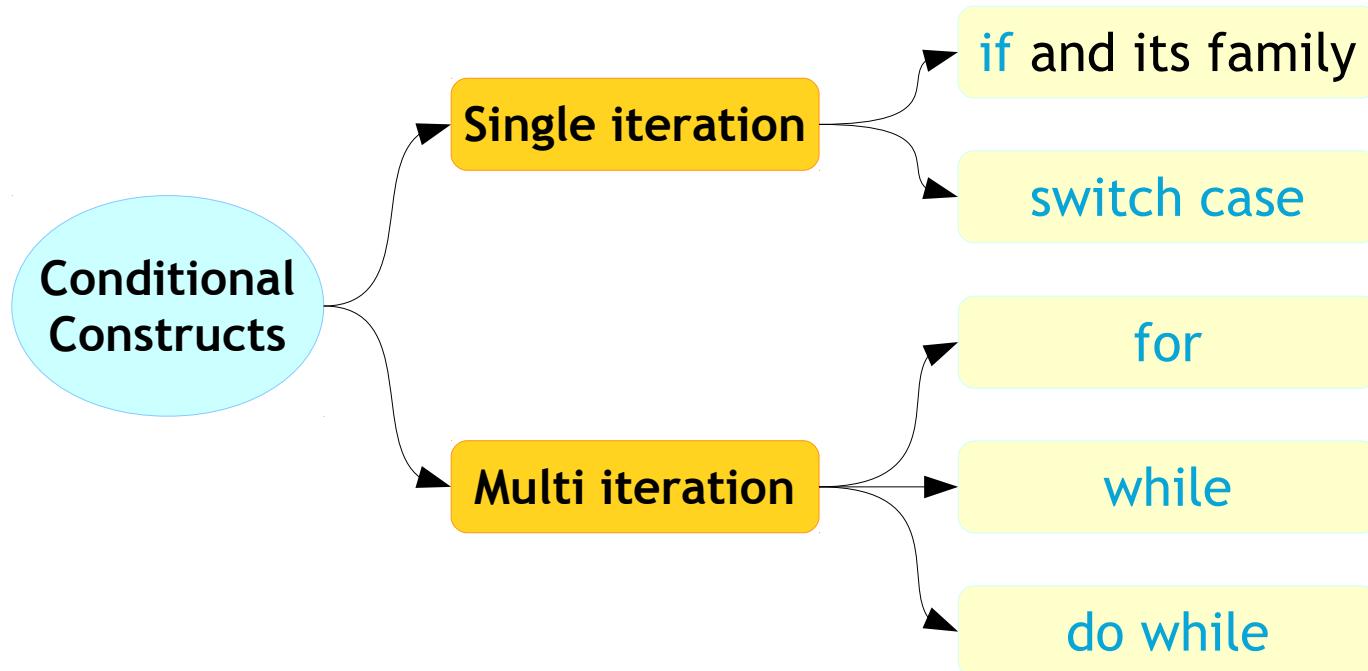
Nested if statement

# Conditional Constructs



# Advanced C

## Conditional Constructs



# Advanced C

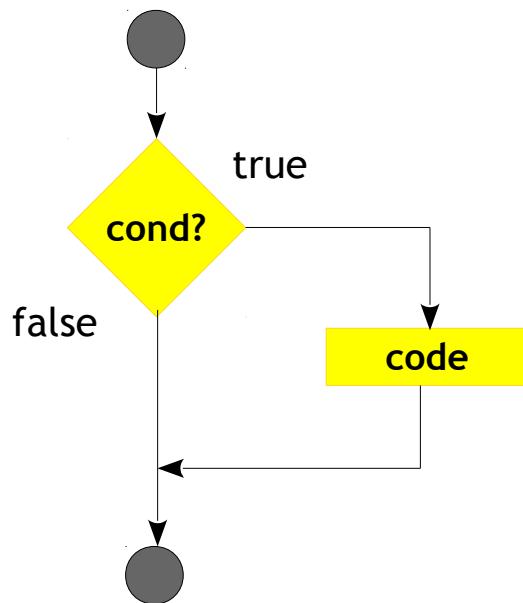
## Conditional Constructs - if



### Syntax

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

### Flow



### 008\_example.c

```
#include <stdio.h>

int main()
{
    int num = 2;

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

    return 0;
}
```

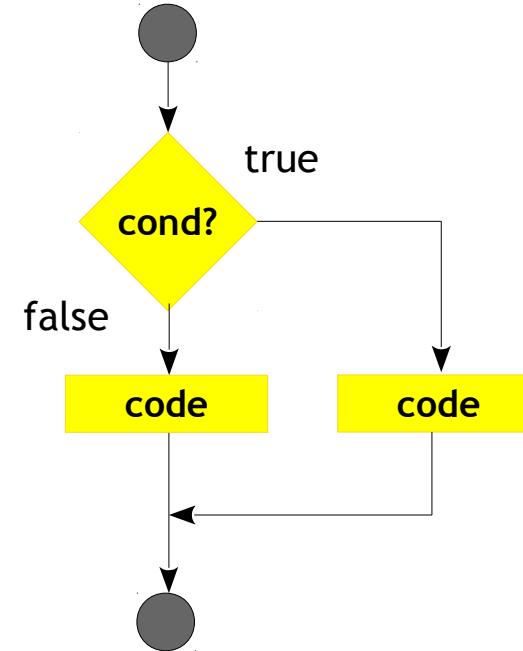
# Advanced C

## Conditional Constructs - if else

### Syntax

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

### Flow



# Advanced C

## Conditional Constructs - if else

### 009\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;

    if (num < 5)
    {
        printf("num is smaller than 5\n");
    }
    else
    {
        printf("num is greater than 5\n");
    }

    return 0;
}
```

# Advanced C

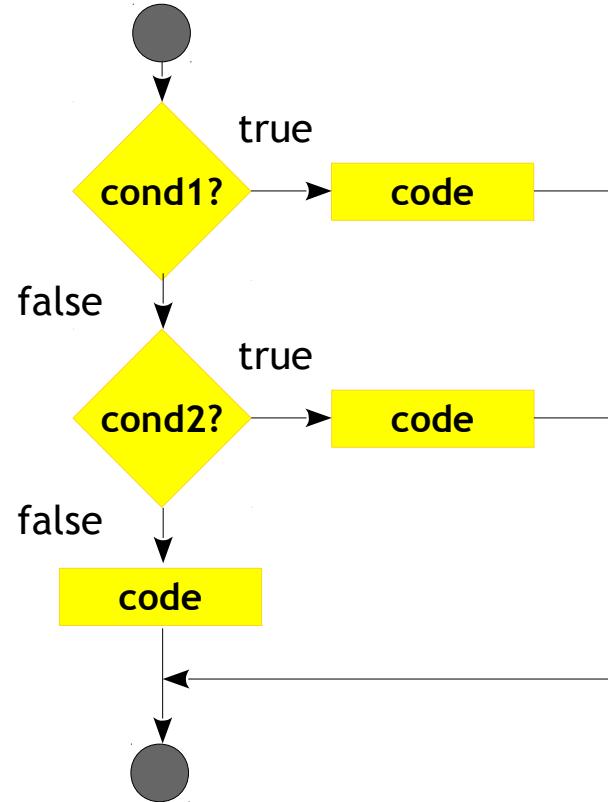
## Conditional Constructs - if else if



### Syntax

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

### Flow



# Advanced C

## Conditional Constructs - if else if



### 009\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;

    if (num < 5)
    {
        printf("num is smaller than 5\n");
    }
    else if (num > 5)
    {
        printf("num is greater than 5\n");
    }
    else
    {
        printf("num is equal to 5\n");
    }

    return 0;
}
```



# Advanced C

## Conditional Constructs - Exercise



- WAP to find the max of two numbers
- WAP to print the grade for a given percentage
- WAP to find the greatest of given 3 numbers
- WAP to check whether character is
  - Upper case
  - Lower case
  - Digit
  - No of the above
- WAP to find the middle number (by value) of given 3 numbers



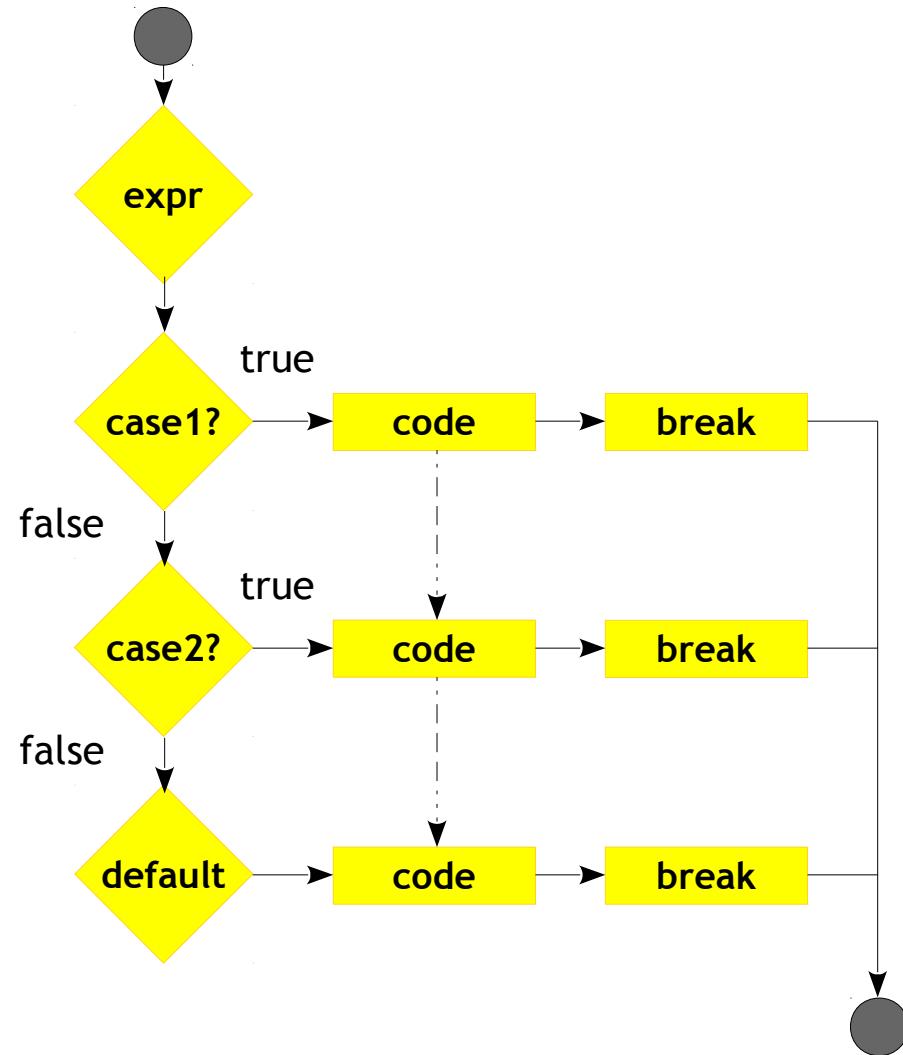
# Advanced C

## 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



# Advanced C

## Conditional Constructs - switch

### 010\_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;
}
```

# Advanced C

## Conditional Constructs - switch - DIY



- W.A.P to check whether character is
  - Upper case
  - Lower case
  - Digit
  - None of the above
- W.A.P for simple calculator



# Advanced C

## Conditional Constructs - while



### Syntax

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

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

### 011\_example.c

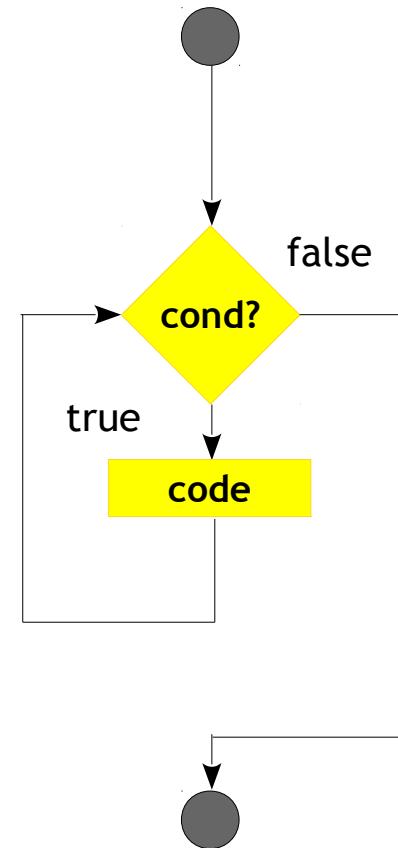
```
#include <stdio.h>

int main()
{
    int iter;

    iter = 0;
    while (iter < 5)
    {
        printf("Looped %d times\n", iter);
        iter++;
    }

    return 0;
}
```

### Flow



# Advanced C

## Conditional Constructs - do while

### Syntax

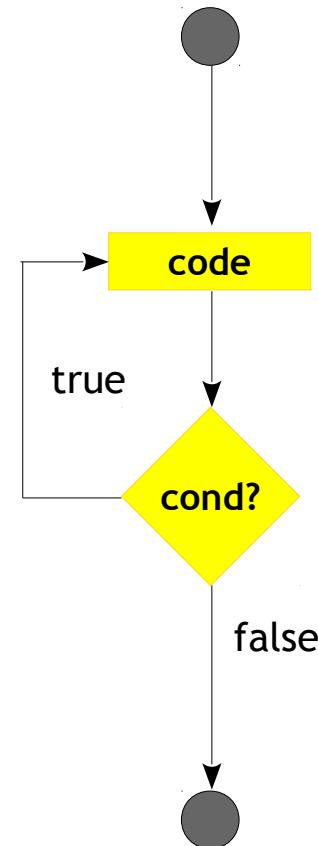
```
do  
{  
    statement(s);  
} while (condition);
```

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

### 012\_example.c

```
#include <stdio.h>  
  
int main()  
{  
    int iter;  
  
    iter = 0;  
    do  
    {  
        printf("Looped %d times\n", iter);  
        iter++;  
    } while (iter < 10);  
  
    return 0;  
}
```

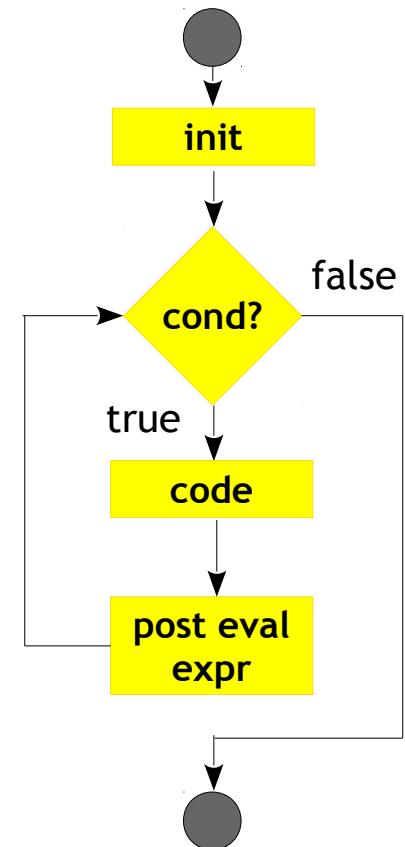
### Flow



# Advanced C

## Conditional Constructs - for

### Flow



### Syntax

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

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

### 013\_example.c

```
#include <stdio.h>

int main()
{
    int iter;

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

    return 0;
}
```

# Advanced C

## Conditional Constructs - Classwork



- W.A.P to print the power of two series using for loop
  - $2^1, 2^2, 2^3, 2^4, 2^5 \dots$
- W.A.P to print the power of N series using Loops
  - $N^1, N^2, N^3, N^4, N^5 \dots$
- W.A.P to multiply 2 nos without multiplication operator
- W.A.P to check whether a number is palindrome or not



# Advanced C

## Conditional Constructs - for - DIY



- WAP to print line pattern
  - Read total (n) number of pattern chars in a line (number should be “odd”)
  - Read number (m) of pattern char to be printed in the middle of line (“odd” number)
  - Print the line with two different pattern chars
  - Example - Let's say two types of pattern chars '\$' and '\*' to be printed in a line. Total number of chars to be printed in a line are 9. Three '\*' to be printed in middle of line.
    - Output ==> \$\$\$\* \* \*\$\$\$

# Advanced C

## Conditional Constructs - for - DIY

- Based on previous example print following pyramid

```
*  
* * *  
* * * * *  
* * * * * * *
```

# Advanced C

## Conditional Constructs - for - DIY



- Print rhombus using for loops

```
*  
* * *  
* * * * *  
* * * * * * *  
* * * * *  
* * *  
*
```



# Advanced C

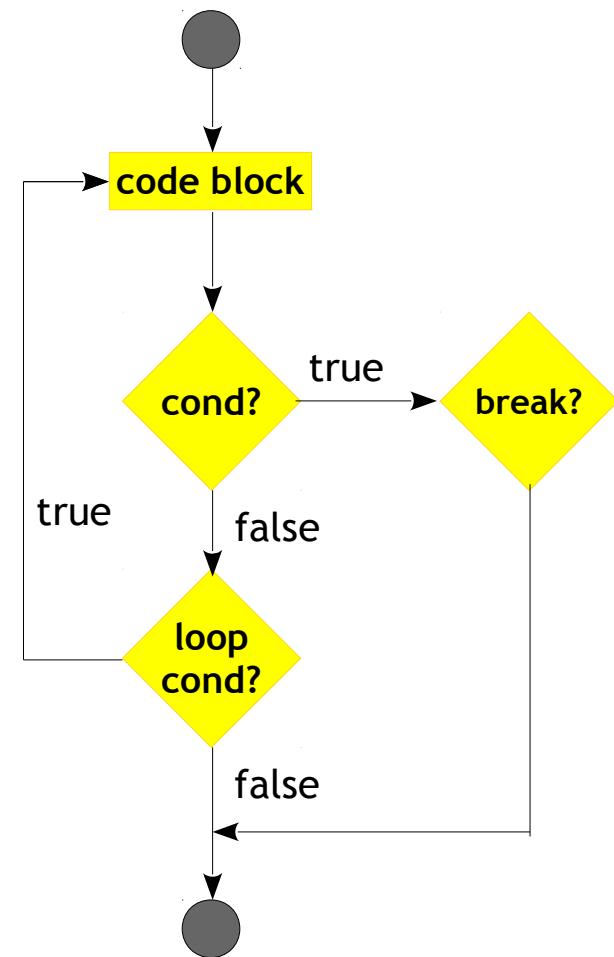
## Conditional Constructs - break

- 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



# Advanced C

## Conditional Constructs - break



### 014\_example.c

```
#include <stdio.h>

int main()
{
    int iter;

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

    return 0;
}
```



# Advanced C

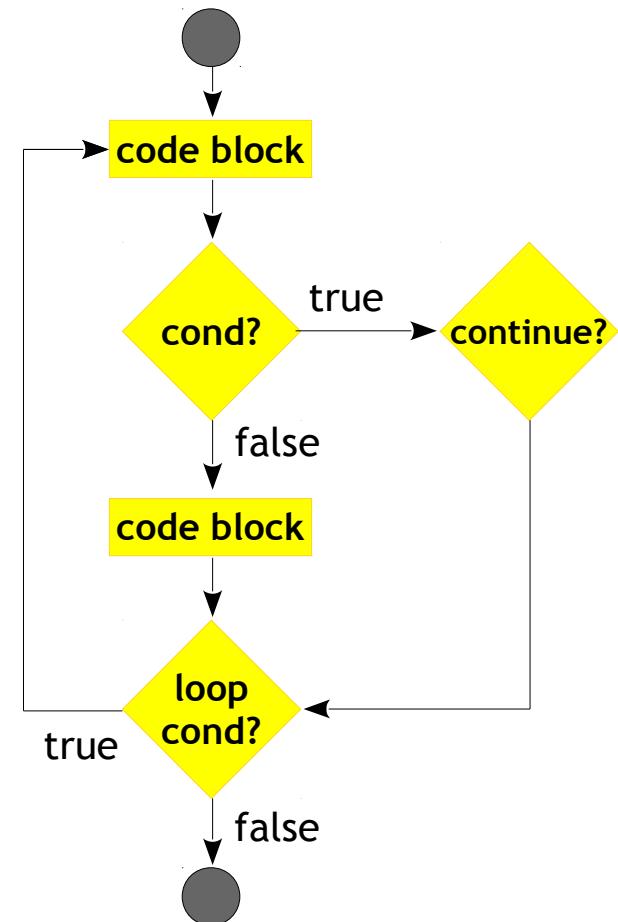
## Conditional Constructs - continue

- 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



# Advanced C

## Conditional Constructs - continue



### 015\_example.c

```
#include <stdio.h>

int main()
{
    int iter;

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

    return 0;
}
```



# Advanced C

## Conditional Constructs - goto



- A *goto* statement causes a unconditional jump to a labeled statement
- Generally avoided in general programming, since it sometimes becomes tough to trace the flow of the code

### Syntax

```
goto label;
```

```
...
```

```
...
```

```
label:
```

```
...
```

# Operators

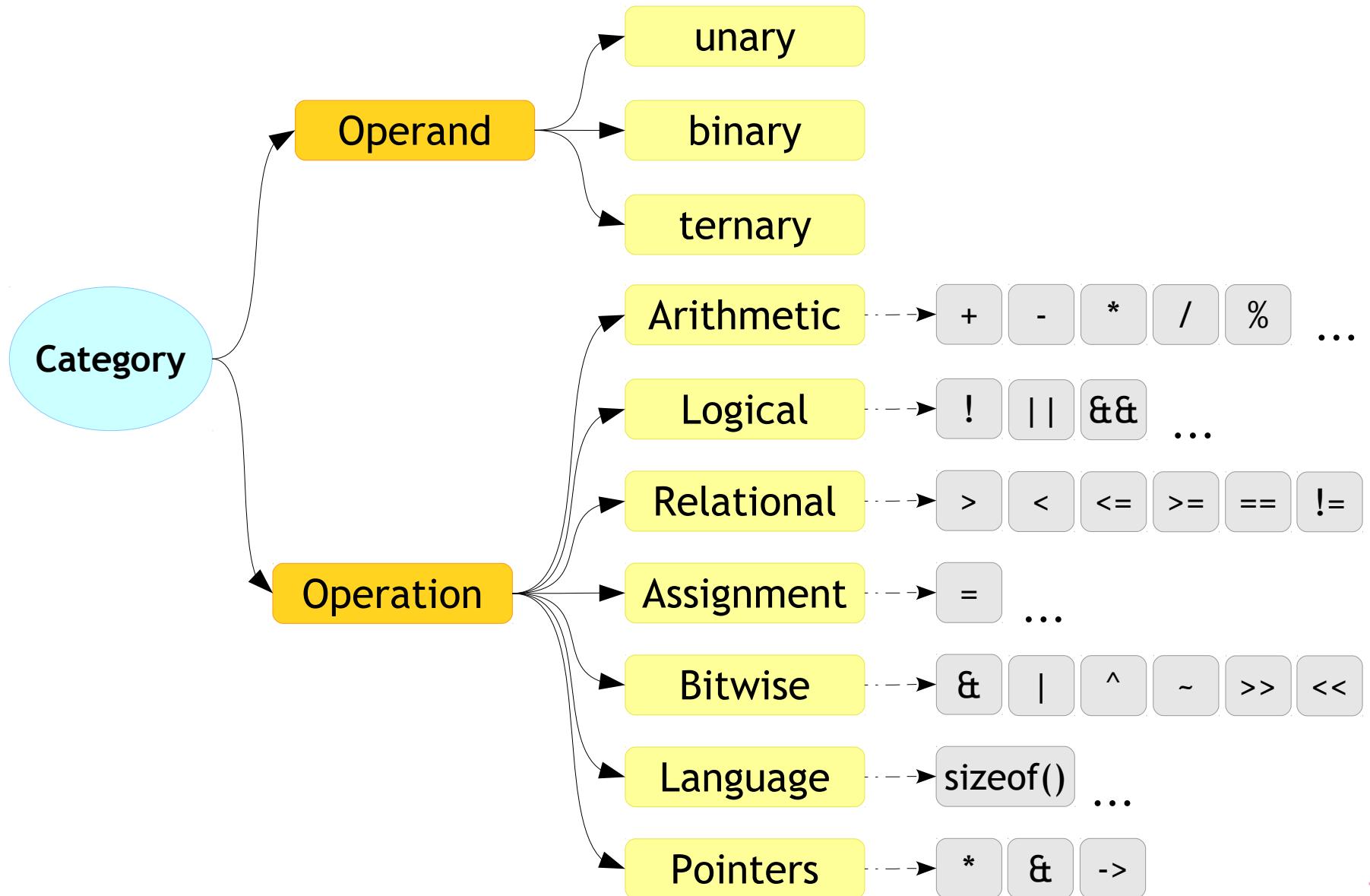


# Advanced C Operators



- 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

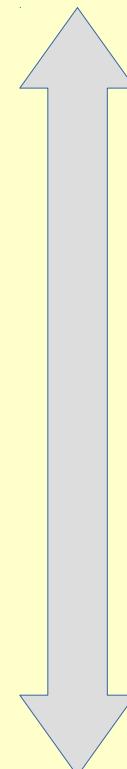
# Advanced C Operators



# Advanced 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)

# Advanced C

## Operators - Arithmetic



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

### 016\_example.c

```
#include <stdio.h>

int main()
{
    int num;

    num = 7 - 4 * 3 / 2 + 5;

    printf("Result is %d\n", num);

    return 0;
}
```

What will be  
the output?



# Advanced C

## Operators - Language - sizeof()

### 017\_example.c

```
#include <stdio.h>

int main()
{
    int num = 5;

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

    return 0;
}
```

### 018\_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;
}
```

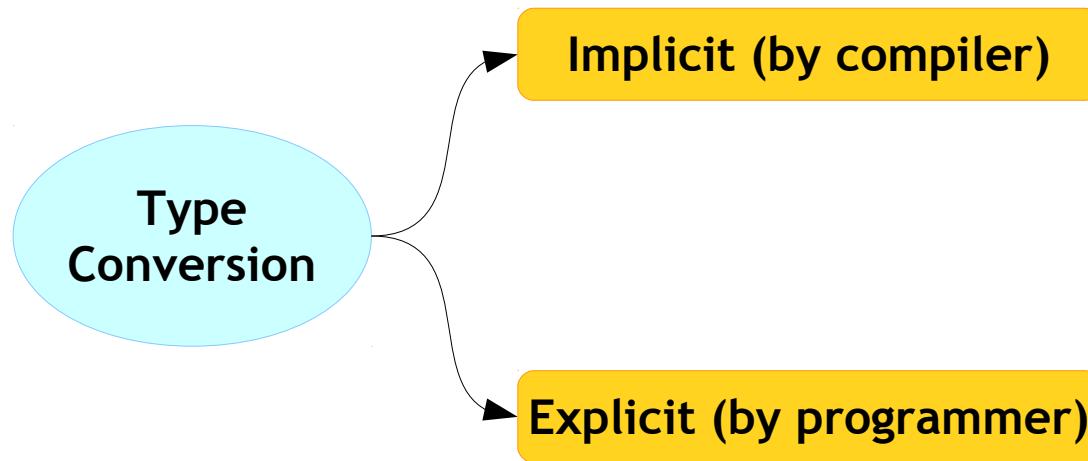
# Advanced C

## 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

# Advanced C

## Type Conversion



# Advanced C

## Type Conversion Hierarchy



# Advanced 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

# Advanced 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

# Advanced C

## Type Conversion - Explicit (Type Casting)



### Syntax

```
(data_type) expression
```

### 019\_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;
}
```

# Advanced C

## Operators - Logical



020\_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;
}
```

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

What will be  
the output?

# Advanced C

## Operators - Circuit Logical



- Have the ability to “short circuit” a calculation if the result is definitely known, this can improve efficiency
  - Logical AND operator ( `&&` )
    - If one operand is false, the result is false.
  - Logical OR operator ( `||` )
    - If one operand is true, the result is true.

# Advanced C

## Operators - Relational



021\_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;
}
```

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

What will be  
the output?

# Advanced C

## Operators - Assignment

### 022\_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;
}
```

### 023\_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;
}
```

# Advanced C

## Operators - Bitwise



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



# Advanced C

## Operators - Bitwise



&amp;

### Bitwise AND

Bitwise ANDing of all the bits in two operands

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

|

### Bitwise OR

Bitwise ORing of all the bits in two operands

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

# Advanced C

## Operators - Bitwise



^

### Bitwise XOR

Bitwise XORing of all the bits in two operands

| Operand      | Value           |
|--------------|-----------------|
| A            | 0x61            |
| B            | 0x13            |
| $A \wedge B$ | 0x72            |
|              | 0 1 1 1 0 0 1 0 |

~

### Compliment

Complimenting all the bits of the operand

| Operand  | Value           |
|----------|-----------------|
| A        | 0x61            |
| $\sim A$ | 0x9E            |
|          | 1 0 0 1 1 1 1 0 |

# Advanced C

## Operators - Bitwise - Shift



### Syntax

#### Left Shift :

*shift-expression << additive-expression*

(left operand)

(right operand)

#### Right Shift :

*shift-expression >> additive-expression*

(left operand)

(right operand)



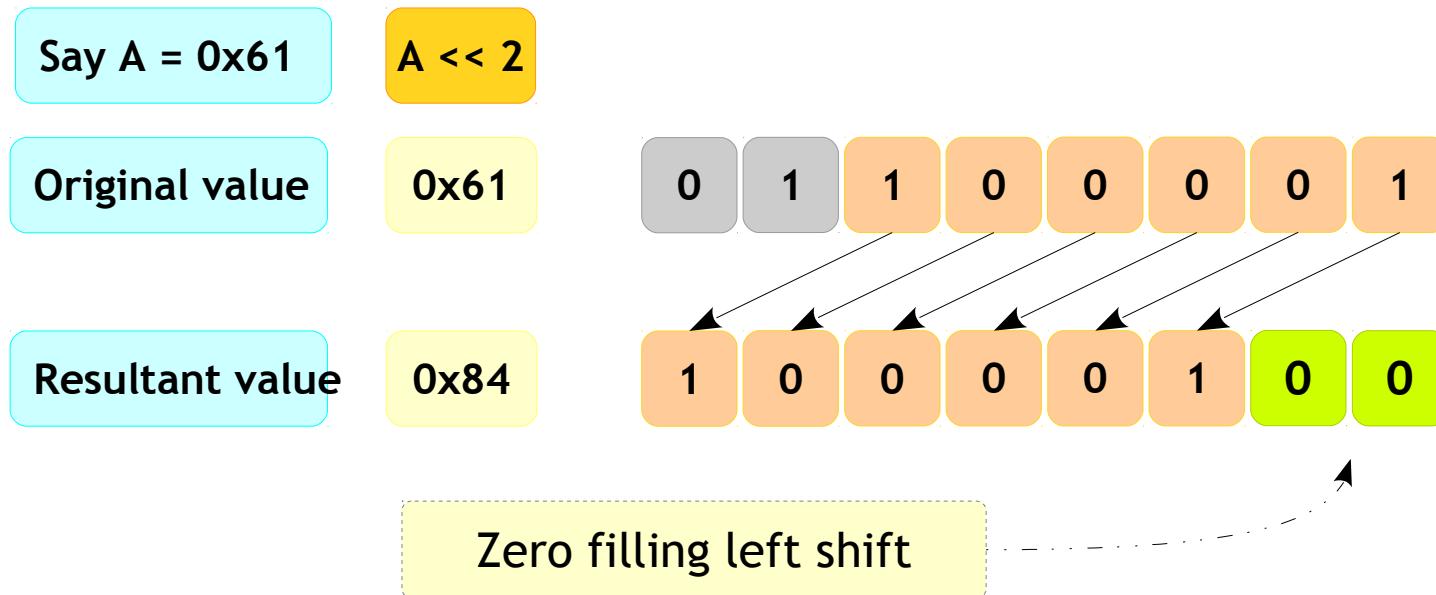
# Advanced C

## Operators - Bitwise - Left Shift



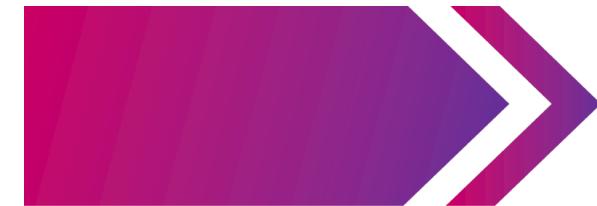
### 'Value' << 'Bits Count'

- 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



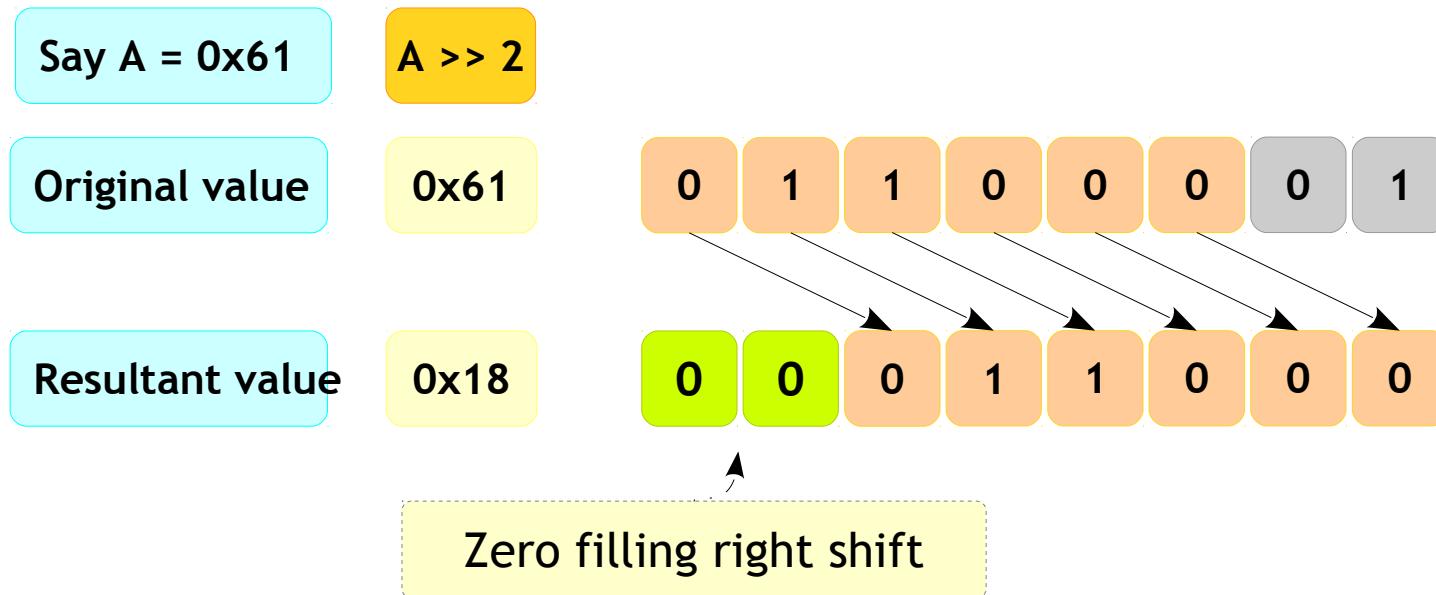
# Advanced C

## 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

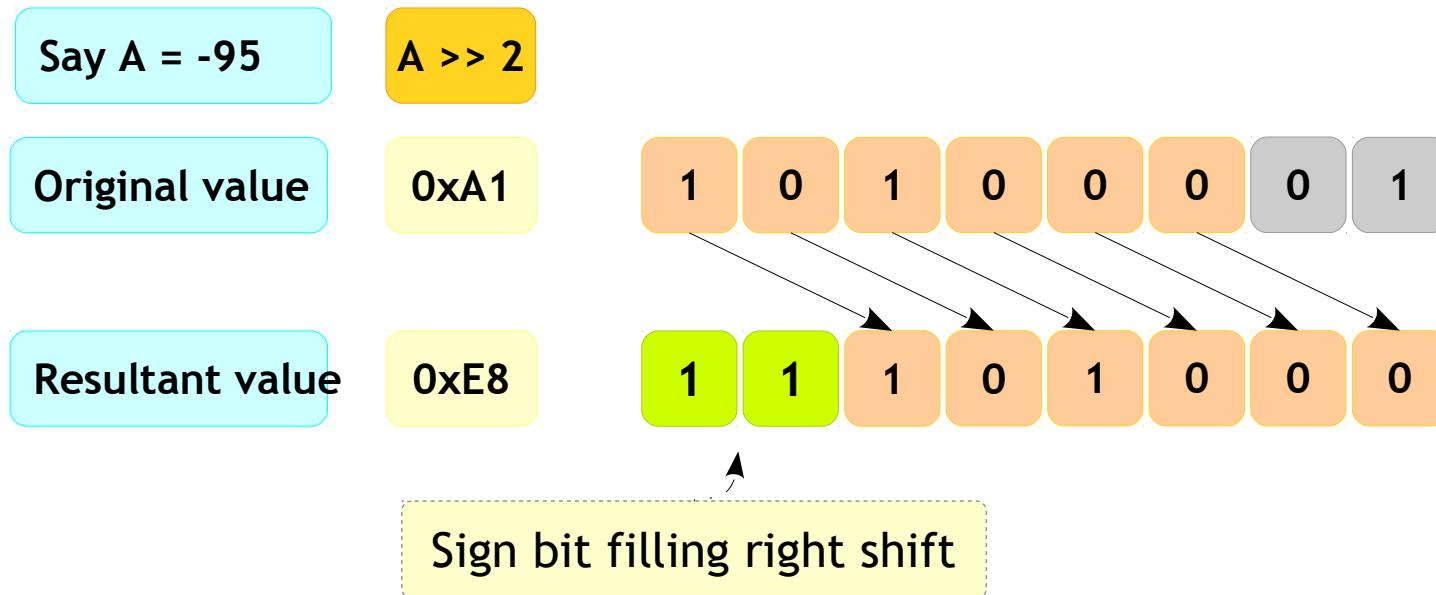


# Advanced C

## Operators - Bitwise - Right Shift - Signed Value

**“Signed Value” >> ‘Bits Count’**

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



# Advanced C

## Operators - Bitwise



### 024\_example.c

```
#include <stdio.h>

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

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

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

    return 0;
}
```



# Advanced C

## Operators - Bitwise - Shift



- Each of the operands shall have integer type
- The integer promotions are performed on each of the operands
- If the value of the right operand is **negative** or is greater than or equal to the **width of the promoted left operand**, the behavior is undefined
- Left shift (`<<`) operator : If left operand has a signed type and nonnegative value, and  $(\text{left\_operand} * (2^n))$  is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined

# Advanced C

## Operators - Bitwise - Shift

- The below example has undefined behaviour

### 025\_example.c

```
#include <stdio.h>

int main()
{
    int x = 7, y = 7;

    x = 7 << 32;
    printf("x is %x\n", x);

    x = y << 32;
    printf("x is %x\n", x);

    return 0;
}
```

# Advanced C

## Operators - Bitwise - Mask



- If you want to create the below art assuming your are not a good painter, What would you do?

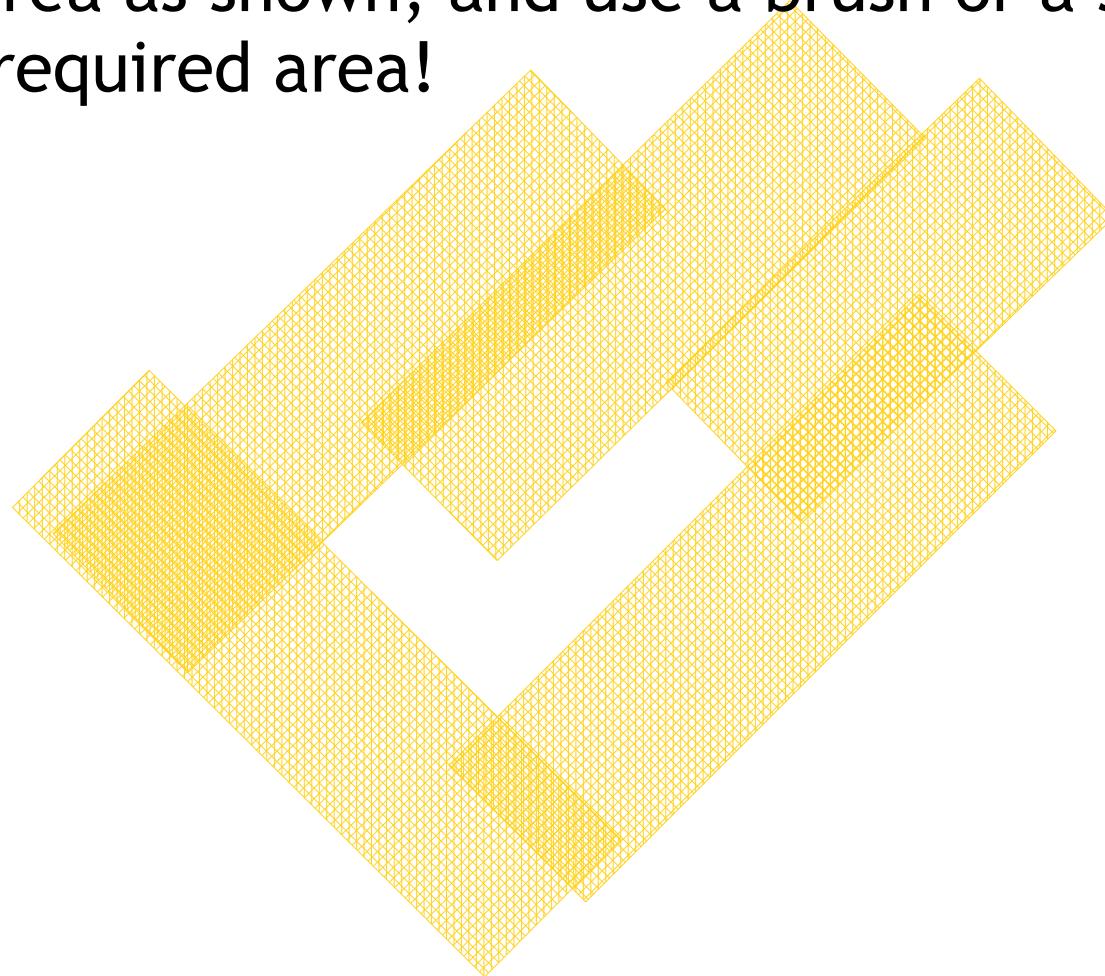


# Advanced C

## Operators - Bitwise - Mask



- Mask the area as shown, and use a brush or a spray paint to fill the required area!

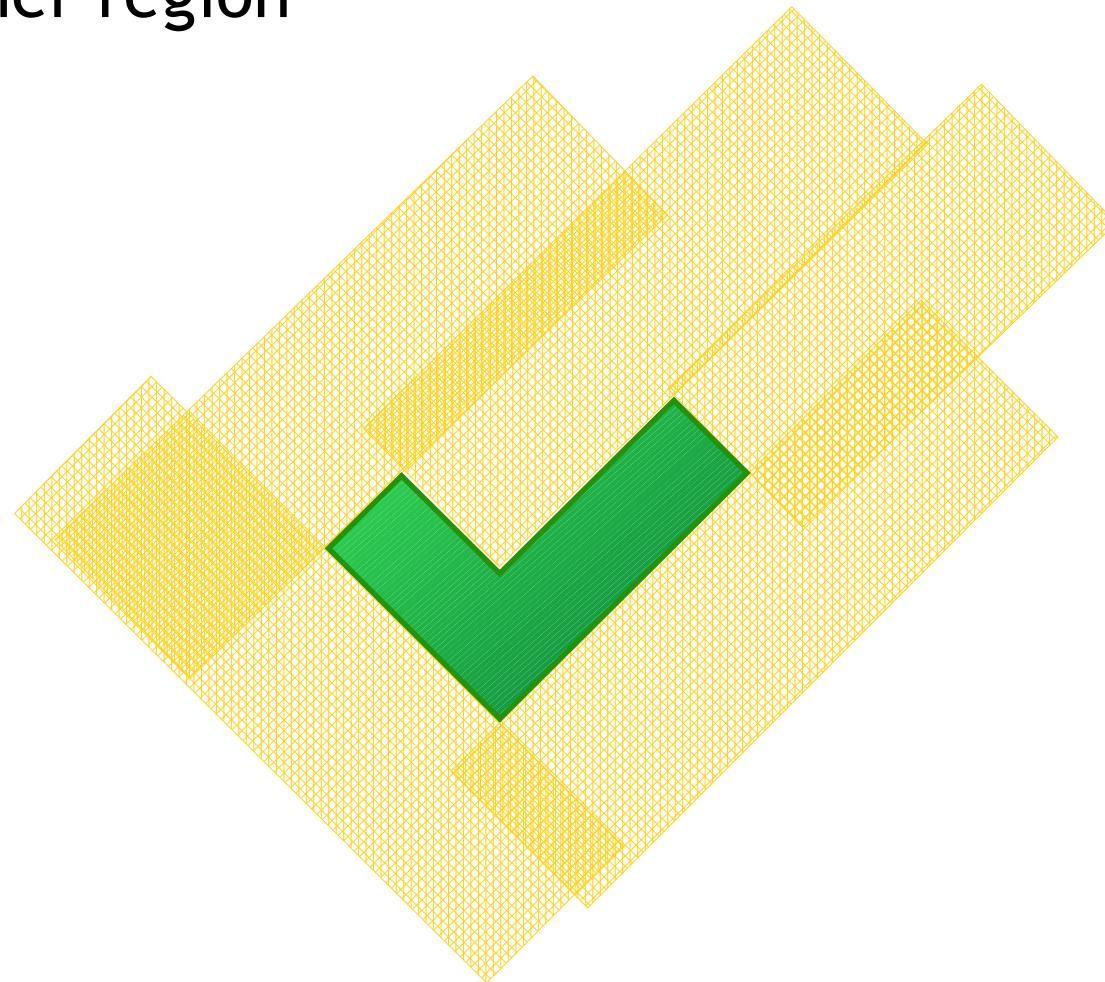


# Advanced C

## Operators - Bitwise - Mask



- Fill the inner region



# Advanced C

## Operators - Bitwise - Mask



- Remove the mask tape



# Advanced C

## Operators - Bitwise - Mask



- So masking, technically means unprotecting the required bits of register and perform the actions like
  - Set Bit
  - Clear Bit
  - Get Bit
  - etc,..

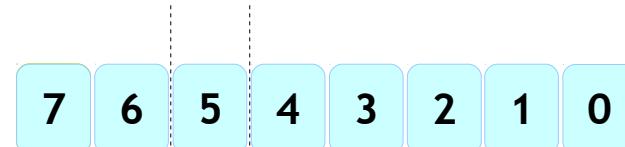


# Advanced C

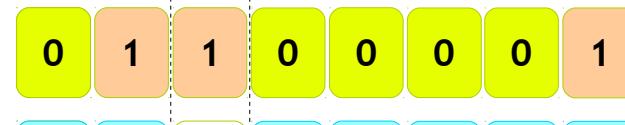
## Operators - Bitwise - Mask



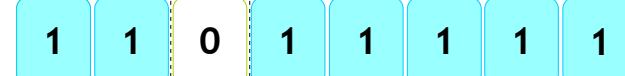
& Operator



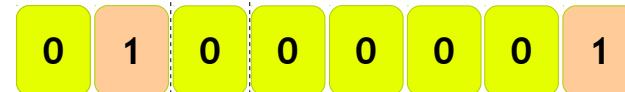
Bit Position



The register to be modified

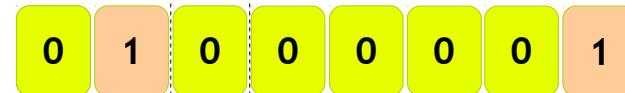


Mask to **CLEAR** 5<sup>th</sup> bit position



The result

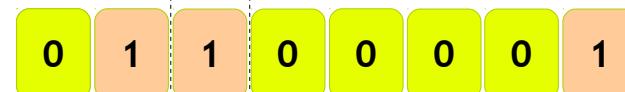
| Operator



The register to be modified



Mask to **SET** 5<sup>th</sup> bit position



The result

# Advanced C

## Operators - Bitwise - Shift



- W.A.P to count set bits in a given number
- W.A.P to print bits of given number
- W.A.P to swap nibbles of given number



# Advanced C

## Operators - Ternary



### Syntax

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

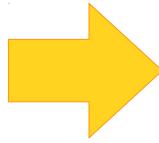
### 025\_example.c

```
#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;
}
```



```
#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;
}
```

# Advanced C

## Operators - Comma



- The left operand of a comma operator is evaluated as a void expression (result discarded)
- 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 function
  - 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`

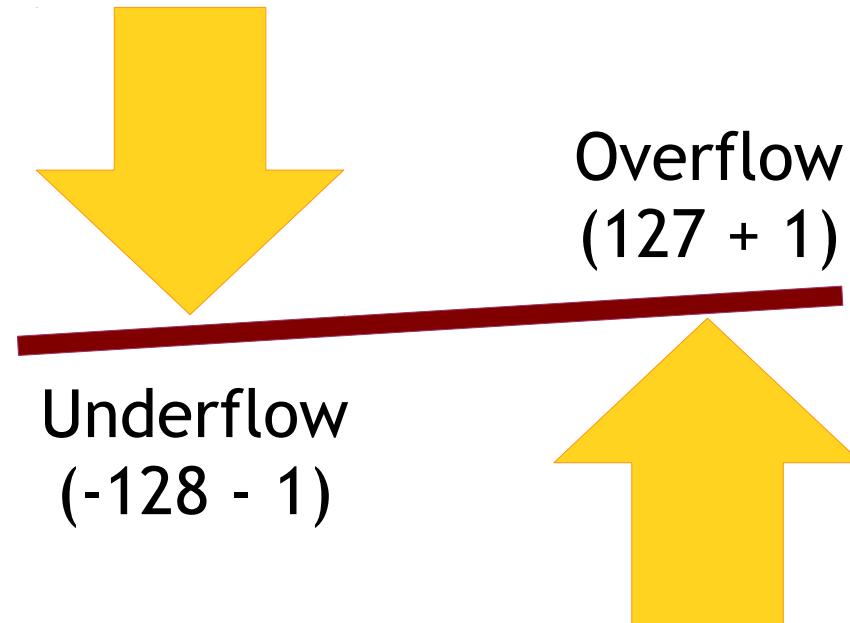


# Advanced C

## Over and Underflow



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



# Advanced C

## Overflow - Signed Numbers



Say A = +127

Original value

0x7F

Add

1

Resultant value

0x80

0 1 1 1 1 1 1 1

0 0 0 0 0 0 0 1

1 0 0 0 0 0 0 0

Sign bit

# Advanced C

## Underflow - Signed Numbers



Say A = -128

Original value

0x80



Add

-1



Resultant value

0x7F



Sign bit  
Spill over bit is discarded

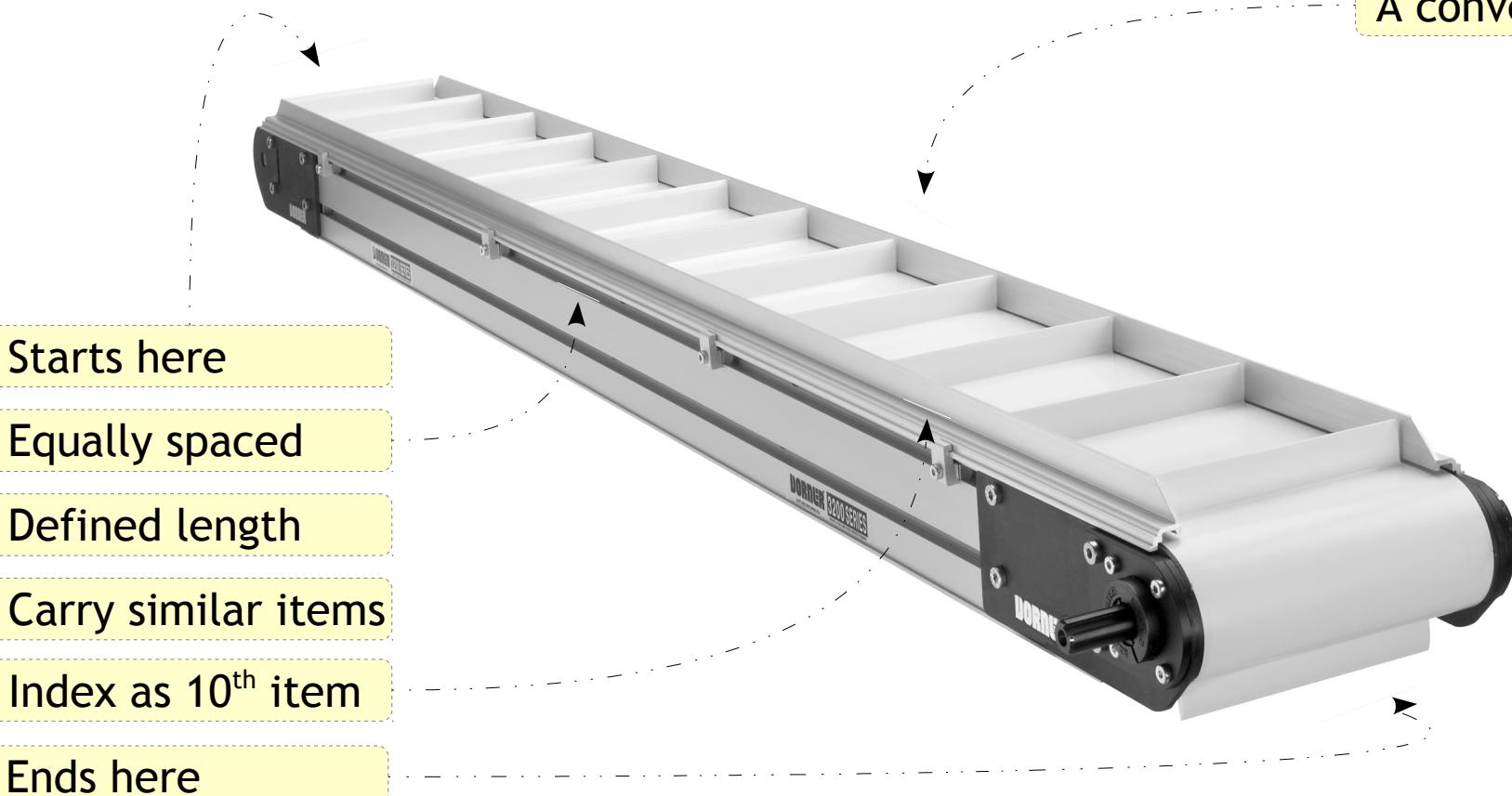
# Arrays



# Advanced C

## Arrays - Know the Concept

A conveyor belt

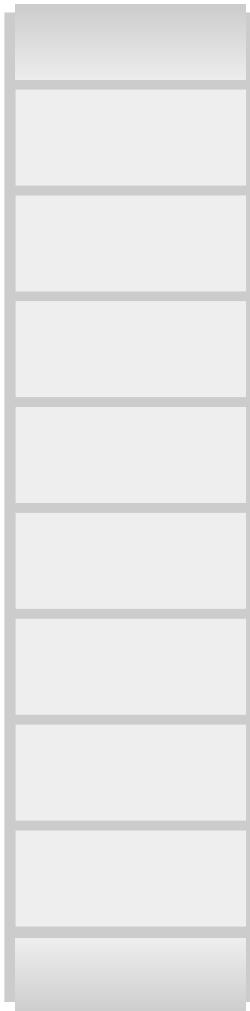


# Advanced C

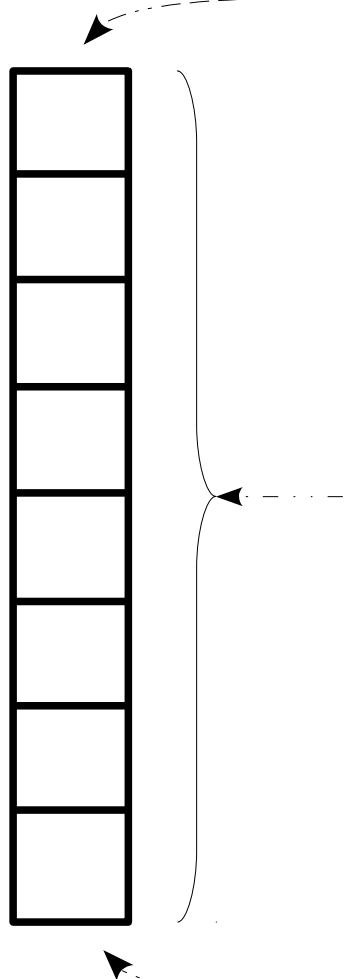
## Arrays - Know the Concept



Conveyor Belt  
Top view



An Array



First Element  
Start (Base) address

- Total Elements
- Fixed size
- Contiguous Address
- Elements are accessed by indexing
- Legal access region

Last Element  
End address

# Advanced C

## Arrays



### Syntax

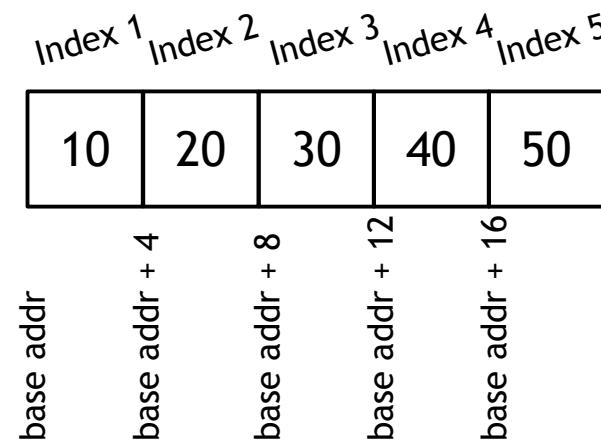
```
data_type name[SIZE];
```

Where SIZE represents number of elements

Memory occupied by array = (number of elements \* size of an element)  
= (SIZE \* <size of data\_type>)

### Example

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



# Advanced C

## Arrays - Points to be noted



- An array is a collection of similar data type
- Elements occupy consecutive memory locations (addresses)
- First element with lowest address and the last element with highest address
- Elements are indexed from 0 to SIZE - 1. Example : 5 elements array (say array[5]) will be indexed from 0 to 4
- Accessing out of range array elements would be “illegal access”
  - Example : Do not access elements array[-1] and array[SIZE]
- Array size can't be altered at run time

# Advanced C

## Arrays - Why?

### 027\_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;
}
```



```
#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;
}
```

# Advanced C

## Arrays - Reading



### 028\_example.c

```
#include <stdio.h>

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

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

    return 0;
}
```



# Advanced C

## Arrays - Storing



### 029\_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;
}
```



# Advanced C

## Arrays - Initializing



### 030\_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;
}
```



# Advanced C

## Arrays - Copying



- Can we copy 2 arrays? If yes how?

031\_example.c

```
#include <stdio.h>

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

    array_bak = array_org;

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

    return 0;
}
```



# Advanced 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**



# Advanced C

## Arrays - DIY



- W.A.P to find the average of elements stored in a array.
  - Read value of elements from user
  - For given set of values : { 13, 5, -1, 8, 17 }
  - Average Result = 8.4
- W.A.P to find the largest array element
  - Example 100 is the largest in {5, **100**, -2, 75, 42}

# Advanced C

## Arrays - DIY



- W.A.P to compare two arrays (element by element).
  - Take equal size arrays
  - Arrays shall have unique values stored in random order
  - Array elements shall be entered by user
  - Arrays are compared “EQUAL” if there is one to one mapping of array elements value
  - Print final result “EQUAL” or “NOT EQUAL”

Example of Equal Arrays :

- A[3] = {2, -50, 17}
- B[3] = {17, 2, -50}

# Advanced C

Arrays - Oops!! what is this now?



## *Recommended Coding Standards at Emertxe*

### Introduction :

The intension of having a coding guidelines is to have:

Code quality - Having the code in a uniform pattern so that it is easily readable by every one working in a same organization.

Productivity - Your code will be used, read, extended, interfaced-to, modified, documented, and ported by many other people, so it is important that every one use a consistent coding style. Good style should encourage consistent layout, improve portability, and reduce errors.

Ultimately, the goal of these standards is to increase portability, reduce maintenance, and above all improve clarity.

*``To be clear is professional; not to be clear is unprofessional.'' -- Sir Ernest Gowers.*

### File Organization :

A file consists of various sections that should be separated by several blank lines. Although there is no maximum length limit for source files, files with more than about 1000 lines are cumbersome to deal with. The editor may not have enough temp space to edit the file, compilations will go more slowly, etc. Many rows of asterisks, for example, present little information compared to the time it takes to scroll past, and are discouraged. Lines longer than 80 columns are not handled well by all terminals and should be avoided if possible. Excessively long lines which result from deep indenting are often a symptom of poorly-organized code.

### File Naming Conventions :

File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter, and all characters (except the period) should be lower-case letters and numbers.

- Any project should have a main file along with its header file named as "main.\*" where \* is the file extension

eg:

main.c

main.cc,

main.h, etc.,,

- The associated file should be named as based on its application with appropriate extensions

eg: If you are using i2c protocol then it should be named as

i2c.c

i2c.h, etc.,.

In addition, it is conventional to use “Makefile” (not “makefile”) for the control file for *make* (for systems that support it) and “README” for a summary of the contents of the directory or directory tree.

#### Program Files :

The suggested order of sections for a program file is as follows:

1. First in the file is a prologue that tells what is in that file. A description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names. The prologue may optionally contain author(s), revision control information, references, etc.
2. Any header file includes should be next. If the include is for a non-obvious reason, the reason should be commented. In most cases, system include files like stdio.h should be included before user include files.
3. Any defines and typedefs that apply to the file as a whole are next. One normal order is to have “constant” macros first, then “function” macros, then typedefs and enums.
4. Next come the global (external) data declarations, usually in the order: externs, non-static globals, static globals. If a set of defines applies to a particular piece of global data (such as a flags word), the defines should be immediately after the data declaration or embedded in structure declarations, indented to put the defines one level deeper than the first keyword of the declaration to which they apply.
5. The functions come last, and should be in some sort of meaningful order. Like functions should appear together. A “breadth-first” approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls). Considerable judgment is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

eg:

```
/*
 * Author : Xyz
 * Organization : Emertxe Information Technologies (P) Ltd.
 * Date : 01-12-2006
 * Usage : This is a example to show how to maintain a program file
 */
```

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

static int xyz;

static void increment()
{
    xyz++;
}

void main(void)
{
    increment();

    return 0;
}
```

### Header Files :

Header files are files that are included in other files prior to compilation by the C preprocessor. Some, such as stdio.h, are defined at the system level and must be included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program. Header files should be functionally organized, i.e., declarations for separate subsystems should be in separate header files. Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

Avoid private header filenames that are the same as library header filenames. The statement

```
#include "math.h"
```

will include the standard library math header file if the intended one is not found in the current directory. If this is what you *want* to happen, comment this fact. Don't use absolute pathnames for header files. Use the `<name>` construction for getting them from a standard place, or define them relative to the current directory. The ``include-path'' option of the C compiler (-I on many systems) is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

Header files that declare functions or external variables should be included in the file that defines the function or variable. That way, the compiler can do type checking and the external declaration will always agree with the definition.

Defining variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files. Also, some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. On some systems, repeating uninitialized declarations without the `extern` keyword also causes problems. Repeated declarations can happen if include files are nested and will cause the compilation to fail.

Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be `#included` for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common `#includes` in one include file. Header files should be self contained in header inclusions.

It is common to put the following into each `.h` file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE_H  
#define EXAMPLE_H  
  
...           /*      body      of      example.h      file      */  
  
#endif /* EXAMPLE_H */
```

This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

## Other Files :

It is conventional to have a file called ``README'' to document both ``the bigger picture'' and issues for the program as a whole. For example, it is common to include a list of all conditional compilation flags and what they mean. It is also common to list files that are machine dependent, etc.

## Comments :

``When the code and the comments disagree, both are probably wrong." -- Norm Schryer

The comments should describe *what* is happening, *how* it is being done, what parameters mean, which globals are used and which are modified, and any restrictions or bugs. Avoid, however, comments that are clear from the code, as such information rapidly gets out of date. Comments that disagree with the code are of negative value. Short comments should be *what* comments, such as "compute mean value", rather than *how* comments such as "sum of values divided by n". C is not assembler; putting a comment at the top of a 3--10 line section telling what it does overall is often more useful than a comment on each line describing micrologic.

Comments should justify offensive code. The justification should be that something bad will happen if unoffensive code is used. Just making code faster is not enough to rationalize a hack; the performance must be *shown* to be unacceptable without the hack. The comment should explain the unacceptable behavior and describe why the hack is a "good" fix.

Comments that describe data structures, algorithms, etc., should be in block comment form.

```
/*
 *           Here           is           a           block           comment.
 *           The           comment           text           should           spaced           over           uniformly.
 *           The           opening           slash-star           and           closing           star-slash           are           alone           on           a           line.
 */
```

Note that grep '^.\e\*' will catch all block comments in the file. Very long block comments such as drawn-out discussions and copyright notices often start with /\* in columns 1-2, no leading \* before lines of text, and the closing \*/ in columns 1-2. Block comments inside a function are appropriate, and they should be tabbed over to the same tab setting as the code that they describe. One-line comments alone on a line should be indented to the tab setting of the code that follows.

```
if           (argc           >           1)
{
    /*           Get           input           file           from           command           line.           */
    if           (fopen(argv[1],           "r",           stdin)           ==           NULL)
    {
        perror(argv[1]);
    }
}
```

Very short comments may appear on the same line as the code they describe, and should be tabbed over to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

```

if                               (a          ==          EXCEPTION)
{
    b      =      TRUE;           /*      special      case      */
}
else
{
    b      =      isprime(a);   /*      works      only      for      odd      a      */
}

```

## Declarations

Global declarations should begin in column 1. All external data declaration should be preceded by the `extern` keyword. If an external variable is an array that is defined with an explicit size, then the array bounds must be repeated in the `extern` declaration unless the size is always encoded in the array (e.g., a read-only character array that is always null-terminated). Repeated size declarations are particularly beneficial to someone picking up code written by another.

The “pointer” qualifier, `*`, should be with the variable name rather than with the type.

```
char *s, *t, *u;
```

instead of

```
char* s, t, u;
```

which is wrong, since `t` and `u` do not get declared as pointers.

Unrelated declarations, even of the same type, should be on separate lines. A comment describing the role of the object being declared should be included, with the exception that a list of `#defined` constants do not need comments if the constant names are sufficient documentation. The names, values, and comments are usually tabbed so that they line up underneath each other. Use the tab character rather than blanks (spaces). For structure and union template declarations, each element should be alone on a line with a comment describing it. The opening brace (`{`) should be on the next line as the structure tag, and the closing brace (`}`) should be in column 1.

```

struct                                boat
{
    int     wllength;           /*      water      line      length      in      meters      */
    int     type;              /*      see      below      */
    long    sailarea;          /*      sail      area      in      square      mm      */
};

/*
     defines          for          boat.type          */
#define          KETCH          (1)
#define          YAWL           (2)
#define          SLOOP          (3)

```

```
#define SQRIG
#define MOTOR (5) (4)
```

These defines are sometimes put right after the declaration of *type*, within the struct declaration, with enough tabs after the # to indent define one level more than the structure member declarations. When the actual values are unimportant, the enum facility is better .

```
enum bt
{
e_ketch=1,
e_yawl,
e_sloop,
e_sqrig,
e_motor
};

struct boat
{
    int    wllength;          /* water line length in meters */
    enum   bt     Type;      /* what kind of boat */
    long   sailarea;         /* sail area in square mm */
};
```

Any variable whose initial value is important should be *explicitly* initialized, or at the very least should be commented to indicate that C's default initialization to zero is being relied upon. The empty initializer, “{},” should never be used. Structure initializations should be fully parenthesized with braces. Constants used to initialize longs should be explicitly long. Use capital letters; for example two long 21 looks a lot like 21, the number twenty-one.

```
int           x           =           1;
char          *msg        =           "message";
struct boat winner[] = {{ 40, YAWL, 6000000L }, { 28, MOTOR, 0L }, { 0 }, };
```

In any file which is part of a larger whole rather than a self-contained program, maximum use should be made of the static keyword to make functions and variables local to single files. Variables in particular should be accessible from other files only when there is a clear need that cannot be filled in another way. Such usage should be commented to make it clear that another file's variables are being used; the comment should name the other file. If your debugger hides static objects you need to see during debugging, declare them as STATIC and #define STATIC as needed.

The most important types should be highlighted by typedeffing them, even if they are only integers, as the unique name makes the program easier to read (as long as there are only a few things typedeffed to integers!). Structures may be typedeffed when they are declared. Give the struct and the typedef the same name.

```
typedef struct sSplodge_tT
{
    int sp_count;
```

```
char *sp_name, *sp_alias;  
} Splodge_T;
```

The return type of functions should always be declared. If function prototypes are available, use them. One common mistake is to omit the declaration of external math functions that return double. The compiler then assumes that the return value is an integer and the bits are dutifully converted into a (meaningless) floating point value.

*“C takes the point of view that the programmer is always right.”* -- Michael DeCorte

## Function Declarations

Each function should be preceded by a block comment prologue that gives a short description of what the function does and (if not clear) how to use it. Discussion of non-trivial design decisions and side-effects is also appropriate. Avoid duplicating information clear from the code.

The function return type should be alone on a line, (optionally) indented one stop . Do not default to int; if the function does not return a value then it should be given return type *void* . If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be on the same line as the return type, one space over. The function return type, name (and the formal parameter list) should be on a same line, one space after return type. Destination (return value) parameters should generally be first (on the left). All formal parameter declarations, local declarations and code within the function body should be tabbed over one stop. The opening brace of the function body should be alone on a line beginning in column 1.

Each parameter should be declared (do not default to int). In general the role of each variable in the function should be described. This may either be done in the function comment or, if each declaration is on its own line, in a comment on that line. Loop counters called “i”, string pointers called “s”, and integral types called “c” and used for characters are typically excluded. If a group of functions all have a like parameter or local variable, it helps to call the repeated variable by the same name in all functions. (Conversely, avoid using the same name for different purposes in related functions.) Like parameters should also appear in the same place in the various argument lists.

Comments for parameters and local variables should be tabbed so that they line up underneath each other. Local variable declarations should be separated from the function's statements by a blank line.

Be careful when you use or declare functions that take a variable number of arguments (“varargs”). There is no truly portable way to do varargs in C. Better to design an interface that uses a fixed number of arguments. If you must have varargs, use the library macros for declaring functions with variant argument lists.

If the function uses any external variables (or functions), they should come from some header file that are not declared globally in the file, these should have their own declarations in the function body using the *extern* keyword.

Avoid local declarations that override declarations at higher levels. In particular, local variables should not be redeclared in nested blocks. Although this is valid C, the potential confusion is enough that *splint* will complain about it when given the --h option.

## Whitespace

```
int i;main(){for(;i["]<i;++i){--i;}"]};read('---',i+++ "hell\nworld!\n", '/\');}read(j,i,p){write(j/p+p,i--j,i/i);}  
-- An example of code without whitespaces (decide how it is).
```

Use vertical and horizontal whitespace generously. Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 1 blank lines between the end of one function and the comments for the next.

A long string of conditional operators should be split onto separate lines.

```
if (emertxe->next == NULL && totalcount < needed && needed <= MAX_ALLOC  
&& server_active(current_input))  
{ ...}
```

Might be better as

```
if (emertxe->next == NULL  
&& totalcount < needed && needed <= MAX_ALLOC  
&& server_active(current_input))  
{  
...}
```

Similarly, elaborate for loops should be split onto different lines.

```
for (curr = *listp, trail = listp;  
curr != NULL;  
trail = &(curr->next), curr = curr->next)  
{  
...}
```

Other complex expressions, particularly those using the ternary ?: operator, are best split on to several lines, too.

```
c = (a == b)  
? d + f(a)  
: f(b) - d;
```

Keywords that are followed by expressions in parentheses should be separated from the left parenthesis by a blank. (The sizeof operator is an exception.) Blanks should also appear after commas in argument lists to help separate the arguments visually. On the other hand, macro definitions with arguments must not have a blank between the name and the left parenthesis, otherwise the C preprocessor will not recognize the argument list.

## Examples

```

/*
 * Determine if the sky is blue by checking that it isn't night.
 * CAVEAT: Only sometimes right. May return TRUE when the answer
 * is FALSE. Consider clouds, eclipses, short days.
 * NOTE: Uses 'hour' from 'hightime.c'. Returns 'int' for
 * compatibility with the old version.
*/
int skyblue() /* true or false */
{
    extern int hour; /* current hour of the day */
    return (hour >= MORNING && hour <= EVENING);
}
/*
 * Find the last element in the linked list
 * pointed to by nodep and return a pointer to it.
 * Return NULL if there is no last element.
*/
node_t *tail(nodep)
node_t *nodep;
{
    register node_t *np; /* advances to NULL */
    register node_t *lp; /* follows one behind np */

    if (nodep == NULL) /* VOID */
    {
        return (NULL);
    }
    for (np = lp = nodep; np != NULL; lp = np, np = np->next)
    {
        ;
    }
    return (lp);
}

```

## Simple Statements

There should be only one statement per line.

```
case
  oogle(zork);
```

BAS:

```

    boogle(zork);
    break;
case                                BAR:
    oogle(bork);
    boogle(zork);
    break;
case BAZ:
    oogle(gork);
    boogle(bork);
    break;

```

The null body of a for or while loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```

while          (*dest++      =      *src++)
{
; /* VOID */
}

```

Do not default the test for non-zero, i.e.

```
if (f() != FAIL)
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be --1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g.,

```
if (!(bufsize % sizeof(int)))
```

should be written instead as

```
if ((bufsize % sizeof(int)) == 0)
```

to reflect the *numeric* (not *boolean*) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*.

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

The non-zero test *is* often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Evaluates to 0 for false, nothing else.
- Is named so that the meaning of (say) a 'true' return is absolutely obvious.

Call a predicate *isValid* or *valid*, not *checkvalid*.

It is common practice to declare a boolean type bool in a global include file. The special names improve readability immensely.

```
typedef int bool;
#define FALSE 0
#define TRUE 1
```

or

```
typedef enum {
    NO = 0,
    YES e_false,
    e_true
} Bool;
```

Even with these declarations, do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (func() == TRUE)
{
    ...
}
```

must be written

```
if (func() != FALSE)
{
    ...
}
```

It is even better (where possible) to rename the function/variable or rewrite the expression so that the meaning is obvious without a comparison to true or false (e.g., rename to *isValid()*).

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

```
while ((c = getchar()) != EOF)
{
    process the character
}
```

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

```
a = b + c;  
d = a + r;
```

should not be replaced by

```
d = (a = b + c) + r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

As far as possible avoid *goto*. *goto* statements should be used sparingly, as in any well-structured code. The main place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

```
for (...) {  
    while (...) {  
        {  
            ...  
            if (...) {  
                goto error; // (disaster)  
            }  
        }  
    }  
}  
...  
error:  
    clean up the mess
```

When a *goto* is necessary the accompanying label should be alone on a line and tabbed one stop to the left of the code that follows. The *goto* should be commented (possibly in the block header) as to its utility and purpose. *continue* should be used sparingly and near the top of the loop. *break* is less troublesome.

Parameters to non-prototyped functions sometimes need to be promoted explicitly. If, for example, a function expects a 32-bit long and gets handed a 16-bit int instead, the stack can get misaligned. Problems occur with pointer, integral, and floating-point values. Function calls should always have a declaration before it, either explicitly or implicitly through its definition

## Compound Statements :

A compound statement is a list of statements enclosed by braces. There are many common ways of formatting the braces. Be consistent with the following standard.

```
control {  
    statement;
```

```
    statement;  
}
```

When a block of code has several labels (unless there are a lot of them), the labels are placed on separate lines. The fall-through feature of the C switch statement, (that is, when there is no break between a code segment and the next case statement) must be commented for future maintenance. A splint-style comment/directive is best.

```
switch   (expr)  
{  
case   ABC:  
case   DEF:  
    statement;  
    break;  
case   UVW:  
    statement;  
    /*FALLTHROUGH*/  
case   XYZ:  
    statement;  
    break;  
}
```

Here, the last break is unnecessary, but is required because it prevents a fall-through error if another case is added later after the last one. The default case, if used, should be used and as the last one and does not require with a break. if it is last.

Whenever an if-else statement has a compound statement for either the if or else section, the statements of both the if and else sections should both be enclosed in braces (called *fully bracketed syntax*).

```
if  (expr)  
{  
    statement;  
}  
else  
{  
    statement;  
    statement;  
}
```

Braces are also essential in *if-if-else* sequences with no second *else* such as the following, which will be parsed incorrectly if the brace after (ex1) and its mate are omitted:

```
if  (ex1)  
{  
    if  (ex2)  
    {  
        funcA();  
    }
```

```

}
else
{
    funcb();
}

```

An *if-else* with *else if* should be written with the *else* conditions left-justified.

```

if                      (STREQ             (reply,           "yes"))
{
    statements          for               yes
    ...
}
else                    if                  (STREQ             (reply,           "no"))
{
    ...
}
else                    if                  (STREQ             (reply,           "maybe"))
{
    ...
}
else
{
    statements          for               default
    ...
}

```

The format then looks like a generalized *switch* statement and the tabbing reflects the switch between exactly one of several alternatives rather than a nesting of statements.

Try to have an else always with an if

do-while loops should always have braces around the body.

Sometimes an if causes an unconditional control transfer via break, continue ,goto, or return. The else should be implicit and the code should not be indented.

```

if                      (level            >              limit)
{
    return
}
normal();
return (level);

```

The “flattened” indentation tells the reader that the boolean test is invariant over the rest of the enclosing block.

## Operators

Unary operators should not be separated from their single operand. Generally, all binary operators except '.' and '->' should be separated from their operands by blanks, on either side. Some judgement is called for in the case of complex expressions, which may be clearer if the “inner” operators are not surrounded by spaces and the “outer” ones are.

If you think an expression will be hard to read, consider breaking it across lines. Splitting at the lowest-precedence operator near the break is best. Since C has some unexpected precedence rules, expressions involving mixed operators should be parenthesized. Too many parentheses, however, can make a line *harder* to read because humans aren't good at parenthesis-matching.

## Naming Conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names.
- `#define` constants should be in all CAPS.
- `enum` constants are *e\_name* where *e* is for enum.
- Lower-case macro names are only acceptable if the macros behave like a function call, that is, they evaluate their parameters *exactly* once and do not assign values to named parameters. Sometimes it is impossible to write a macro that behaves like a function even though the arguments are evaluated exactly once.
- Avoid names that differ only in case, like *emertxe* and *Emertxe*. Similarly, avoid *emertxebar* and *Emertxe\_bar*. The potential for confusion is considerable. Use Hungarian notation for all typedefs, and classic C notation for all the remaining identifiers
- Similarly, avoid names that look like each other. On many terminals and printers, 'l', '1' and 'I' look quite similar. A variable named 'l' is particularly bad because it looks so much like the constant '1'.

The priority of using variables should be in the following order: local, static local, static global, global – local being the first. Globals, if used as a final resort, may alternatively should be grouped in a global structure. variable starting with g\_.typedeffed names have \_t appended to their name.

Avoid names that might conflict with various standard library names. Some systems will include more library code than you want. Also, your program may be extended someday.

## Constants

Numerical constants should not be coded directly. The `#define` feature of the C preprocessor should be used to give constants meaningful names. Symbolic constants make the code easier to read. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define.

The enumeration data type is a better way to declare variables that take on only a discrete set of values. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

Constants should be defined consistently with their use; e.g. use 540.0 for a floatdouble instead of 540 with an implicit floatdouble cast. There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a for loop indexes through an array, then

```
for (i = 0; i < ARYBOUND; i++)
```

is reasonable while the code

```
door_t *front_door = opens(door[i], 7);
if           (front_door == 0)
{
    error("can't open %s\\n", door[i]);
}
```

is not. In the last example front\_door is a pointer. When a value is a pointer it should be compared to NULL instead of 0. NULL is available as part of the standard I/O library's header file *stdio.h*. Even simple values like 1 or 0 are often better expressed using defines like TRUE and FALSE (sometimes YES and NO read better).

Simple character constants should be defined as character literals rather than numbers. Non-text characters are discouraged as non-portable. If non-text characters are necessary, particularly if they are used in strings, they should be written using a escape character of three octal digits rather than one (e.g., '\007'). Even so, such usage should be considered machine-dependent and treated as such.

## Macros

Complex expressions can be used as macro parameters, and operator-precedence problems can arise unless all occurrences of parameters have parentheses around them.

Some macros also exist as functions (e.g., getc and fgetc). The macro should be used in implementing the function so that changes to the macro will be automatically reflected in the function. Care is needed when interchanging macros and functions since function parameters are passed by value, while macro parameters are passed by name substitution. Carefree use of macros requires that they be declared carefully.

Macros should avoid using globals, since the global name may be hidden by a local declaration. Macros that change named parameters (rather than the storage they point at) or may be used as the left-hand side of an assignment should mention this in their comments. Macros that take no parameters but reference variables, are long, or are aliases for function calls should be given an empty parameter list, e.g.,

```
#define OFF_A() (a_global+OFFSET)
#define BORK() (zork())
#define SP3() if (b) { int x; av = f(&x); bv += x; }
```

Macros save function call/return overhead, but when a macro gets long, the effect of the call/return becomes negligible, so a function should be used instead.

In some cases it is appropriate to make the compiler insure that a macro is terminated with a semicolon.

```
if (x==3)
{
    SP3();
}
else
{
    BORK();
}
```

If the semicolon is omitted after the call to SP3, then the else will (silently!) become associated with the if in the SP3 macro. With the semicolon, the else doesn't match *any* if ! The macro SP3 can be rewritten for that safely as: (Review this b4 accepting)

```
#define SP3()          \\\\
    do { if (b) { int x; av = f(&x); bv += x; } } while (0)
```

Writing out the enclosing do-while by hand is awkward and some compilers and tools may complain that there is a constant in the while conditional. A macro for declaring statements may make programming easier.

```
#ifdef                      splint
    static                   int
#else                         ZERO;
#else
#endif                         define      ZERO      0
#define STMT( stuff )     do { stuff } while (ZERO)
```

Declare SP3 with

```
#define SP3()          \\\\
    STMT( if (b) { int x; av = f(&x); bv += x; } )
```

Using STMT will help prevent small typos from silently changing programs.

Except for type casts, sizeof, and hacks such as the above, macros should contain keywords only if the entire macro is surrounded by braces.

## Conditional Compilation

Conditional compilation is useful for things like machine-dependencies, debugging, and for setting certain options at compile-time. Beware of conditional compilation. Various controls can easily combine in unforeseen ways. If you `#ifdef` machine dependencies, make sure that when no machine is specified, the result is an error, not a default machine. (Use `#error` and indent it so it works with older compilers.) If you `#ifdef` optimizations, the default should be the unoptimized code rather than an un compilable program. Be sure to test the unoptimized code.

Note that the text inside of an `#ifdef'fed` section may be scanned (processed) by the compiler, even if the `#ifdef` is false. Thus, even if the `#ifdef'fed` part of the file never gets compiled (e.g., `#ifdef COMMENT`), it cannot be arbitrary text.

Put `#ifdefs` in header files instead of source files when possible. Use the `#ifdefs` to define macros that can be used uniformly in the code. For instance, a header file for checking memory allocation might look like (omitting definitions for `REALLOC` and `FREE`):

```
#ifdef DEBUG
    extern void *mm_malloc();
#define MALLOC(size) (mm_malloc(size))
#else
    extern void *malloc();
#define MALLOC(size) (malloc(size))
#endif
```

Conditional compilation should generally be on a feature-by-feature basis. Machine or operating system dependencies should be avoided in most cases.

```
#ifdef BSD4
    long t = time((long *)NULL);
#endif
```

The preceding code is poor for two reasons: there may be 4BSD systems for which there is a better choice, and there may be non-4BSD systems for which the above *is* the best code. Instead, use `define` symbols such as `TIME_LONG` and `TIME_STRUCT` and define the appropriate one in a configuration file such as `config.h`.

## Debugging

“C Code. C code run. Run, code, run... PLEASE!!!” -- Barbara Tongue

If you use enums, the first enum constant should have a non-zero value, or the first constant should indicate an error.

```
enum {
    e_state_err,
    e_state_start,
    e_state_normal,
```

```

e_state_end
} State;

enum
{
e_val_new=1,
e_val_normal,
e_val_dying,
e_val_dead
} Value;

```

Uninitialized values will then often “catch themselves”.

Check for error return values, even from functions that “can't” fail. Consider that close() and fclose() can and do fail, even when all prior file operations have succeeded. Write your own functions or use standard error functions like *perror* so that they test for errors and return error values or abort the program in a well-defined way. Include a lot of debugging and error-checking code and leave most of it in the finished product. Check even for “impossible” errors – may be assert on such errors.

Use the assert facility to insist that each function is being passed well-defined values, and that intermediate results are well-formed.

Build in the debug code using as few #ifdefs as possible. For instance, if mm\_malloc is a debugging memory allocator, then MALLOC will select the appropriate allocator, avoids littering the code with #ifdefs, and makes clear the difference between allocation calls being debugged and extra memory that is allocated only during debugging.

```

#ifndef DEBUG
#define MALLOC(size) (mm_malloc(size))
#else
#define MALLOC(size) (malloc(size))
#endif

```

Check bounds even on things that “can't” overflow. A function that writes on to variable-sized storage should take an argument maxsize that is the size of the destination. If there are times when the size of the destination is unknown, some 'magic' value of maxsize should mean “no bounds checks”. When bound checks fail, make sure that the function does something useful such as abort or return an error status.

```

/*
*   INPUT:  A null-terminated source string `src' to copy from and
*   a `dest' string to copy to.      `maxsize' is the size of `dest'
*   or UINT_MAX if the size is not known. `src' and `dest' must
*   both be shorter than UINT_MAX, and `src' must be no longer than
*   `dest'.

```

```

*      OUTPUT: The address of `dest' is modified even when NULL if the copy fails.
*      `dest' is modified even when NULL if the copy fails.
*/
char* copy(dest, maxsize, src)
char *dest, *src;
maxsize;
{
    char *dp = dest;
    while ((maxsize-- > 0) && (*dp++ = *src++))
    {
        if (*dp == '\0')
            return (dest);
    }
    return (NULL);
}

```

In all, remember that a program that produces wrong answers twice as fast is infinitely slower. The same is true of programs that crash occasionally or clobber valid data.

## Portability

“C combines the power of assembler with the portability of assembler.”  
-- Anonymous, alluding to Bill Thacker.

Portable means that a source file can be compiled and executed on different machines with the only change being the inclusion of possibly different header files and the use of different compiler flags. The header files will contain #defines and typedefs that may vary from machine to machine. In general, a new “machine” is different hardware, a different operating system, a different compiler, or any combination of these. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

- Write portable code first, worry about detail optimizations only on machines where they prove necessary. Optimized code is often obscure. Optimizations for one machine may produce worse code on another. Document performance hacks and localize them as much as possible. Documentation should explain *how* it works and *why* it was needed (e.g., “loop executes 6 zillion times”).
- Recognize that some things are inherently non-portable. Examples are code to deal with particular hardware registers such as the program status word, and code that is designed to support a particular piece of hardware, such as an assembler or I/O driver. Even in these cases there are many routines and data organizations that can be made machine independent.
- Organize source files so that the machine-independent code and the machine-dependent code are in separate files. Then if the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed. Comment the machine dependence in the headers of the appropriate files.

- Any behavior that is described as “implementation defined” should be treated as a machine (compiler) dependency. Assume that the compiler or hardware does it some completely screwy way.
- Pay attention to word sizes. Objects may be non-intuitive sizes., Pointers are not always the same size as *ints*, the same size as each other, or freely interconvertible.
- The void\* type is guaranteed to have enough bits of precision to hold a pointer to any data object. The void(\*)() type is guaranteed to be able to hold a pointer to any function. Use these types when you need a generic pointer. (Use char\* and char(\*)(), respectively, in older compilers). Be sure to cast pointers back to the correct type before using them.
- Even when, say, an int\* and a char\* are the same *size*, they may have different *formats*. For example, the following will fail on some machines that have sizeof(int\*) equal to sizeof(char\*). The code fails because free expects a char\* and gets passed an int\* .
- int \*p = (int \*) malloc(sizeof(int));
- free (p);Check the validity
- Note that the *size* of an object does not guarantee the *precision* of that object.
- The integer *constant* zero may be cast to any pointer type. The resulting pointer is called a *null pointer* for that type, and is different from any other pointer of that type. A null pointer always compares equal to the constant zero. A null pointer might *not* compare equal with a variable that has the value zero. Null pointers are *not* always stored with all bits zero. Null pointers for two different types are sometimes different. A null pointer of one type cast in to a pointer of another type will be cast in to the null pointer for that second type.Never compare pointers with 0, rather use NULL.
- On ANSI compilers, when two pointers of the same type access the same storage, they will compare as equal. When non-zero integer constants are cast to pointer types, they may become identical to other pointers. On non-ANSI compilers, pointers that access the same storage may compare as different. The following two pointers, for instance, may or may not compare equal, and they may or may not access the same storage.
- ((int \*) ((int \*) 3 )) 2 )
- If you need 'magic' pointers other than NULL, either allocate some storage or treat the pointer as a machine dependence.
- extern int x\_int\_dummy; /\* in x.c \*/  

```
#define X_FAIL (NULL)
#define X_BUSY (&x_int_dummy)
#define X_FAIL (NULL)
```
- #define X\_BUSY MD\_PTR1 /\* MD\_PTR1 from "machdep.h" \*/
- Floating-point numbers have both a *precision* and a *range*. These are independent of the size of the object. Thus, overflow (underflow) for a 32-bit floating-point number will happen at different values on different machines. Also, 4.9 times 5.1 will yield two different numbers on two different machines. Differences in rounding and truncation can give surprisingly different answers.
- Watch out for signed characters' default qualifier: signed vs unsigned – it is compiler dependent. Use explicitly, if needed.
- Code that takes advantage of the two's complement representation of numbers on most machines should not be used. Optimizations that replace arithmetic operations with equivalent shifting

operations are particularly suspect. If absolutely necessary, machine-dependent code should be `#ifdef'fed` or operations should be performed by `#ifdef'fed` macros. You should weigh the time savings with the potential for obscure and difficult bugs when your code is moved.

- In general, if the word size or value range is important, use the `typedef` “sized” types. Large programs should have a available in central header file which supplies `typedefs` for commonly-used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code.
- Data *alignment* is also important.
- There may be unused holes in structures. Suspect unions used for type cheating. Specifically, a value should not be stored as one type and retrieved as another. An explicit tag field for unions may be useful.
- Different compilers use different conventions for returning structures. This causes a problem when libraries return structure values to code compiled with a different compiler. Do not return structures rather return `Sstructure` pointers are not a problem.
- Do not make assumptions about the parameter passing mechanism. especially pointer sizes and parameter evaluation order, size, etc. The following code, for instance, is *very* nonportable.

- |      |              |                    |                             |
|------|--------------|--------------------|-----------------------------|
| c    | =            | emertxe(getchar(), | getchar());                 |
|      |              |                    |                             |
| char | emertxe(char | c1,                | char                        |
| {    |              |                    | c2,                         |
|      | char         | bar                | char                        |
|      |              | =                  | *(&c1 + 1);                 |
|      |              |                    | /* often won't return c2 */ |
| }    |              |                    |                             |

- This example has lots of problems. The stack may grow up or down (indeed, there need not even be a stack!). Parameters may be widened when they are passed, so a `char` might be passed as an `int`, for instance. Arguments may be pushed left-to-right, right-to-left, in arbitrary order, or passed in registers (not pushed at all). The order of evaluation may differ from the order in which they are pushed. One compiler may use several (incompatible) calling conventions.
- On some machines, the null character pointer (`((char *)0)`) is treated the same way as a pointer to a null string. Do *not* depend on this.
- Do not modify string constants (see `emertxetnote 7`). One particularly notorious (bad) example is
- |   |   |                          |  |
|---|---|--------------------------|--|
| s | = | "/dev/tty??";            |  |
|   |   |                          |  |
|   |   | strcpy(&s[8], ttychars); |  |
- The address space may have holes. Simply *computing* the address of an unallocated element in an array (before or after the actual storage of the array) may crash the program. In ANSI C, a pointer into an array of objects may legally point to the first element after the end of the array; this is usually safe in older implementations. This “outside” pointer may not be dereferenced.
- Only the `==` and `!=` comparisons are defined for all pointers of a given type. It is only portable to use `<<`, `<=`, `>`, or `>=` to compare pointers when they both point in to (or to the first element after) the same array. It is likewise only portable to use arithmetic operators on pointers that both point into the same array or the first element afterwards.

- Word size also affects shifts and masks. The following code will clear only the three rightmost bits of an int on *some* 68000s. On other machines it will also clear the upper two bytes. `x &= 0177770`. Use instead `x &= ~07` which works properly on all machines. Bitfields do not have these problems.
- Side effects within expressions can result in code whose semantics are compiler-dependent, since C's order of evaluation is explicitly undefined in most places. Notorious examples include the following.
- `a[i] = b[i++];`

In the above example, we know only that the subscript into `b` has not been incremented. The index into `a` could be the value of `i` either before or after the increment.

- `struct` `bar_t`  
`{`  
`struct` `bar_t`  
`}` `*next;`  
`Bar_t;`  
`bar->next = bar = tmp;`

- In the second example, the address of `bar->next` may be computed before the value is assigned to `bar`.
- `bar = bar->next = tmp;`
- In the third example, `bar` can be assigned before `bar->next`. Although this *appears* to violate the rule that “assignment proceeds right-to-left”, it is a legal interpretation. Consider the following example:

- `long` `i;`  
`short` `a[N];`  
`i` `=`  
`i = a[i] = new;` `old`

- The value that `i` is assigned must be a value that is typed as if assignment proceeded right-to-left. However, `i` may be assigned the value “(long)(short)`new`” before `a[i]` is assigned to. Compilers do differ. Check the validity
- Be suspicious of numeric values appearing in the code (“magic numbers”).
- Avoid preprocessor tricks. Tricks such as using `/* */` for token pasting and macros that rely on argument string expansion will break reliably.
- `#define emertxe(string)` `(printf("string = %s", (string)))`  
`...`  
`emertxe(filename);`
- Will only sometimes be expanded to
- `(printf("filename = %s", (filename)))`
- Be aware, however, that tricky preprocessors may cause macros to break *accidentally* on some machines. Consider the following two versions of a macro.

- ```
#define LOOKUP(chr)      (a['c' + (chr)]) /* Works as intended. */
#define LOOKUP(c)          (a['c' + (c)]) /* Sometimes breaks. */
```
- The second version of LOOKUP can be expanded in two different ways and will cause code to break mysteriously. Check the validity
- Use *splint* when it is available. It is a valuable tool for finding machine-dependent constructs as well as other inconsistencies or program bugs that pass the compiler. If your compiler has switches to turn on warnings, use them (-Wallw option to *gcc*). Treat all warnings as errors (-Werror), so that they are not ignored.
- Suspect labels inside blocks with the associated switch or goto outside the block.
- Wherever the type is in doubt, parameters should be cast to the appropriate type. Always cast NULL when it appears in non-prototyped function calls. Do not use function calls as a place to do type cheating. C has confusing promotion rules, so be careful. For example, if a function expects a 32-bit long and it is passed a 16-bit int the stack can get misaligned, the value can get promoted wrong, etc.
- Use explicit casts when doing arithmetic that mixes signed and unsigned values, to make the desired result, explicit.
- The inter-procedural goto, longjmp, should be used with caution. Many implementations “forget” to restore values in registers. Declare critical values as volatile if you can or comment them as VOLATILE.
- Beware of compiler extensions. If used, document and consider them as machine dependencies.
- A program cannot generally execute code in the data segment or write into the code segment. Even when it can, there is no guarantee that it can do so reliably.

## Prototypes

Function prototypes should be used to make code more robust and to make it run faster. Unfortunately, the prototyped *declaration*

```
extern void bork(char c);
```

is incompatible with the *definition*

<pre>void char ...</pre>	<pre>bork(c) c;</pre>
----------------------------------	---------------------------

The prototype says that c is to be passed as the most natural type for the machine, possibly a byte. The non-prototyped (backwards-compatible) definition implies that c is always passed as an int. If a function has promotable parameters then the caller and callee must be compiled identically. Either both must use function prototypes or neither can use prototypes. The problem can be avoided if parameters are promoted when the program is designed. For example, bork can be defined to take an int parameter.

The above declaration works if the definition is prototyped.

```
void bork(char c)
{
...
}
```

Unfortunately, the prototyped syntax will cause non-ANSI compilers to reject the program.

It *is* easy to write external declarations that work with both prototyping and with older compilers

```
#if __STDC__
#define PROTO(x) x
#else
#define PROTO(x) ()
#endif

extern char **ncopies PROTO((char *s, short times));
```

Note that PROTO must be used with *double* parentheses.

In the end, it may be best to write in only one style (e.g., with prototypes). When a non-prototyped version is needed, it is generated using an automatic conversion tool.

## Pragmas

Pragmas are used to introduce machine-dependent code in a controlled way. Obviously, pragmas should be treated as machine dependencies. Unfortunately, the syntax of ANSI pragmas makes it impossible to isolate them in machine-dependent headers.

Pragmas are of two classes. *Optimizations* may safely be ignored. Pragmas that change the system behavior (“required pragmas”) may not. Required pragmas should be #ifdeffed so that compilation will abort if no pragma is selected.

Two compilers may use a given pragma in two very different ways. For instance, one compiler may use haggis to signal an optimization. Another might use it to indicate that a given statement, if reached, should terminate the program. Thus, when pragmas are used, they must always be enclosed in machine-dependent #ifdefs. Pragmas must always be #ifdefed out for non-ANSI compilers. Be sure to indent the `#' character on the #pragma, as older preprocessors will halt on it otherwise.

```
#if defined(__STDC__)
    #pragma
#endif
```

*“The '#pragma' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '#pragma' first attempts to run the game ‘rogue’; if that fails, it tries to run the game ‘hack’; if that fails, it tries to run GNU Emacs displaying the Tower of Hanoi; if that fails, it reports a fatal error. In any case, preprocessing does not continue.”*  
-- Manual for the GNU C preprocessor for GNU CC 1.34.

## Special Considerations

This section contains some miscellaneous do's and don'ts.

- Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.
- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as  $\leq$  or  $\geq$ , never use an exact comparison ( $=$  or  $\neq$ ).
- Compilers have bugs. Common trouble spots include structure assignment and bitfields. You cannot generally predict which bugs a compiler has. You *could* write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write "around" compiler bugs only when you are *forced* to use a particular buggy compiler.
- Do not rely on automatic beautifiers (indent). Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer. (In other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds.)
- Accidental omission of the second  $=$  of the logical compare is a problem. Use explicit tests. Avoid assignment with implicit test.
- `abool = bbool;`
- `if (abool)`
- `{`
  - ...
- When embedded assignment *is* used, make the test explicit so that it doesn't get "fixed" later.
- `while ((abool = bbool) != FALSE)`
- `{`
  - ...
- `while (abool = bbool)`
- `{`
  - ... /\* VALUSED \*/
- `while (abool = bbool, abool)`
- `{`

- ...
- Explicitly comment variables that are changed out of the normal control flow, or other code that is likely to break during maintenance.
- Modern compilers will put variables in registers automatically. Use the register sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register and mark the rest as REGISTER. The latter can be #defined to register on those machines with many registers.

## Project-Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. The following issues are some of those that should be addressed by each project program administration group.

- What additional naming conventions should be followed? In particular, systematic prefix conventions for functional grouping of global data and also for structure or union member names can be useful.
- What kind of include file organization is appropriate for the project's particular data hierarchy?
- What procedures should be established for reviewing *splint* complaints? A tolerance level needs to be established in concert with the *splint* options to prevent unimportant complaints from hiding complaints about real bugs or inconsistencies.
- If a project establishes its own archive libraries, it should plan on supplying a lint library file [2] to the system administrators. The lint library file allows *splint* to check for compatible use of library functions.
- What kind of revision control needs to be used?

# Functions



# Advanced C

## Functions - What?



An activity that is natural to or the purpose of a person or thing.

"bridges perform the function of providing access across water"

A relation or expression involving one or more variables.

"the function  $(bx + c)$ "

Source: Google

- In programming languages it can be something which performs a specific service
- Generally a function has 3 properties
  - Takes Input
  - Perform Operation
  - Generate Output

# Advanced C

## Functions - What?



$$f(x) = x + 1$$



$$x + 1$$



$$2 + 1$$



# Advanced C

## Functions - How to write



### Syntax

```
return_data_type function_name(arg_1, arg_2, ..., arg_n)
{
    /* Function Body */
}
```

List of function parameters

### Example

Return data type as int

First parameter with int type

Second parameter with int type

```
int foo(int arg_1, int arg_2)
{
}
```

# Advanced C

## Functions - How to write



$$y = x + 1$$

### Example

```
int foo(int x)
{
    int ret;

    ret = x + 1;

    return ret;
}
```

Return from function

# Advanced C

## Functions - How to call

### 001\_example.c

```
#include <stdio.h>

int main()
{
    int x, y;

    x = 2;
    y = foo(x);
    printf("y is %d\n", y);

    return 0;
}
```

The function call

```
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

# Advanced C

## Functions - Why?



- **Re usability**
  - Functions can be stored in library & re-used
  - When some specific code is to be used more than once, at different places, functions avoids repetition of the code.
- **Divide & Conquer**
  - A big & difficult problem can be divided into smaller sub-problems and solved using divide & conquer technique
- **Modularity** can be achieved.
- Code can be easily **understandable & modifiable**.
- Functions are easy to **debug & test**.
- One can suppress, how the task is done inside the function, which is called **Abstraction**



# Advanced C

## Functions - A complete look



### 002\_example.c

```
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 20;
    int sum = 0;
    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}

int add_numbers(int num1, int num2) {
    int sum = 0;
    sum = num1 + num2;
    return sum;
}
```

The main function

The function call

Actual arguments

Return type

Formal arguments

operation

Return result from function and exit

# Advanced C

## Functions - Ignoring return value



### 003\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    add_numbers(num1, num2); ←
    printf("Sum is %d\n", sum);

    return 0;
}

int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Ignored the return from function  
In C, it is up to the programmer to capture or ignore the return value

# Advanced C

## Functions - DIY



- Write a function to calculate square a number
  - $y = x * x$
- Write a function to convert temperature given in degree Fahrenheit to degree Celsius
  - $C = 5/9 * (F - 32)$
- Write a program to check if a given number is even or odd. Function should return TRUE or FALSE



# Advanced C

## Function and the Stack

### Linux OS



The Linux OS is divided into two major sections

- User Space
- Kernel Space

The user programs cannot access the kernel space. If done will lead to segmentation violation

Let us concentrate on the user space section here

# Advanced C

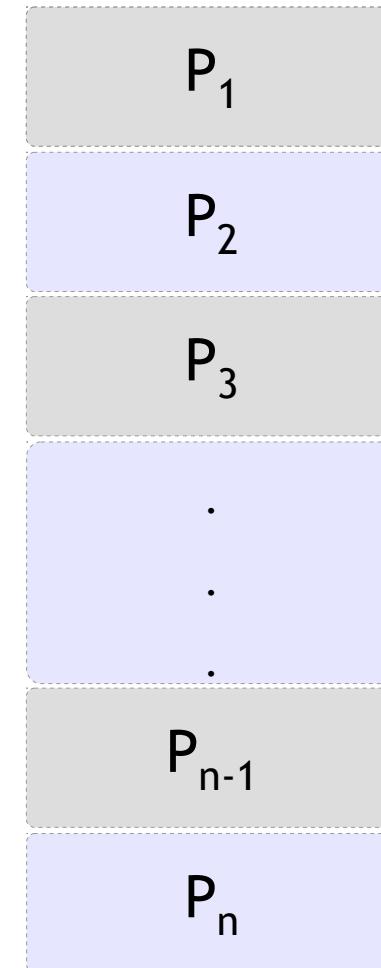
## Function and the Stack



Linux OS



User Space



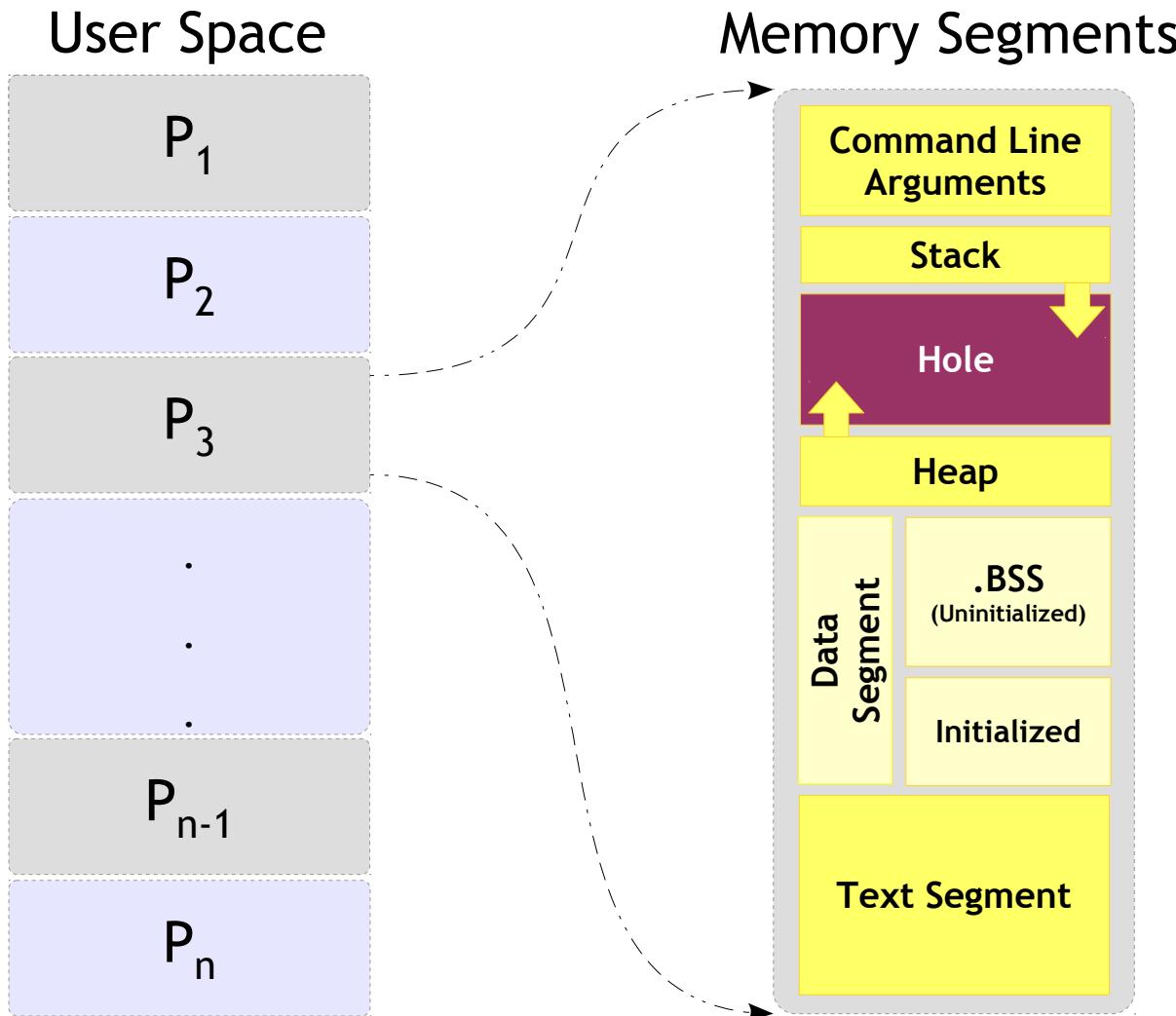
The User space contains many processes

Every process will be scheduled by the kernel

Each process will have its memory layout discussed in next slide

# Advanced C

## Function and the Stack



The memory segment of a program contains four major areas.

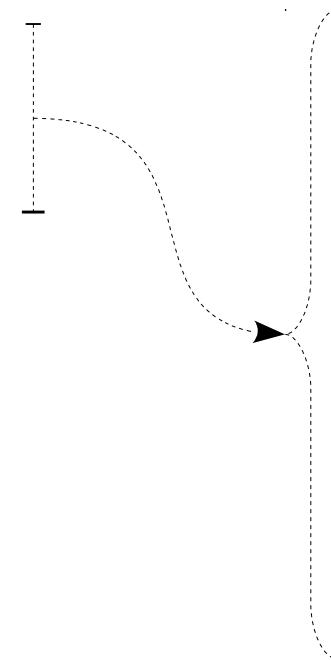
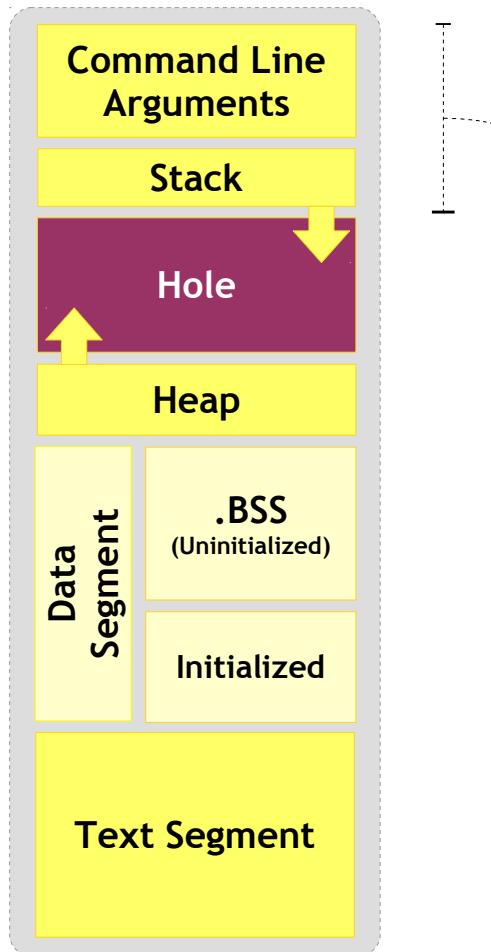
- Text Segment
- Stack
- Data Segment
- Heap

# Advanced C

## Function and the Stack



### Memory Segments



Adjoins the heap area and grow in opposite area of heap when stack and heap pointer meet (Memory Exhausted)

Typically loaded at the higher part of memory

A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack

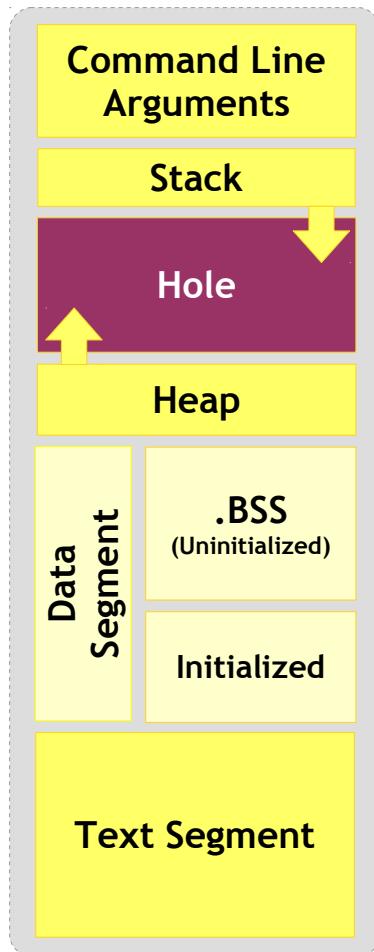
The set of values pushed for one function call is termed a “stack frame”

# Advanced C

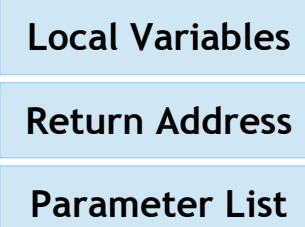
## Function and the Stack



### Memory Segments



### Stack Frame



A stack frame contain at least of a return address

# Advanced C

## Function and the Stack - Stack Frames

### 002\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}

int add_numbers(int n1, int n2)
{
    int s = 0;

    s = n1 + n2;

    return s;
}
```

### Stack Frame

num1 = 10  
num2 = 20  
sum = 0

Return Address to the caller

s = 0

Return Address to the main()

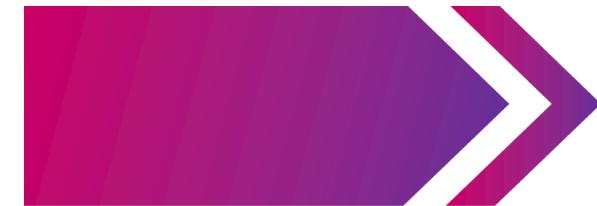
n1 = 10  
n2 = 20

main()

add\_numbers()

# Advanced C

## Functions - Parameter Passing Types



Pass by Value	Pass by reference
<ul style="list-style-type: none"><li>This method copies the actual value of an argument into the formal parameter of the function.</li><li>In this case, changes made to the parameter inside the function have no effect on the actual argument.</li></ul>	<ul style="list-style-type: none"><li>This method copies the address of an argument into the formal parameter.</li><li>Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.</li></ul>

# Advanced C

## Functions - Pass by Value

### 002\_example.c

```
#include <stdio.h>

int add_numbers(int num1, int num2);

int main()
{
    int num1 = 10, num2 = 20, sum;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

# Advanced C

## Functions - Pass by Value

### 004\_example.c

```
#include <stdio.h>

void modify(int num1)
{
    num1 = num1 + 1;
}

int main()
{
    int num1 = 10;

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

    modify(num1);

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

    return 0;
}
```

# Advanced C

## Functions - Pass by Value



Are you sure you understood the previous problem?

Are you sure you are ready to proceed further?

Do you know the prerequisite to proceed further?

If no let's get it cleared



# Advanced C

## Functions - Pass by Reference



### 005\_example.c

```
#include <stdio.h>

void modify(int *iptr)
{
    *iptr = *iptr + 1;
}

int main()
{
    int num = 10;

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

    modify(&num);

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

    return 0;
}
```



# Advanced C

## Functions - Pass by Reference

### 005\_example.c

```
#include <stdio.h>

void modify(int *iptr)
{
    *iptr = *iptr + 1;
}

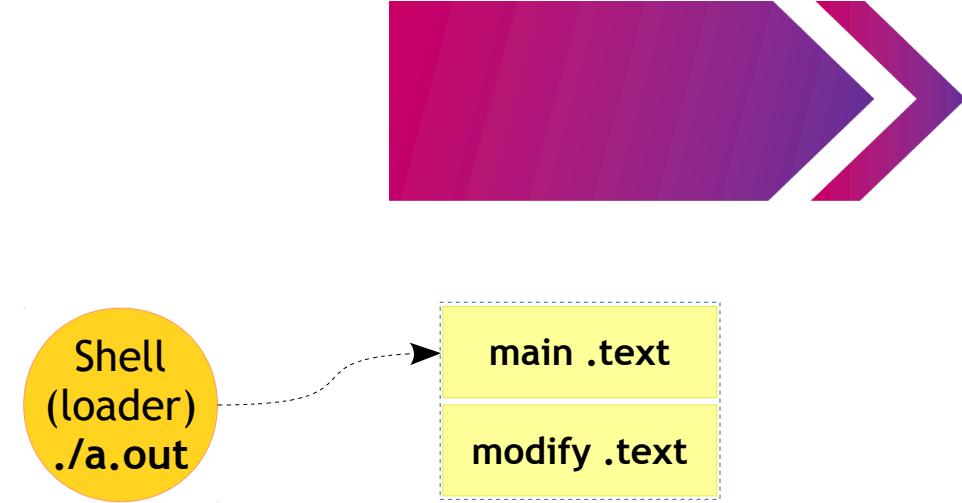
int main()
{
    int num = 10;

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

    modify(&num);

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

    return 0;
}
```



# Advanced C

## Functions - Pass by Reference

005\_example.c

```
#include <stdio.h>

void modify(int *iptr)
{
    *iptr = *iptr + 1;
}

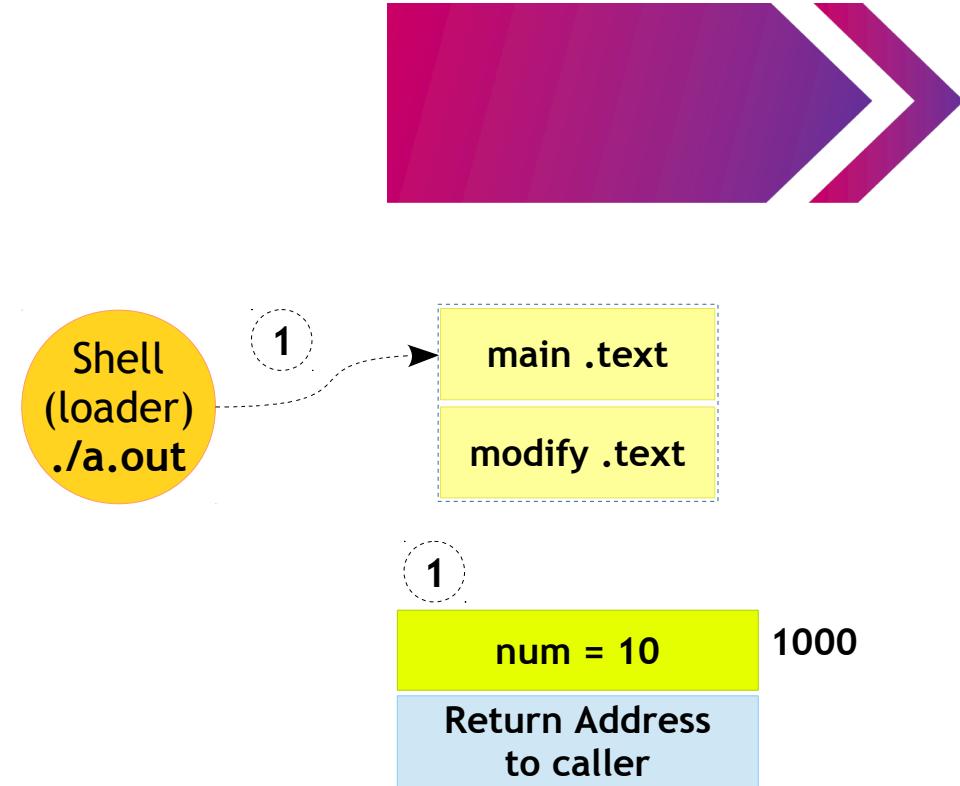
int main()
{
    int num = 10;

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

    modify(&num);

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

    return 0;
}
```



# Advanced C

## Functions - Pass by Reference

### 005\_example.c

```
#include <stdio.h>

void modify(int *iptr)
{
    *iptr = *iptr + 1;
}

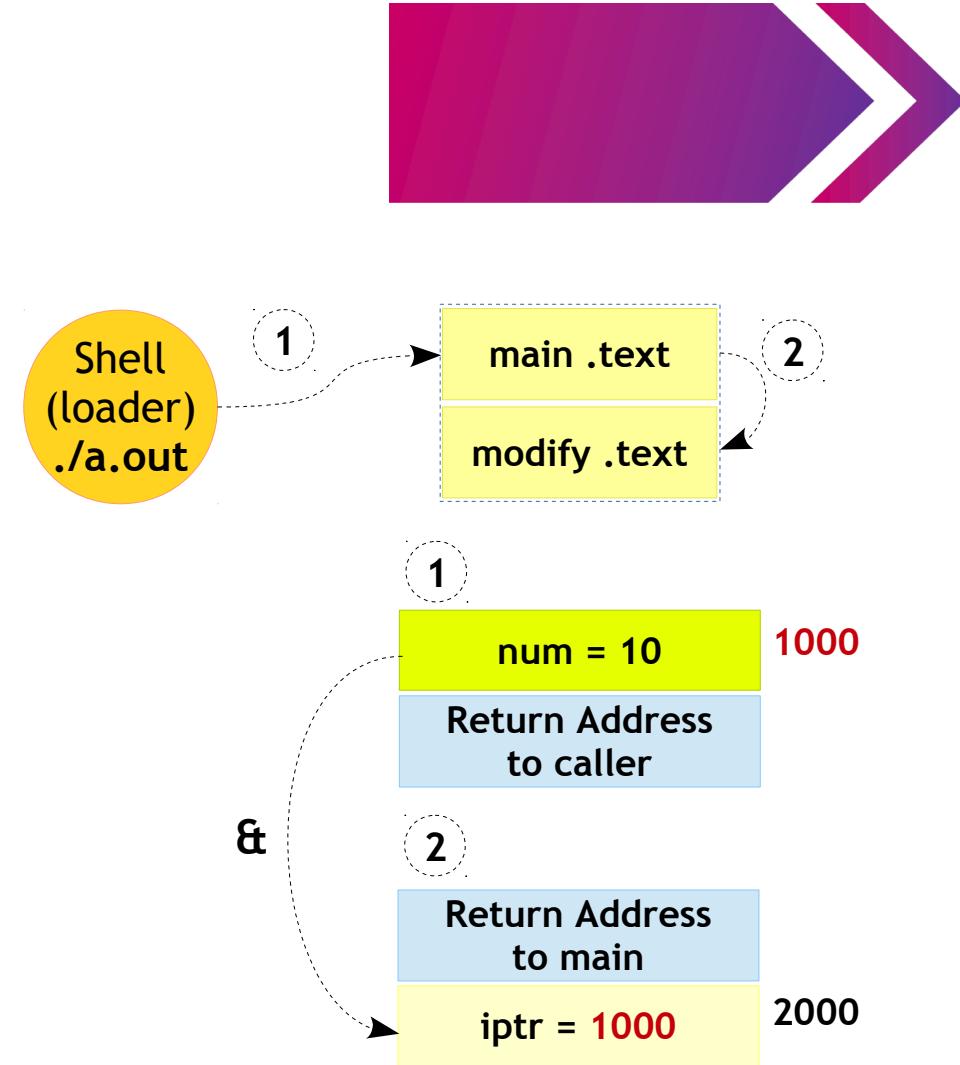
int main()
{
    int num = 10;

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

→ modify(&num);

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

    return 0;
}
```



# Advanced C

## Functions - Pass by Reference

005\_example.c

```
#include <stdio.h>

void modify(int *iptr)
{
    *iptr = *iptr + 1;
}

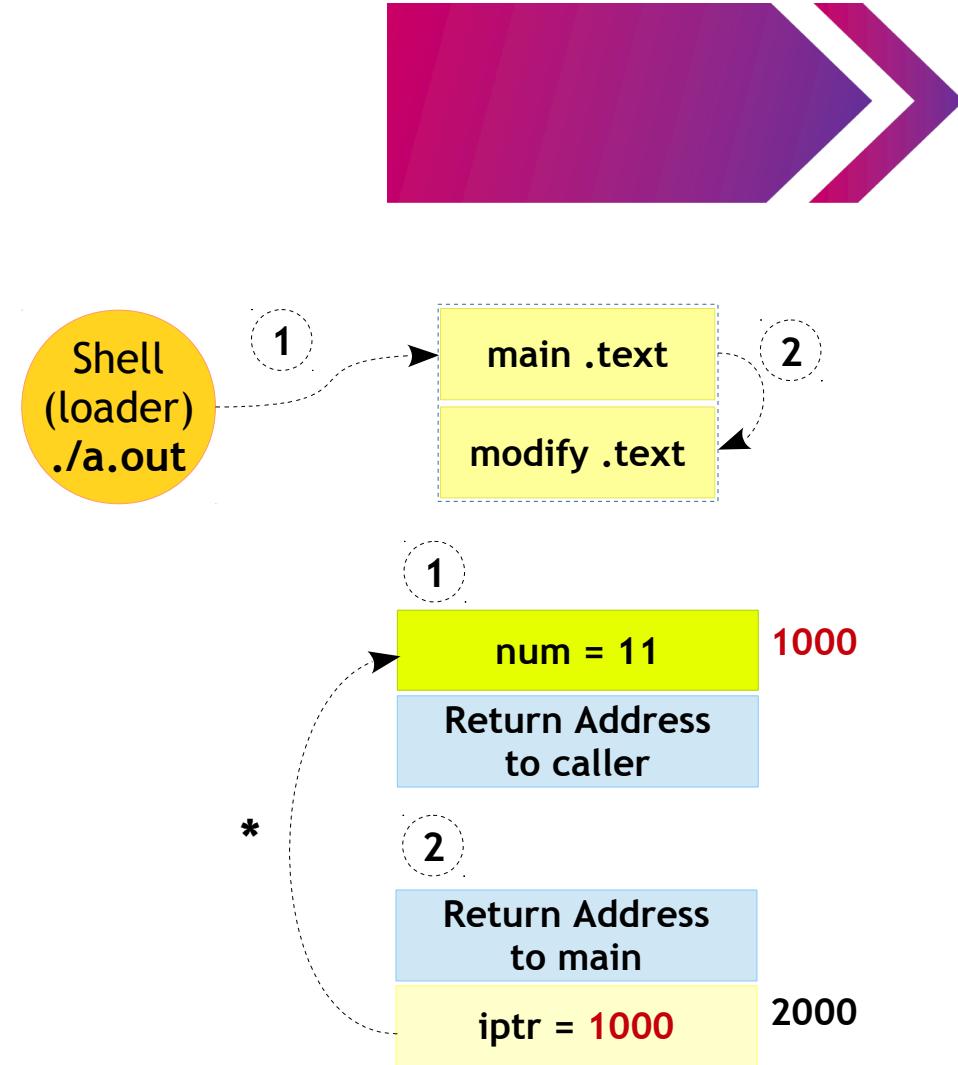
int main()
{
    int num = 10;

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

    modify(&num);

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

    return 0;
}
```



# Advanced C

## Functions - Pass by Reference - Advantages



- Return more than one value from a function
- Copy of the argument is not made, making it fast, even when used with large variables like arrays etc.
- Saving stack space if argument variables are larger (example - user defined data types)



# Advanced C

## Functions - DIY (pass-by-reference)



- Write a program to find the square and cube of a number
- Write a program to swap two numbers
- Write a program to find the sum and product of 2 numbers
- Write a program to find the square of a number



# Advanced C

## Functions - Prototype

- Need of function prototype
- Implicit int rule

# Advanced C

## Functions - Passing Array



- As mentioned in previous slide passing an array to function can be faster
- But before you proceed further it is expected you are familiar with some pointer rules
- If you are OK with your concepts proceed further, else please **know the rules first**



# Advanced C

## Functions - Passing Array

### 006\_example.c

```
#include <stdio.h>

void print_array(int array[]);

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

    print_array(array);

    return 0;
}

void print_array(int array[])
{
    int iter;

    for (iter = 0; iter < 5; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

# Advanced C

## Functions - Passing Array

### 007\_example.c

```
#include <stdio.h>

void print_array(int *array);

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

    print_array(array);

    return 0;
}

void print_array(int *array)
{
    int iter;

    for (iter = 0; iter < 5; iter++)
    {
        printf("Index %d has Element %d\n", iter, *array);
        array++;
    }
}
```

# Advanced C

## Functions - Passing Array

### 008\_example.c

```
#include <stdio.h>

void print_array(int *array, int size);

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

    print_array(array, 5);

    return 0;
}

void print_array(int *array, int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, *array++);
    }
}
```

# Advanced C

## Functions - Returning Array

### 009\_example.c

```
#include <stdio.h>

int *modify_array(int *array, int size);
void print_array(int array[], int size);

int main()
{
    int array[5] = {10, 20, 30, 40, 50};
    int *new_array_val;

    new_array_val = modify_array(array, 5);
    print_array(new_array_val, 5);

    return 0;
}
```

```
void print_array(int array[], int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

```
int *modify_array(int *array, int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        *(array + iter) += 10;
    }

    return array;
}
```

# Advanced C

## Functions - Returning Array

### 010\_example.c

```
#include <stdio.h>

int *return_array(void);
void print_array(int *array, int size);

int main()
{
    int *array_val;

    array_val = return_array();
    print_array(array_val, 5);

    return 0;
}
```

```
int *return_array(void)
{
    static int array[5] = {10, 20, 30, 40, 50};

    return array;
}
```

```
void print_array(int *array, int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

# Advanced C

## Functions - DIY



- Write a program to find the average of 5 array elements using function
- Write a program to square each element of array which has 5 elements



# Advanced C

## Functions - Return Type

- Local return
- Void return

# Recursive Function



# Advanced C Functions



# Advanced C

## Functions - Recursive



- Recursion is the process of repeating items in a self-similar way
- In programming a function calling itself is called as recursive function
- Two steps

**Step 1:** Identification of base case

**Step 2:** Writing a recursive case



# Advanced C

## Functions - Recursive - Example



### 011\_example.c

```
#include <stdio.h>

/* Factorial of 3 numbers */

int factorial(int number)
{
    if (number <= 1) /* Base Case */
    {
        return 1;
    }
    else /* Recursive Case */
    {
        return number * factorial(number - 1);
    }
}

int main()
{
    int ret;

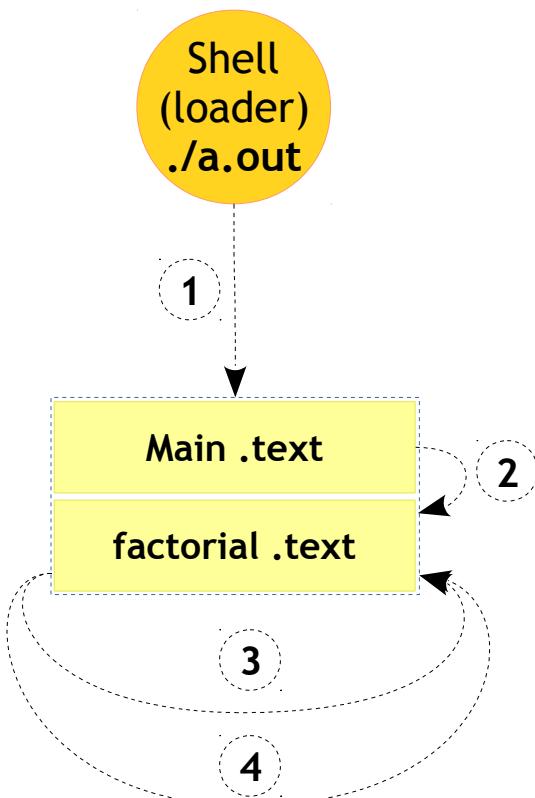
    ret = factorial(3);
    printf("Factorial of 3 is %d\n", ret);

    return 0;
}
```

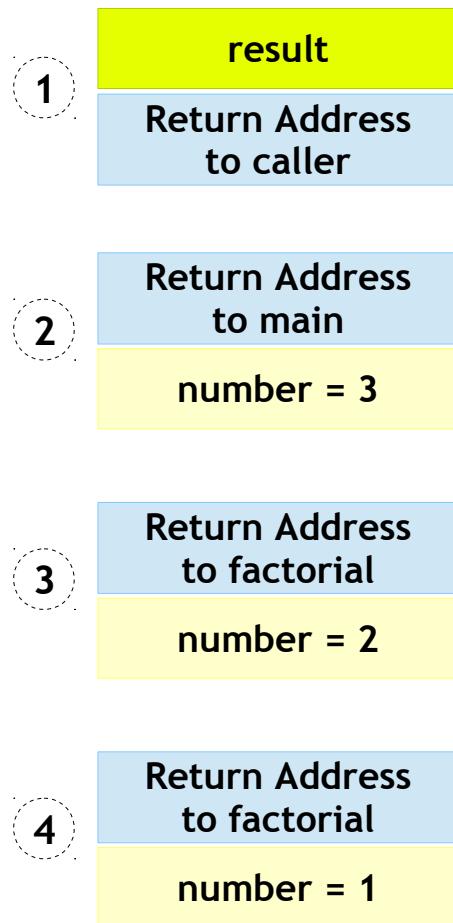
n	$!n$
0	1
1	1
2	2
3	6
4	24

# Embedded C

## Functions - Recursive - Example Flow



### Stack Frames



### Value with calls

factorial(3)

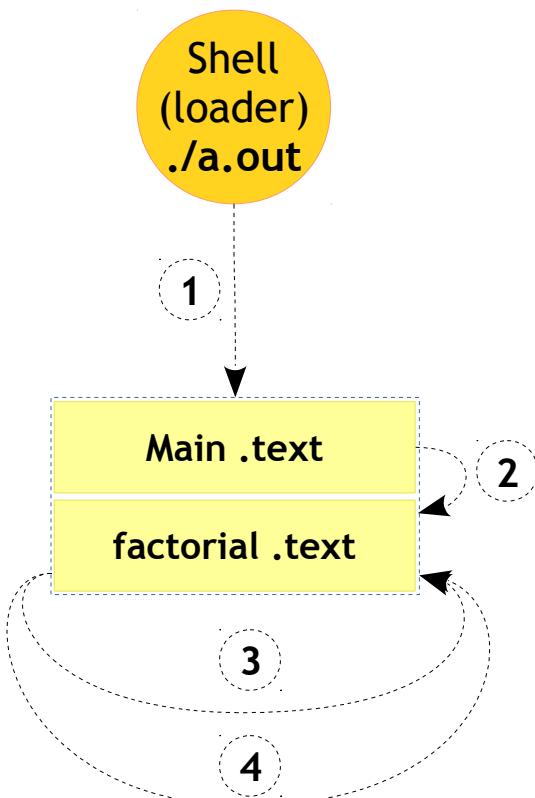
number != 1  
number \* factorial(number -1)  
3 \* factorial(3 -1)

number != 1  
number \* factorial(number -1)  
2 \* factorial(2 -1)

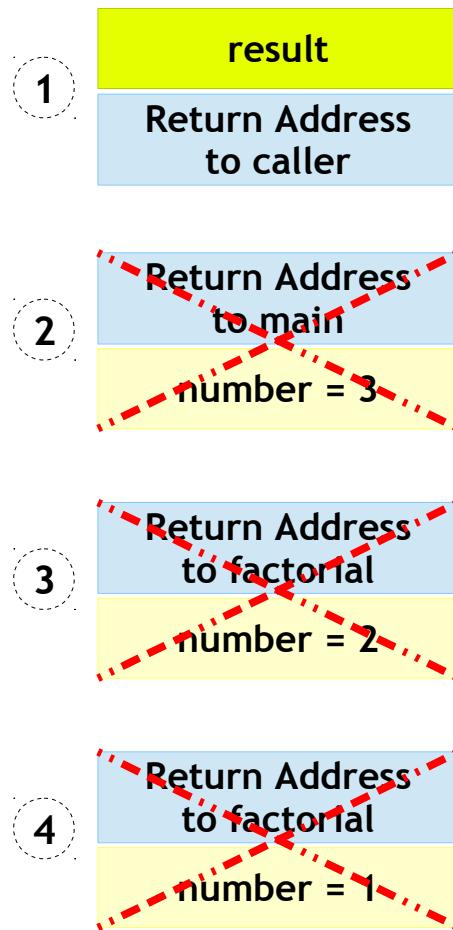
number == 1

# Embedded C

## Functions - Recursive - Example Flow



### Stack Frames



### Results with return

Gets 6 a value

Returns  $3 * 2$  to the caller

Returns  $2 * 1$  to the caller

returns 1 to the caller

# Advanced C

## Functions - DIY



- Write a program to find the sum of sequence of N from starting from 1



# Standard I/O Functions



## Functions

Function in general is nothing but a specific task. In C programming functions are defined as the set of instructions which are named under a specific name and do specific operations. There can be any number of user defined functions written apart from the main function. Main function has to be written only once and that will be called by the operating system. Other user defined functions can be called anywhere according to the requirements. Some of the built-in functions are printf, scanf, etc.

If we have to be writing a function to perform some task, the function definition should first have the data type of the value it might return which is called the return type followed by the name of the function and then the input types along with the names of the variables called the formal arguments.

Syntax: return\_type func\_name(Formal argument type name1, ...)

```
{  
}  
}
```

Eg: write a function to add two numbers.

The function will return an integer, it might take two inputs x and y. The function definition is as follows.

```
int add(int x, int y)  
{  
    int res;  
    res = x + y;  
    return res;  
}
```

To get this function executed, the function needs to be called somewhere in the program as shown below.

```
int main()  
{  
    int num1 = 10, num2 = 20, sum;  
    sum = add(num1, num2);  
    printf("Sum = %d\n", sum);
```

```

    return 0;
}

}

```

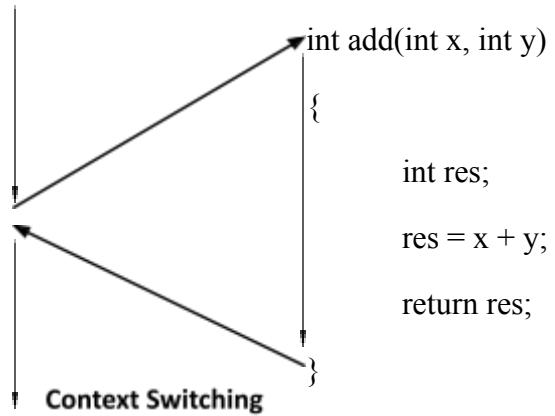
With respect to the add function here, the main function is called the caller function and the add function is called the called function.

Whenever a function call is done, control moves from function call to function definition, gets all the instructions executed and comes back to the place where the function was called. This is called context switching.

```

int main()
{
    int num1 = 10, num2 = 20, sum;
    sum = add(num1, num2);
    printf("Sum = %d\n", sum);
    return 0;
}

```



Advantages of writing a function are,

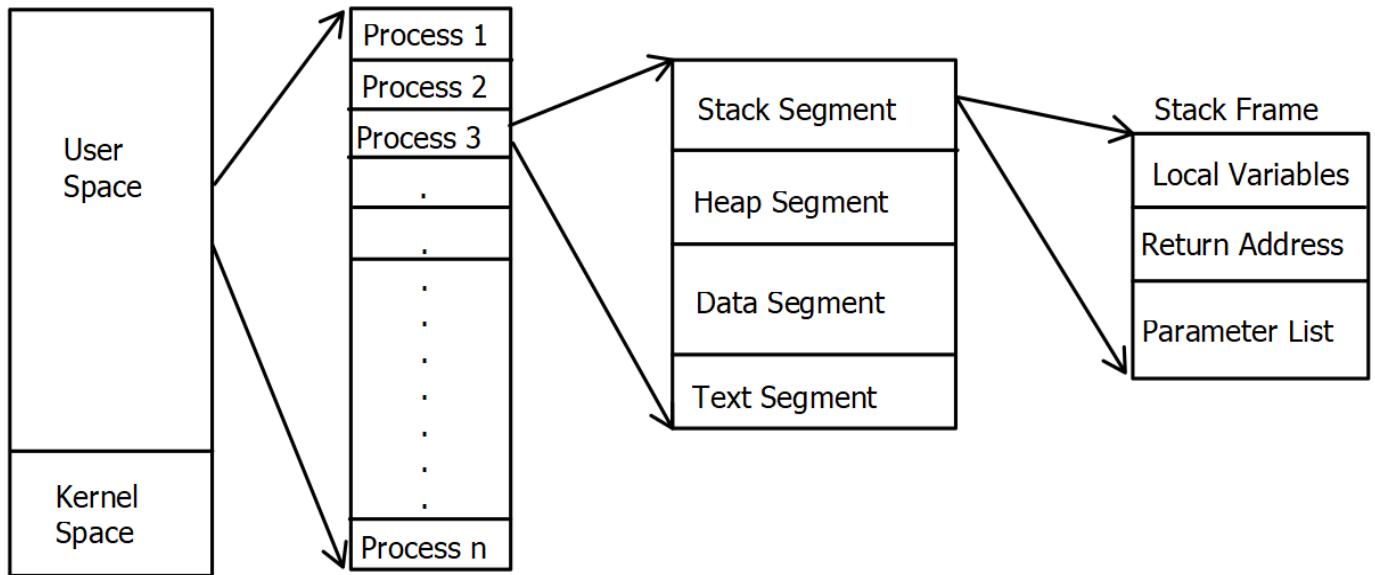
- \*To avoid repetitions i.e. to reuse the code whenever necessary.
- \*To achieve divide and conquer methodology.
- \*To achieve modularity and abstraction
- \*For easy debugging and testing

The major changes when a function is called are seen in the memory segment. The memory in any operating system is divided into 2 sections, the user space and kernel space. Any normal user cannot access the kernel space, doing so will lead to segmentation fault. All the normal users like us can only access user space. There can be 'n' number of processes being executed and each of them gets its turn for execution depending on the scheduler in the OS. If ./a.out process is getting executed, it gets certain bytes of memory allocated which is further divided into 4 segments. They are text segment, data segment, heap segment and stack segment.

Whenever a function is called, a stack frame gets created in the stack segment. Stack frame has 3 sections, local variables, return address and parameter list. Any variables which are declared or defined inside a function without any storage class gets the memory allocated in the local variables section. Return address section has the address of the caller function which helps the called function to get back to the place where it was called. Parameter list is that section

where the formal arguments i.e. the arguments which are written in ( ) gets the memory allocated.

Any stack frame created will at least have the return address in it. Once the statements of the function are executed and control is brought back to the place where it was called, the stack frame gets destroyed.



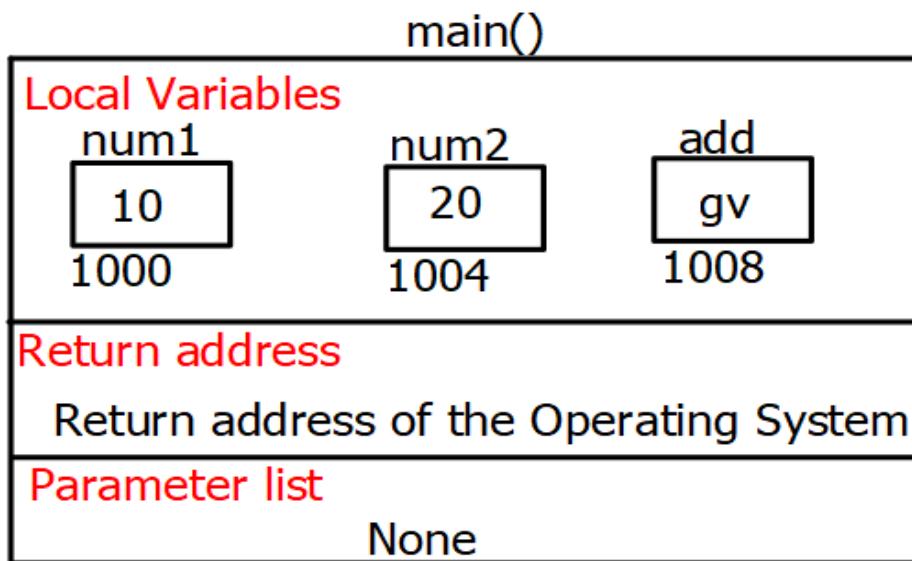
Consider an example shown below.

```
int sum(int x, int y)
{
    int ret;
    ret = x + y;
    return ret;
}

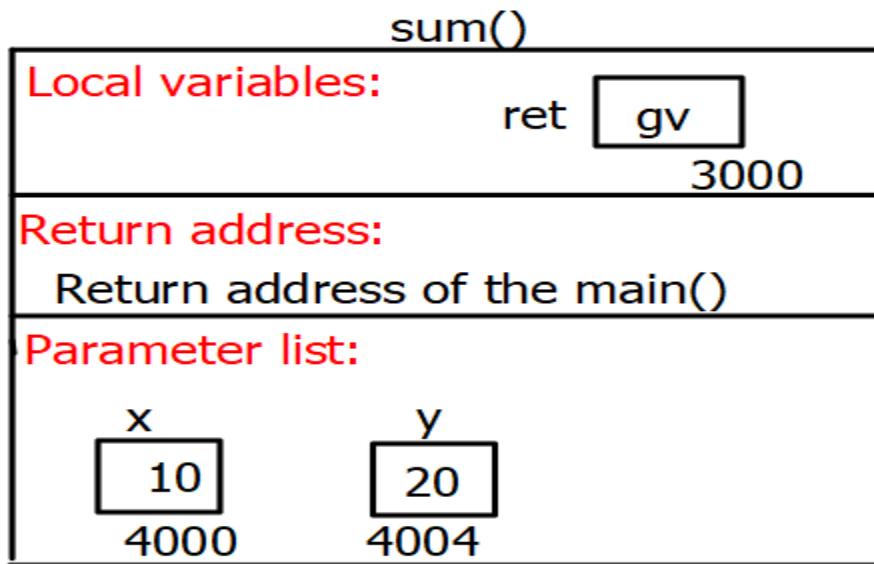
int main()
{
    int num1 = 10, num2 = 20, add;
    add = sum(num1, num2);
    printf("sum of %d and %d is %d\n", num1, num2, sum);
```

```
return 0;  
}
```

In the above given example, num1 and num2 in the main function are the actual arguments and x and y in the sum function are formal arguments. First after compiling the program, when the process gets to execute, the OS calls the main function. The moment main function is called, the corresponding stack frame gets created. Num1, num2 and add are the 3 local variables, return address will be the address of OS and there is no parameter list in this function.



After the stack frame is created, the instructions of the function start to execute. During the function call to the `sum` function, the values of `num1` and `num2` are passed which are then collected by the formal arguments `x` and `y`. Because the function is called, now a stack frame gets created for the function with `ret` in the local variable, return address of `main` function in return address and `x`, `y` in the parameter list.



Now the statements of the sum function are executed and ret get updated to 30. This value will be returned to the place where the sum function was called and the stack frame of the function gets destroyed. Once all the statements of the main function are executed, the stack frame of the main function also gets destroyed.

In the above example, the function call is done by passing the values of variables. This method of function call is called pass by value. There are some of the disadvantages of this method for which we need to do pass by reference method of function call.

Disadvantages of pass by value:

- \*Returning is compulsory to see the change done in the function.

- \*Returning more than one value from a function is not possible.

- \*Returning is compulsory to see the change done in the function.

Eg: void modify (int x)

```
{
```

```
    x = x + 1;
```

```
}
```

```
int main()
```

```
{
```

```
    int x= 10;
```

```

printf("Before modification, x = %d\n", x);
modify (x);
printf("After modification, x = %d\n", x);
return 0;
}

```

In the above example, x in the main function is 10 which is passed to the modify function. In the modify function, it is received by x and is changed to 11. After coming back to the main function, when x is printed, it still shows 10. This is because the change which was done in the modify function is to the copy of the variable which needs to be returned from the function. As the return was not done from the modify function, we do not get to see the change in the main function. Note that x in the main function and modify function are different. In the main function, it is a local variable and in the modify function, it is in the parameter list. Both are in different stack frames.

\*Returning more than one value from a function is not possible.

```

int modify(int x, int y)
{
    x = x + 1;
    y = y + 1;
    return x, y;
}

int main()
{
    int x = 10, y = 20;
    printf("Before modification: x = %d, y = %d\n", x, y);
    x , y = modify (x, y);
    printf("After modification: x = %d, y = %d\n", x, y);
    return 0;
}

```

In the above example, the modify function is trying to return more than one value from the function. But return x, y or return (x, y) both will return the value of y. In the main function,

the value is received by y only . In the statement x, y = modify (x, y); it will be seen by the compiler as x, (y = modify(x, y)); Hence the value will be received by y only.

In order to see the change in more than one value from a function, we need to go with the pass by reference method of function call.

```
int modify(int *ptr1, int *ptr2)
{
    *ptr1 = *ptr1 + 1;
    *ptr2 = *ptr2 + 1;
}

int main()
{
    int x = 10, y = 20;
    printf("Before modification: x = %d, y = %d\n", x, y);
    modify (&x, &y);
    printf("After modification: x = %d, y = %d\n", x, y);
    return 0;
}
```

main()	
Local variables	
x	10
y	20
1000	2000
Return address	
Return address of the OS	
Parameter list	
None	

modify()	
Local variables	
None	
Return address	
Return address of the main()	
Parameter list	
ptr1	1000
	3000
ptr2	2000
	4000

In the above example, when the modify function is called, the addresses of x and y are passed. Hence any change done using ptr will directly affect x and y.

Whenever the user defined functions are written after the main function, the compiler either throws a warning or an error. The compiler should always be made aware of the function's existence before the function call. This is done either by writing the function's prototype or by writing the function definition before the function call. The discrepancy is created because of the implicit int rule. If the compiler is not made aware of the function's existence, it assumes that the function takes inputs of type integer and returns the output of type integer. If the user defined function's definition matched with what compiler has assumed, it throws a warning saying implicit declaration ppf the function else if the function is accepting inputs of some other type or returning an output of some other type which happens to be a mismatch of what compiler has assumed, it throws an error.

Eg:

```
int main()
{
    int num;
    num = fun();
    printf("%d\n", num);
    return 0;
}

int fun()
{
    int x;
    return x + 10;
}
```

In the above example, the compiler will throw a warning as it is not aware of the function's existence before the function call. Compiler will assume that the function takes integer inputs and returns an integer. To avoid the warning, it is necessary to have the function prototype written.

Syntax of writing prototype is:

```
return_type func_name(input arguments type);
```

In the above example, function's prototype should look as follows,

```
int fun();
```

If a function is taking two arguments of type int and returning an output of type int, the prototype should be as follows:

```
int sum(int, int);
```

or

```
int sum(int x, int y);
```

Note that the ; termination is compulsory, arguments names are optional and can be anything.

Necessity of writing function's prototype are:

- \*to make the compiler aware of function's existence

- \*to make the compiler know the number of inputs and its type.

- \*to make the compiler know the output type

# Recursive Functions

- Recursion is the process of repeating items in a self-similar way.
- In C programming recursive functions are the one where a function is called by itself repeatedly.
- Whenever a recursion function is written, there are 2 things should be followed:
  - Identification of base case or termination condition
  - Where the function should be called
- Recursion is often used in implementations of the Backtracking algorithm. For example Sudoku.
- Recursion is applied to problems (situations) where you can break it up (reduce it) into smaller parts, and each part(s) looks similar to the original problem.

## Example:

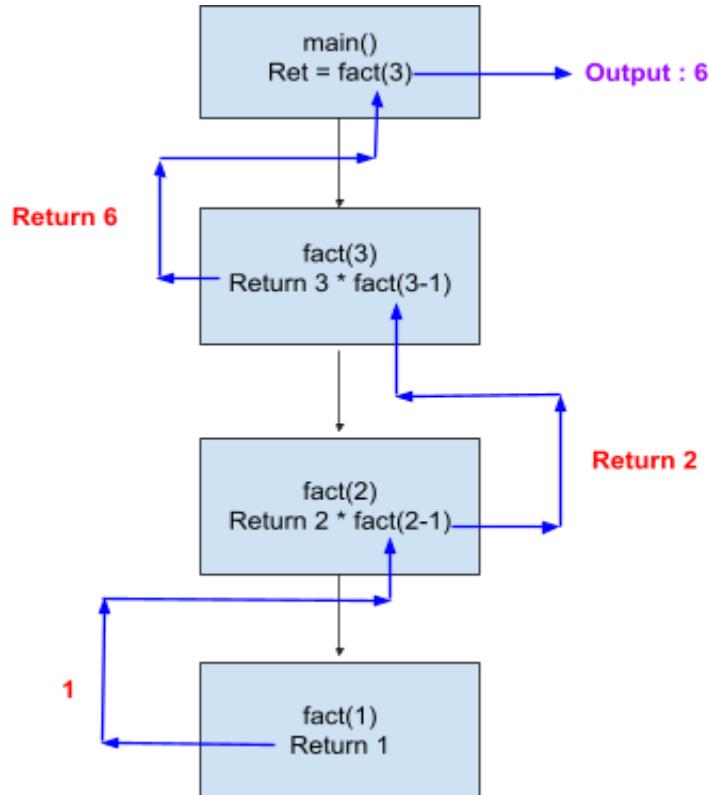
### Factorial of a number

- Factorial of a number is found by multiplying the number by every number below it till 1.
- E.g, number = 3  
Factorial =  $3 * 2 * 1 = 6$

```
int main()
{
    int ret;
    ret = factorial(3);
    printf("Factorial of 3 is %d\n", ret);
    return 0;
}

int factorial(int number)
{
    if (number <= 1) /* Base Case */
    {
        return 1;
    }
    else /* Recursive Case */
    {
        return number * factorial(number - 1);
    }
}
```

- The recursion execution will look as described in the below flow:



## Recursion vs normal loop function

- The ultimate question here is when to use recursion? Or when to go for a normal loop function?
- The answer totally depends on the resources and requirement, which is based on 2 factors: time and space
- For instance if you use recursion the space complexity will be for above factorial program:
  - Stackframe `fact(3)` will consume 4 bytes of integer and 4 bytes of return value = 8bytes
  - `fact(2)` = 8bytes
  - `fact(1)` = 8bytes

- Total of 24 bytes will be used for finding factorials of 3. It will increase if the value increases
- Same program if its written using normal loop then

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

Space complexity:  $8(\text{int fact} + \text{i}) + 4(\text{parameter}) + 4(\text{return}) = 18 \text{ bytes}$   
 This is fixed despite passing the larger value.

- Time complexity will be:
- For recursion:
  - Creation of 3 stack frame : 3 machine cycle
  - Returning 3 values : 3 machine cycle
  - Total it will take 6 machine cycle
- For normal loop:
  - 1 stackframe + Loop (1 initialisation + 4 condition evaluation + 3 increment + 3 time statement execution + 1 return value)
  - Total it will take 13 machine cycles and this will increase if the value is larger.
- So as you can see in the above analysis if the requirement has memory constraints like an embedded system then normal loop function will be preferred.
- If the constraint is on time then recursion is preferred as recursion is faster than the loop.

## Recursion with static variables

- In recursion function variables can be static variables but memory will be in the data segment.

```
void print()
{
    static int self_call = 0;
    if(self_call++ != 99)
    {
        printf("%d\n", self_call);
        print();
    }
    else
    {
        printf("End\n");
    }
}
```

- In the above example `self_call` will be in the data segment so one instance of memory will be created at the beginning, next call onwards it will make use of the same memory.
- Static can be used when the requirement says that all the recursive calls should make use of the same variable.
- Using auto local variables will result in stack overflow error, because its creating a new variable for each call which will have the same value each time.

## Recursive main() function

- Main function can be called recursively, like below example

```
int main()
```

```
{  
    static int self_Call = 0;  
  
    if(self_call != 5)  
    {  
        printf("%d\n",self_call);  
        main();  
    }  
    else  
    {  
        printf("End\n");  
    }  
}
```

- Here, main is called recursively, but without passing any argument. The function which is called recursively without any argument such functions are not true recursive functions. Therefore main can be called recursively but its not true recursive function.
- And also Making use of static variables in the recursion will make the function as not a true recursive function.

# Pointers



# Advanced C

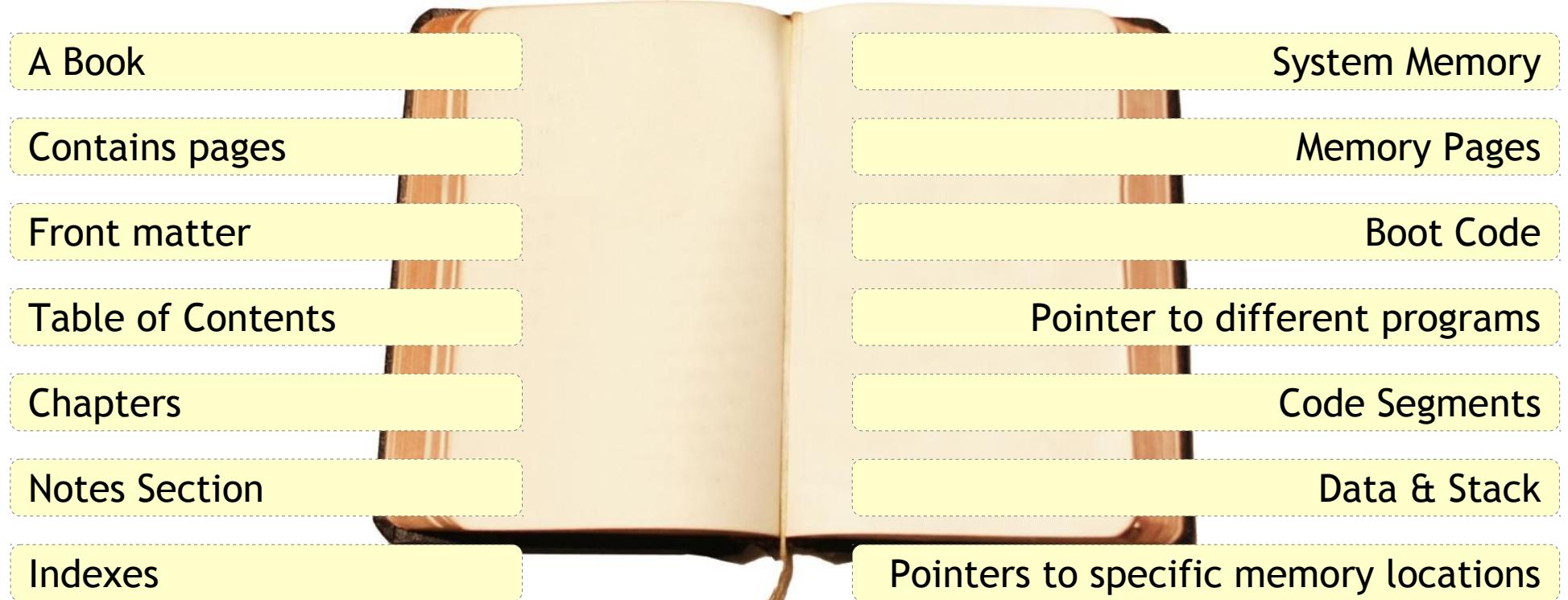
## Pointers - Jargon



- 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

# Advanced C

## Pointers - Analogy with Book



# Advanced 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 an address!!
- So the pointer concept plays a big role here



# Advanced 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 value"
- parameter passing mechanism in function, by passing the reference
- To have the dynamic allocation mechanism



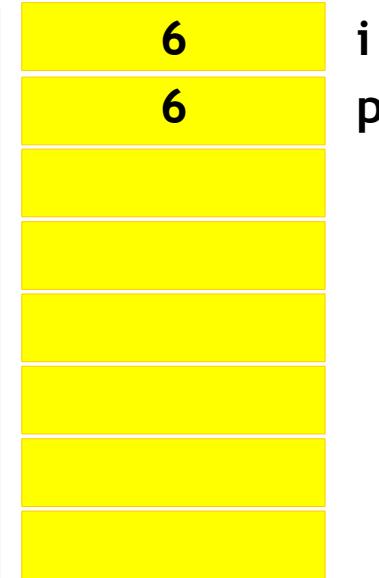
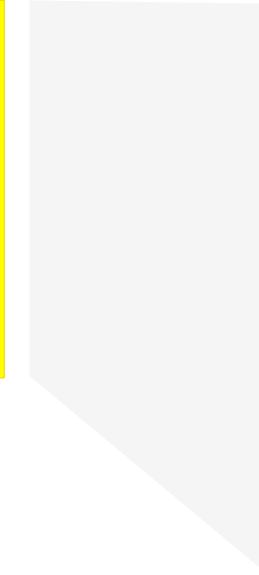
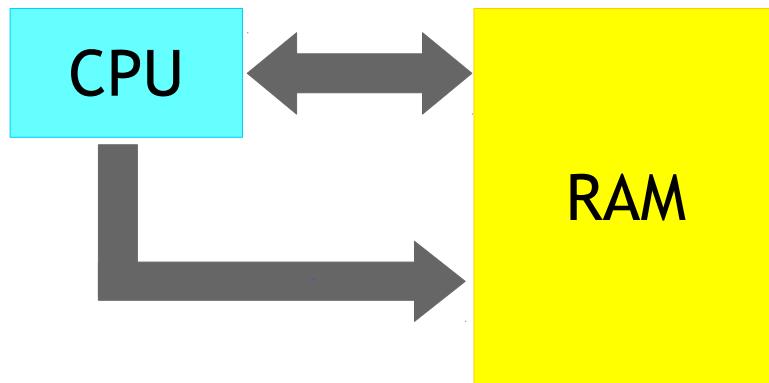
# Advanced C

## Pointers - The 7 Rules

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

# Advanced C

## Pointers - The 7 Rules - Rule 1



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

# Advanced 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)

# Advanced C

## Pointers - Rule 1 in detail

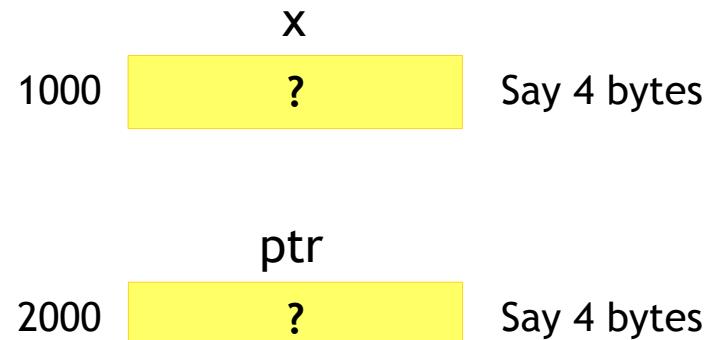
### 001\_example.c

```
#include <stdio.h>

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

    x = 5;
    ptr = &x;

    return 0;
}
```



# Advanced C

## Pointers - Rule 1 in detail



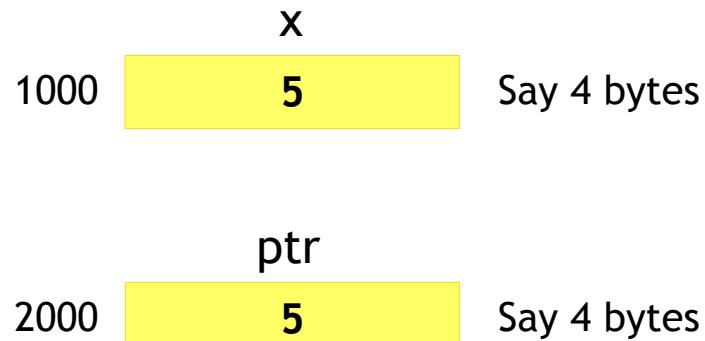
### 001\_example.c

```
#include <stdio.h>

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

    → x = 5;
    ptr = 5;

    return 0;
}
```



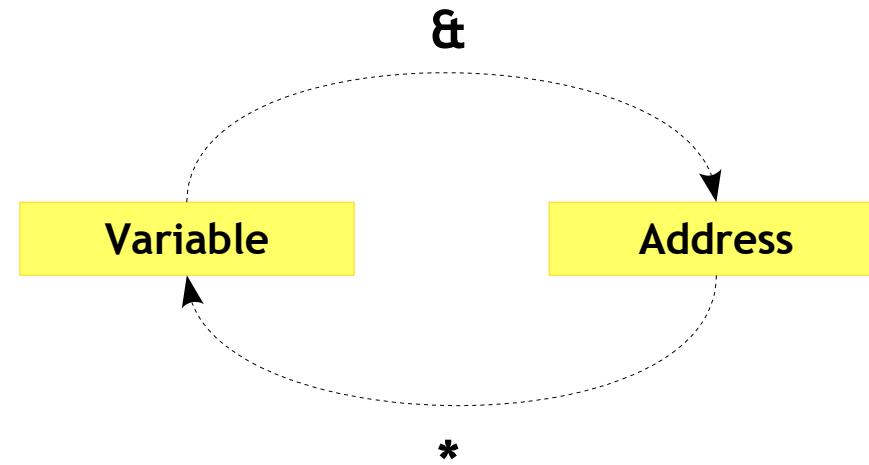
- So pointer is an integer
- But remember the “They may not be of same size”
  - 32 bit system = 4 Bytes
  - 64 bit system = 8 Bytes

# Advanced C

## Pointers - The 7 Rules - Rule 2



- Rule : “Referencing and Dereferencing”



# Advanced C

## Pointers - Rule 2 in detail

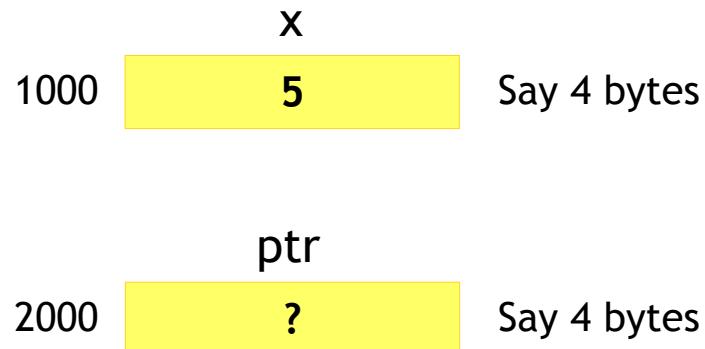
### 002\_example.c

```
#include <stdio.h>

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

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?  
\* 1000

# Advanced C

## Pointers - Rule 2 in detail

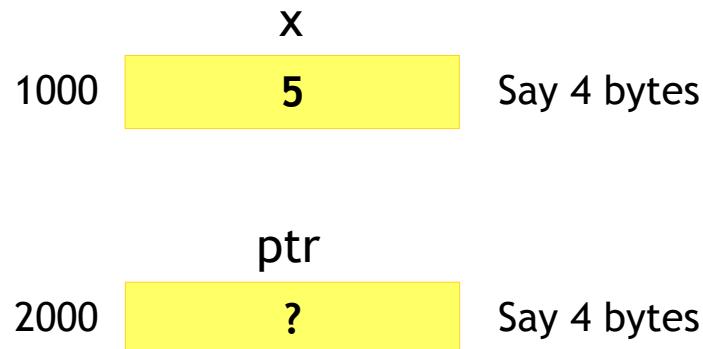
### 002\_example.c

```
#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

# Advanced C

## Pointers - Rule 2 in detail

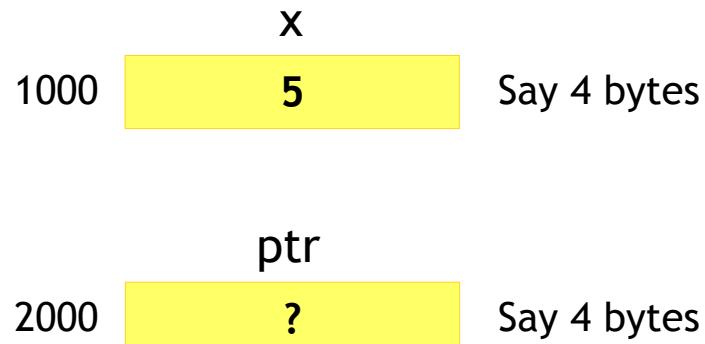
### 002\_example.c

```
#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?

# Advanced C

## Pointers - Rule 2 in detail

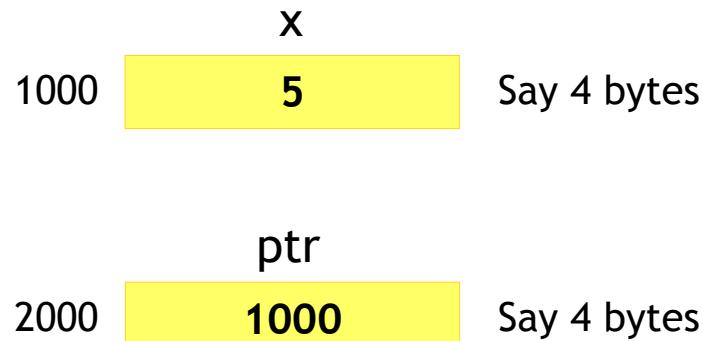
### 002\_example.c

```
#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

# Advanced C

## Pointers - Rule 2 in detail

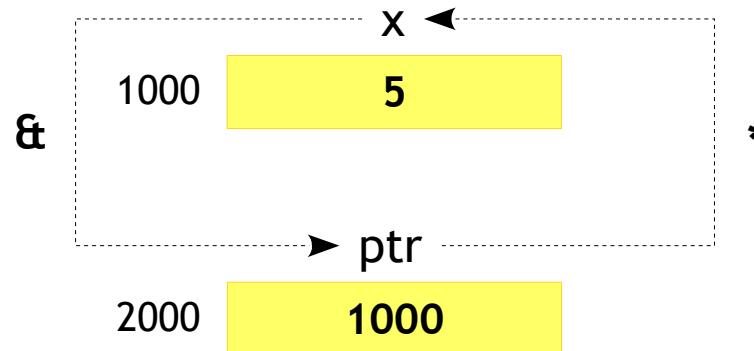
002\_example.c

```
#include <stdio.h>

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

    x = 5;
    ptr = &x;

    return 0;
}
```



“Prefix 'address of operator' (**&**) with variable (x) to get its address and store in the pointer”

“Prefix 'indirection operator' (**\***) with pointer to get the value of variable (x) it is pointing to”

# Advanced C

## Pointers - Rule 2 in detail

### 003\_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;
}
```

# Advanced C

## Pointers - Rule 2 in detail

### 004\_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;
}
```

# Advanced C

## Pointers - Rule 2 in detail



### 005\_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;
}
```

- So, from the above code we can conclude  
“`*ptr <=> number`”

# Advanced C

## Pointers - The 7 Rules - Rule 3

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

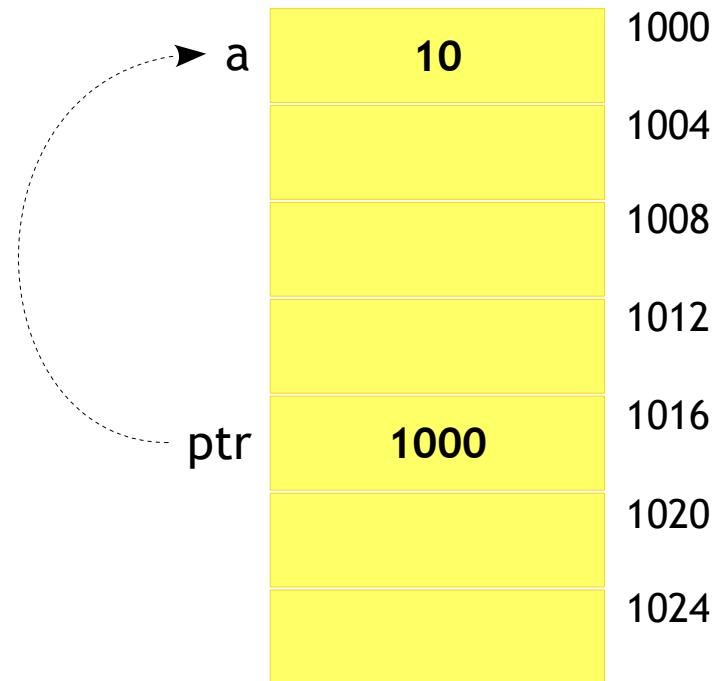
### Example

```
#include <stdio.h>

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

    ptr = &a;

    return 0;
}
```



# Advanced C

## Pointers - The 7 Rules - Rule 4

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

# Advanced C

## Pointers - Rule 4 in detail



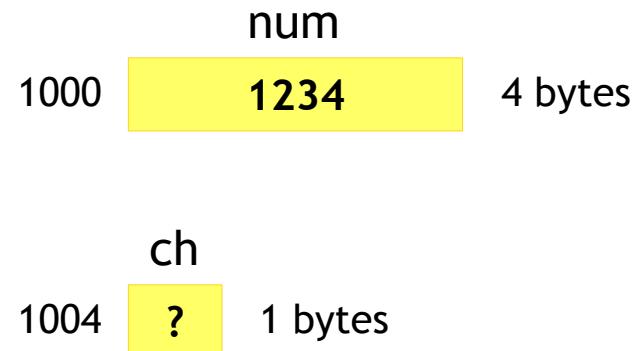
- Does address has a type?

### Example

```
#include <stdio.h>

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

    return 0;
}
```



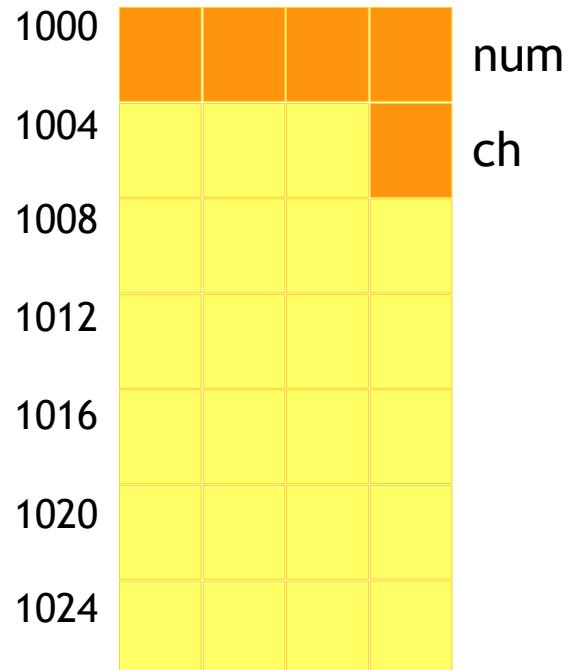
- So from the above diagram can we say `&num` → 4 bytes and `&ch` → 1 byte?

# Advanced C

## Pointers - Rule 4 in detail



- The answer is no!!
- Address size does not depend on type of the variable
- It depends on the system we use and remains same across all pointers
- Then a simple question arises “why type is used with pointers?”



# Advanced C

## Pointers - Rule 4 in detail



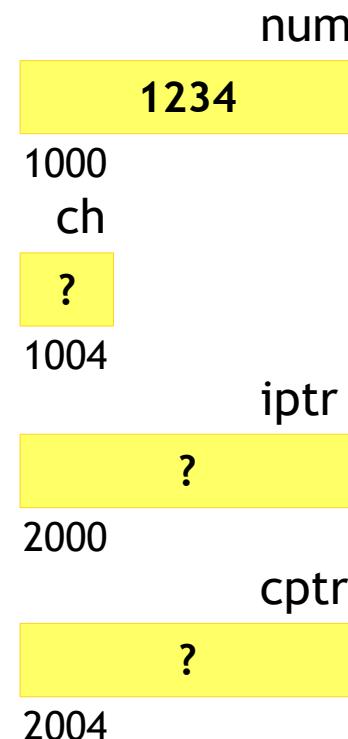
### Example

```
#include <stdio.h>

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

    int *iptr;
    char *cptr;

    return 0;
}
```



- Lets consider above example to understand it
- Say we have an integer and a character pointer

# Advanced C

## Pointers - Rule 4 in detail



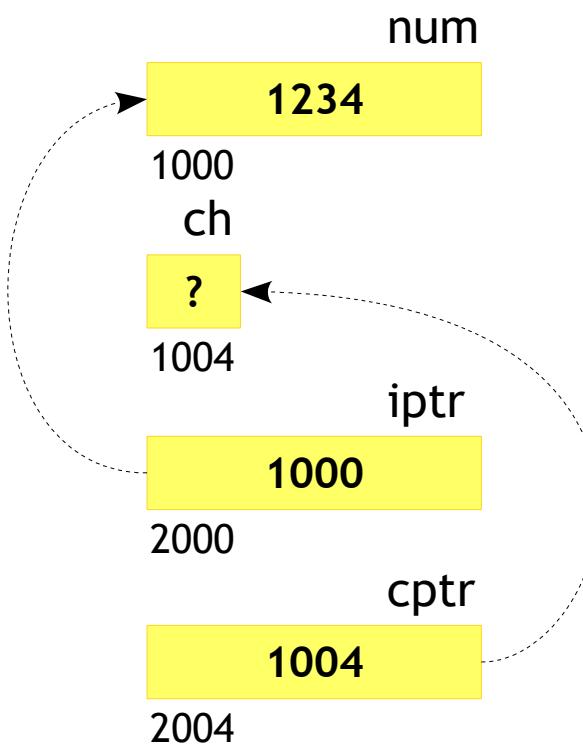
### Example

```
#include <stdio.h>

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

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

    return 0;
}
```



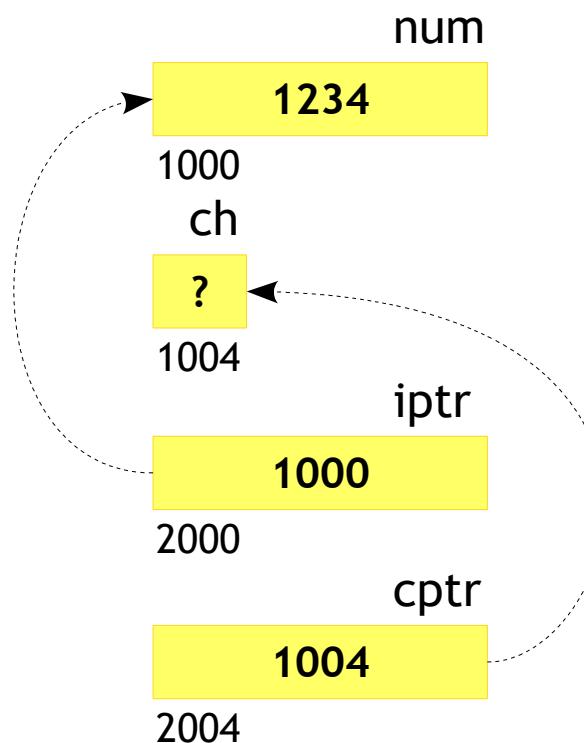
- Lets consider the above examples to understand it
- Say we have a integer and a character pointer

# Advanced C

## Pointers - Rule 4 in detail



- With just the address, can we 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!!

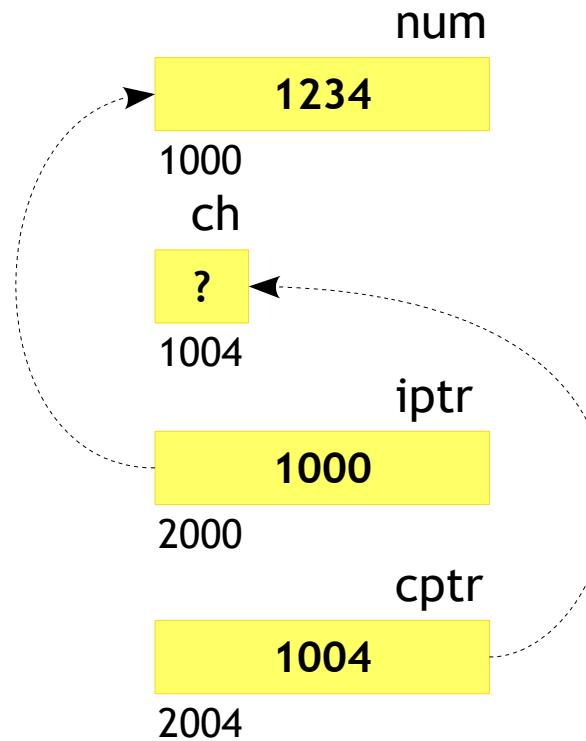


# Advanced C

## Pointers - Rule 4 in detail



- From the diagram right side we can say
  - \*cptr fetches a single byte
  - \*iptr fetches 4 consecutive bytes
- So, in conclusion we can say



(type \*) → fetch sizeof(type) bytes

# Advanced 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



# Advanced C

## Pointers - Rule 4 in detail - Endianness



- LSB (Least Significant Byte)
  - The byte of a multi byte number with the least importance
  - The change in it would have least effect on number's value change
- MSB (Most Significant Byte)
  - The byte of a multi byte number with the most importance
  - The change in it would have larger effect on number's value change



# Advanced C

## Pointers - Rule 4 in detail - Endianness

### Example

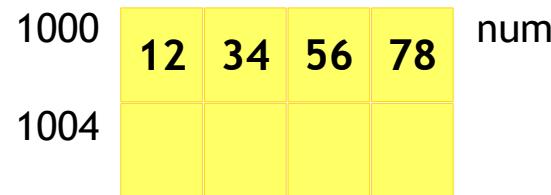
```
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    return 0;
}
```

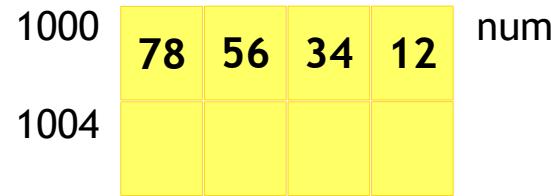
- Let us consider the following example and how it would be stored in both machine types

Big Endian



1000	12	34	56	78
1001				
1002				
1003				

Little Endian



1000	78	56	34	12
1001				
1002				
1003				

# Advanced C

## Pointers - Rule 4 in detail - Endianness



- OK Fine. What now? How is it going to affect the 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;

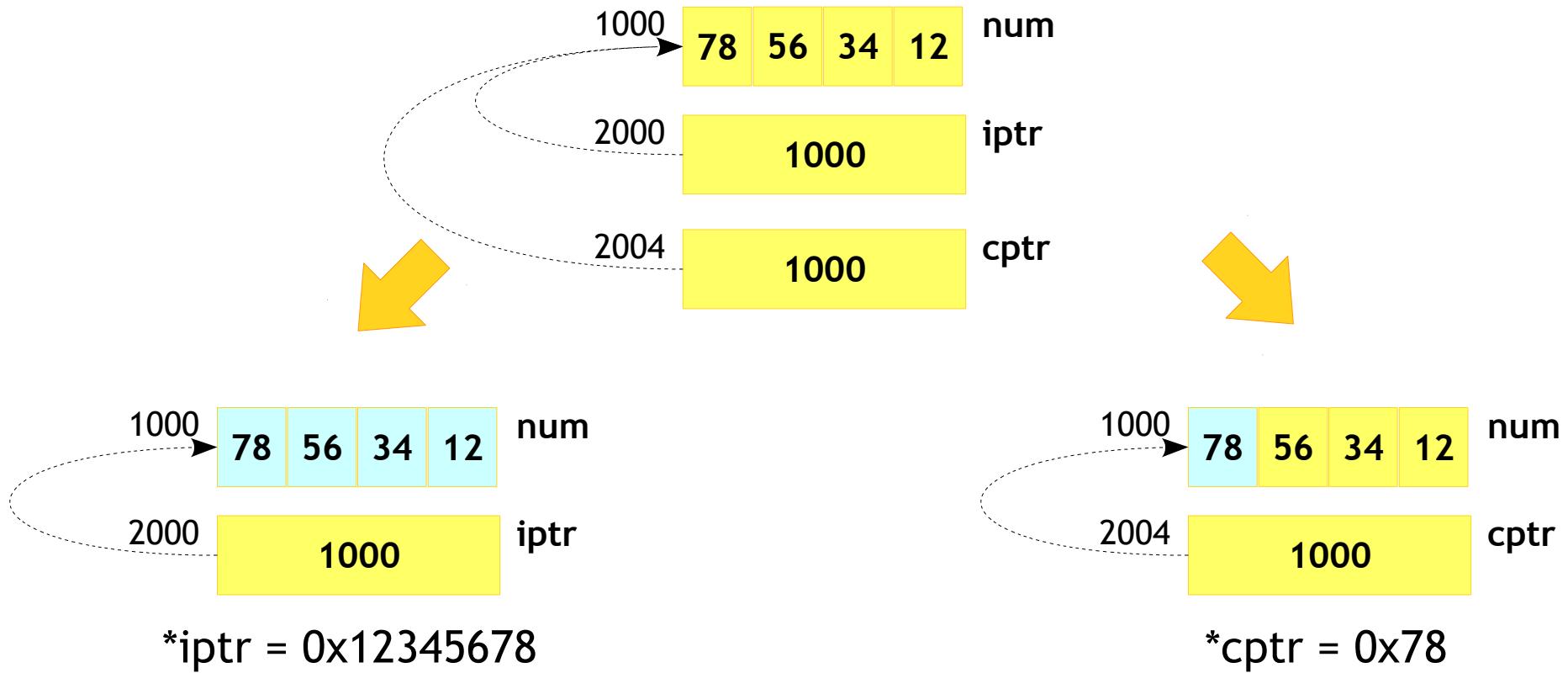
    iptr = &num;
    cptr = &num;

    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 Little Endian system

# Advanced 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

# Advanced C

## Pointers - The 7 Rules - Rule 4

### Example

```
#include <stdio.h>

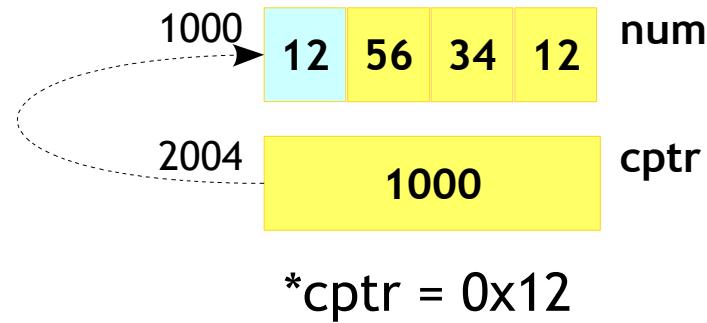
int main()
{
    int num = 0x12345678;
    char ch;

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

    *cptr = 0x12;

    return 0;
}
```

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



- So `*iptr` would contain 0x12345612 now!!

# Advanced C

## Pointers - The 7 Rules - Rule 4



- In conclusion,
  - The **type** of a pointer represents its ability to perform read or write operations on number of bytes (data) starting from address its pointing to
  - **Size** of all different type pointers remains same

### 006\_example.c

```
#include <stdio.h>

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

    return 0;
}
```

# Advanced C

## Pointers - The 7 Rules - Rule 4 - DIY



- WAP to check whether a machine is Little or Big Endian



# Advanced C

## Pointers - The 7 Rules - Rule 5

- Pointer Arithmetic

**Rule:** “ $\text{Value}(p + i) = \text{Value}(p) + i * \text{sizeof}(*p)$ ”

# Advanced C

## Pointers - The Rule 5 in detail



- Before proceeding further let us understand an array interpretation
  - Original Big Variable (bunch of variables, **whole array**)
  - Constant Pointer to the 1st Small Variable in the bunch (**base address**)
- When first interpretation fails than second interpretation applies

# Advanced C

## Pointers - The Rule 5 in detail



- Cases when first interpretation applies
  - When name of array is operand to sizeof operator
  - When “address of operator (&)" is used with name of array while performing pointer arithmetic
- Following are the cases when first interpretation fails
  - When we pass array name as function argument
  - When we assign an array variable to pointer variable



# Advanced C

## Pointers - The Rule 5 in detail

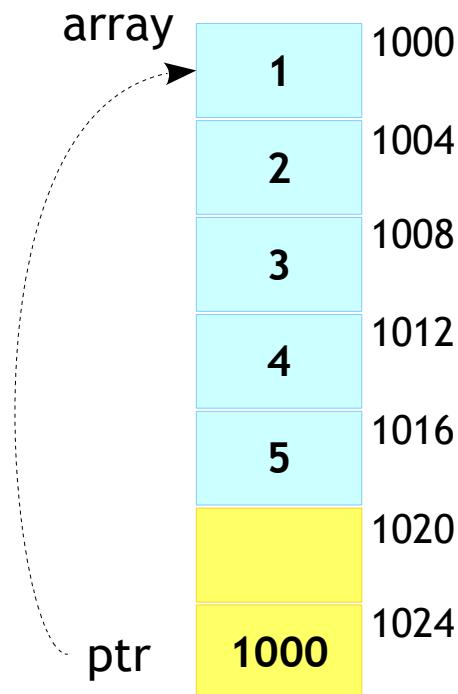
### 007\_example.c

```
#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  
 $\&\text{array}[0] = 1 \rightarrow 1000$   
 $\&\text{array}[1] = 2 \rightarrow 1004$



# Advanced C

## Pointers - The Rule 5 in detail

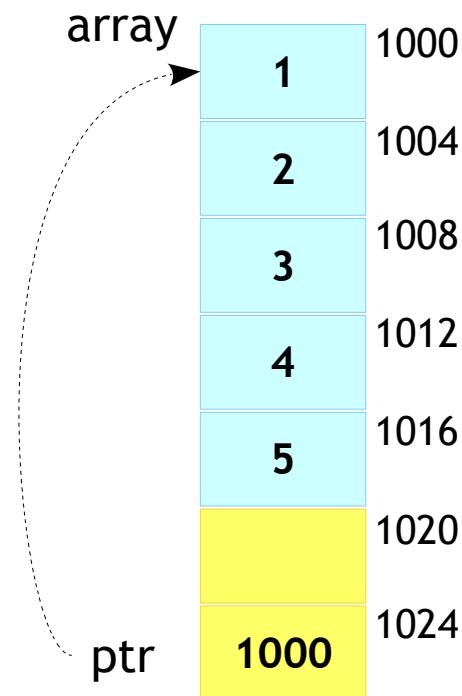
### 007\_example.c

```
#include <stdio.h>

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

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

    return 0;
}
```



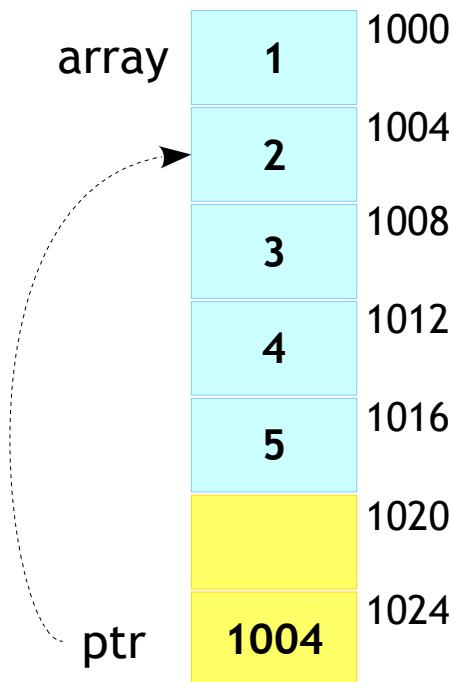
- This code should print 1 as output since its points to the base address
- Now, what should happen if we do  
`ptr = ptr + 1;`

# Advanced C

## Pointers - The Rule 5 in detail

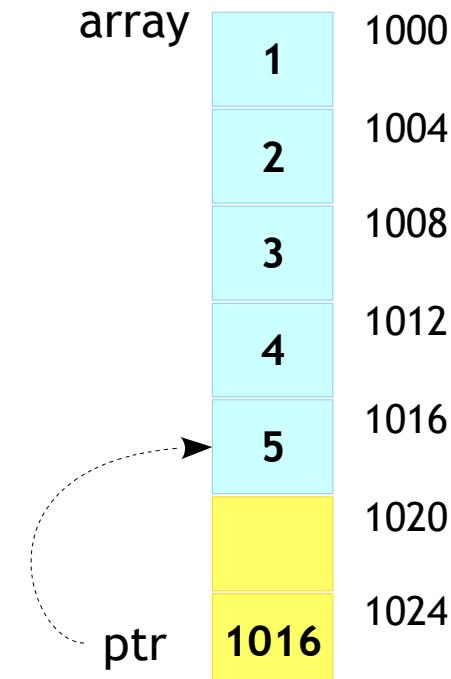
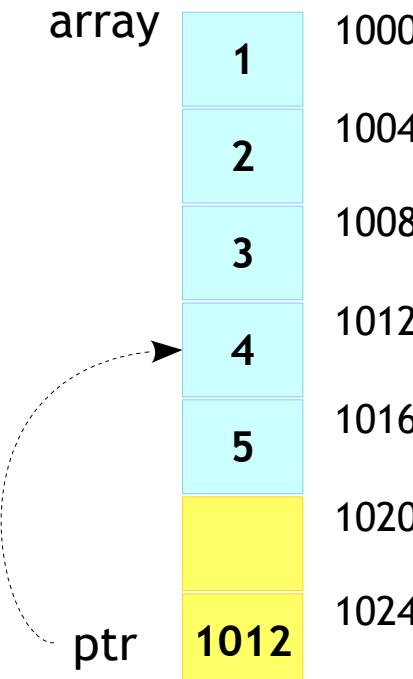
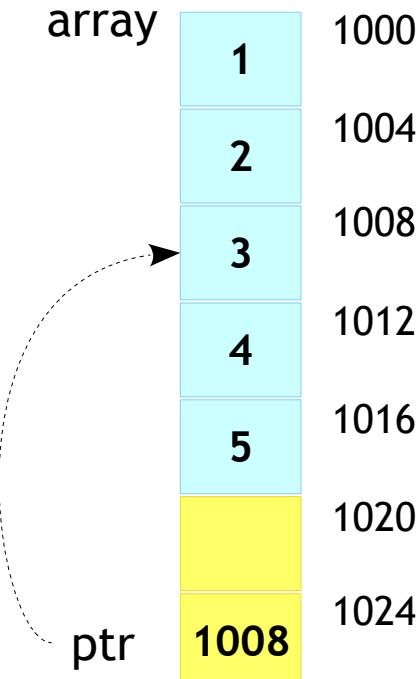


- $\text{ptr} = \text{ptr} + 1;$
- The above line can be described as follows
- $\text{ptr} = \text{ptr} + 1 * \text{sizeof(data type)}$
- In this example we have a integer array, so
- $$\begin{aligned}\text{ptr} &= \text{ptr} + 1 * \text{sizeof(int)} \\ &= \text{ptr} + 1 * 4 \\ &= \text{ptr} + 4\end{aligned}$$
- Here  $\text{ptr} = 1000$  so
$$\begin{aligned}&= 1000 + 4 \\ &= 1004\end{aligned}$$



# Advanced C

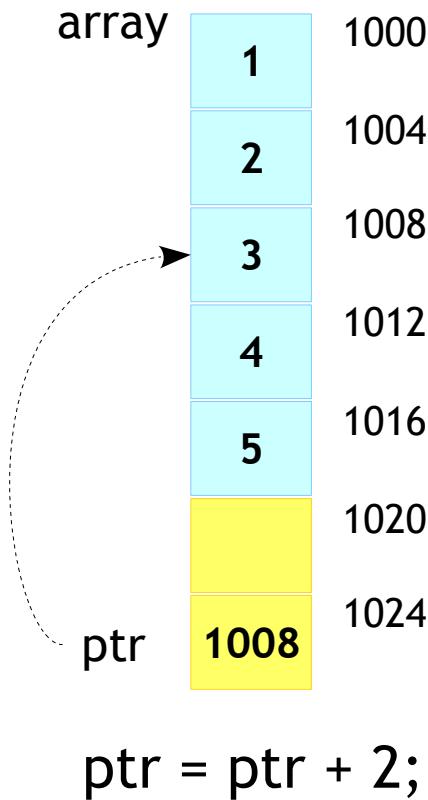
## Pointers - The Rule 5 in detail



- Why does the compiler does this? Just for convenience

# Advanced C

## Pointers - The Rule 5 in detail



- For array, it can be explained as
  - $\text{ptr} + 2$
  - $\text{ptr} + 2 * \text{sizeof(int)}$
  - $1000 + 2 * 4$
  - $1008 \rightarrow \&\text{array}[2]$
- So,
  - $\text{ptr} + 2 \rightarrow 1008 \rightarrow \&\text{array}[2]$
  - $*(\text{ptr} + 2) \rightarrow *(1008) \rightarrow \text{array}[2]$

# Advanced C

## Pointers - The Rule 5 in detail



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

$\ast(\text{ptr} + i) \rightarrow \text{array}[i]$

- This can be written as following too!!

$\text{array}[i] \rightarrow \ast(\text{array} + i)$

- Which results to

$\text{ptr} = \text{array}$

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

`int *ptr = array;`



# Advanced C

## Pointers - The Rule 5 in detail



- Wait can I write

$$*(\text{ptr} + i) \rightarrow *(i + \text{ptr})$$

- Yes. So than can I write

$$\text{array}[i] \rightarrow i[\text{array}]$$

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



# Advanced C

## Pointers - The 7 Rules - Rule 6

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

# Advanced C

## Pointers - Rule 6 in detail - NULL Pointer



### Example

```
#include <stdio.h>

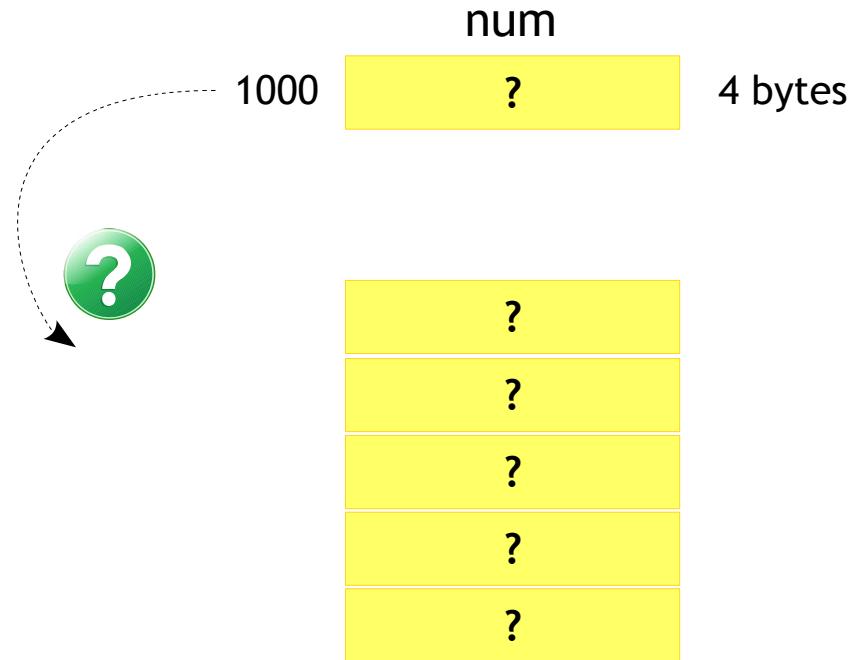
int main()
{
    int *num;

    return 0;
}
```

Where am I  
pointing to?

What does it  
Contain?

Can I read or  
write wherever  
I am pointing?



# Advanced 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!!**



# Advanced 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



# Advanced C

## Pointers - Rule 6 in detail - NULL Pointer



- In pointer context, an integer constant expression with value zero or such an expression typecast to void \* is called **null pointer constant** or NULL
  - [defined as 0 or (void \*)0 ]
- If a pointer is initialized with null pointer constant, it is called **null pointer**
- A Null Pointer is logically understood to be **Pointing to Nothing**
- De-referencing a NULL pointer is illegal and will lead to crash (segment violation on Linux or reboot on custom board), which is better than pointing to some unknown location and failing randomly!



# Advanced 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



# Advanced C

## Pointers - Rule 6 in detail - NULL Pointer



### Example

```
#include <stdio.h>

int main()
{
    int *num;

    num = NULL;

    return 0;
}
```

### Example

```
#include <stdio.h>

int main()
{
    int *num = NULL;

    return 0;
}
```

# Advanced C

## Pointers - Void Pointer



- A pointer with incomplete type
- Due to incomplete type
  - Pointer arithmetic can't be performed
  - Void pointer can't be dereferenced. You **MUST** use type cast operator “(type)” to dereference



# Advanced C

## Pointers - Size of void - Compiler Dependency



### :GCC Extension:

#### 6.22 Arithmetic on void- and Function-Pointers

In GNU C, addition and subtraction operations are supported on “**pointers to void**” and on “**pointers to functions**”. This is done by treating the size of a void or of a function as 1.

A consequence of this is that `sizeof` is also allowed on void and on function types, and returns 1.

The option **-Wpointer-arith** requests a warning if these extensions are used



# Advanced C

## Pointers - Void Pointer - Size of void



- Due to gcc extension, size of void is 1
- Hence, gcc allows pointer arithmetic on void pointer
- Don't forget! Its compiler dependent!

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



# Advanced C

## Pointers - Void Pointer



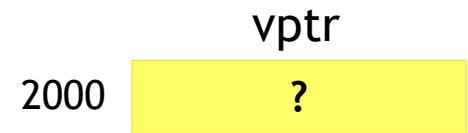
- A generic pointer which can point to data in memory
- The data type has to be mentioned while accessing the memory which has to be done by type casting

### Example

```
#include <stdio.h>

int main()
{
    void *vptr;

    return 0;
}
```



# Advanced C

## Pointers - Void Pointer

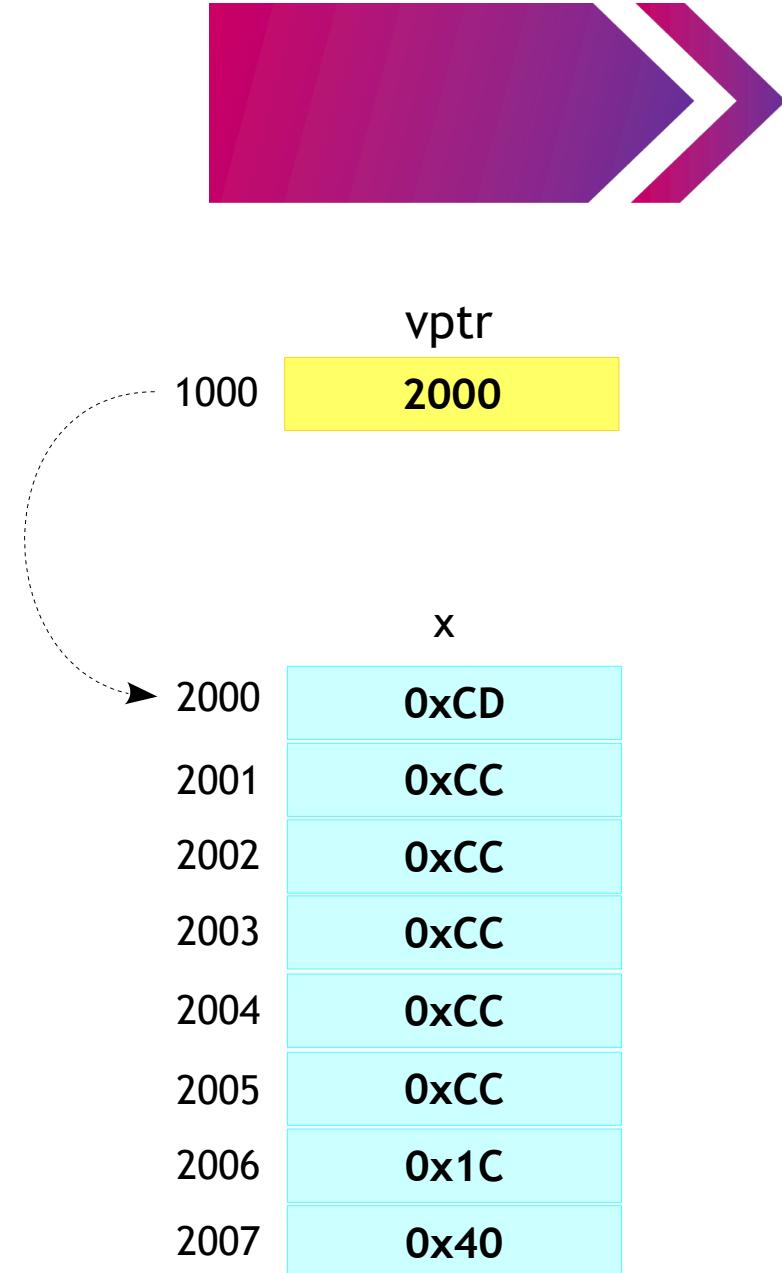
008\_example.c

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    return 0;
}
```

- vptr is a void pointer pointing to address of x which holds floating point data with double type
- These eight bytes are the legal region to the vptr
- We can access any byte(s) within this region by type casting



# Advanced C

## Pointers - Void Pointer



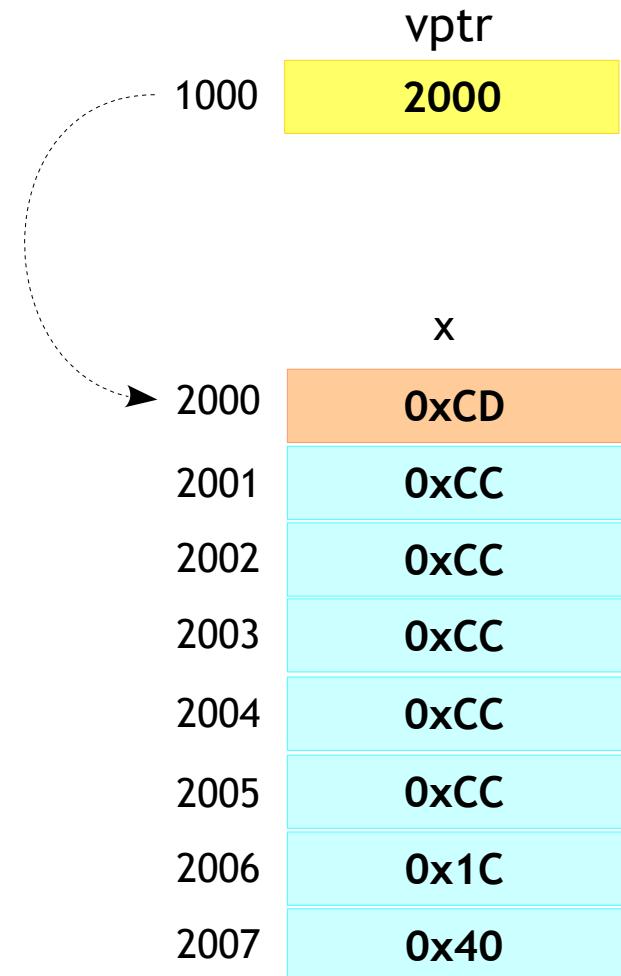
### 008\_example.c

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

→ printf("%hx\n", *(char *)vptr);
    printf("%hx\n", *(char *)(vptr + 7));
    printf("%hu\n", *(short *)(vptr + 3));
    printf("%x\n", *(int *)(vptr + 0));

    return 0;
}
```



# Advanced C

## Pointers - Void Pointer



008\_example.c

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

→ printf("%hx\n", *(char *)vptr);
→ printf("%hx\n", *(char *)(vptr + 7));
printf("%hu\n", *(short *)(vptr + 3));
printf("%x\n", *(int *)(vptr + 0));

    return 0;
}
```

vptr	x
1000	
2000	0xCD
2001	0xCC
2002	0xCC
2003	0xCC
2004	0xCC
2005	0xCC
2006	0x1C
2007	0x40

# Advanced C

## Pointers - Void Pointer



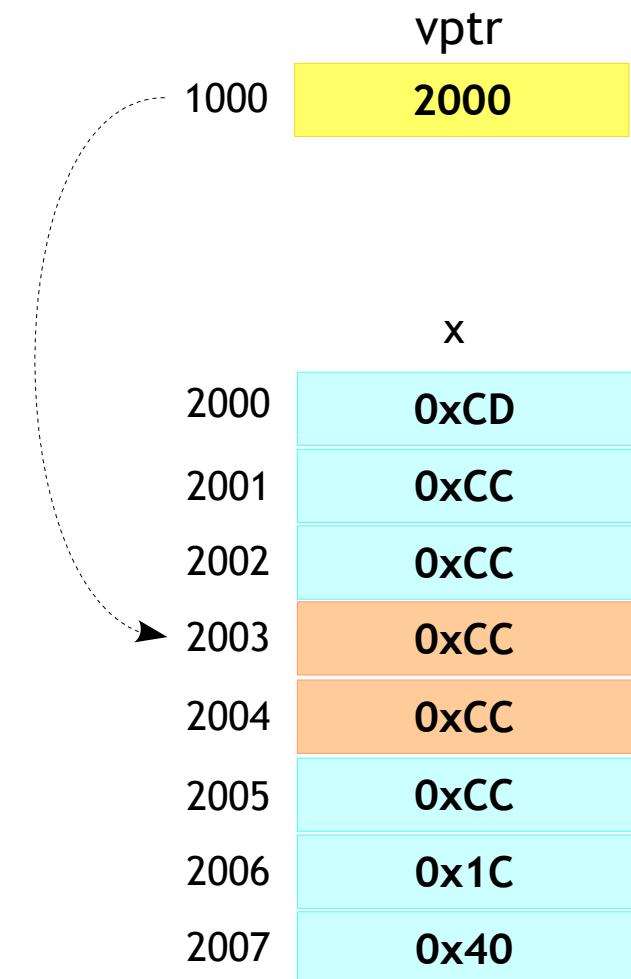
008\_example.c

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf("%hx\n", *(char *)vptr);
    printf("%hx\n", *(char *)(vptr + 7));
    printf("%hu\n", *(short *)(vptr + 3));
    printf("%x\n", *(int *)(vptr + 0));

    return 0;
}
```



# Advanced C

## Pointers - Void Pointer

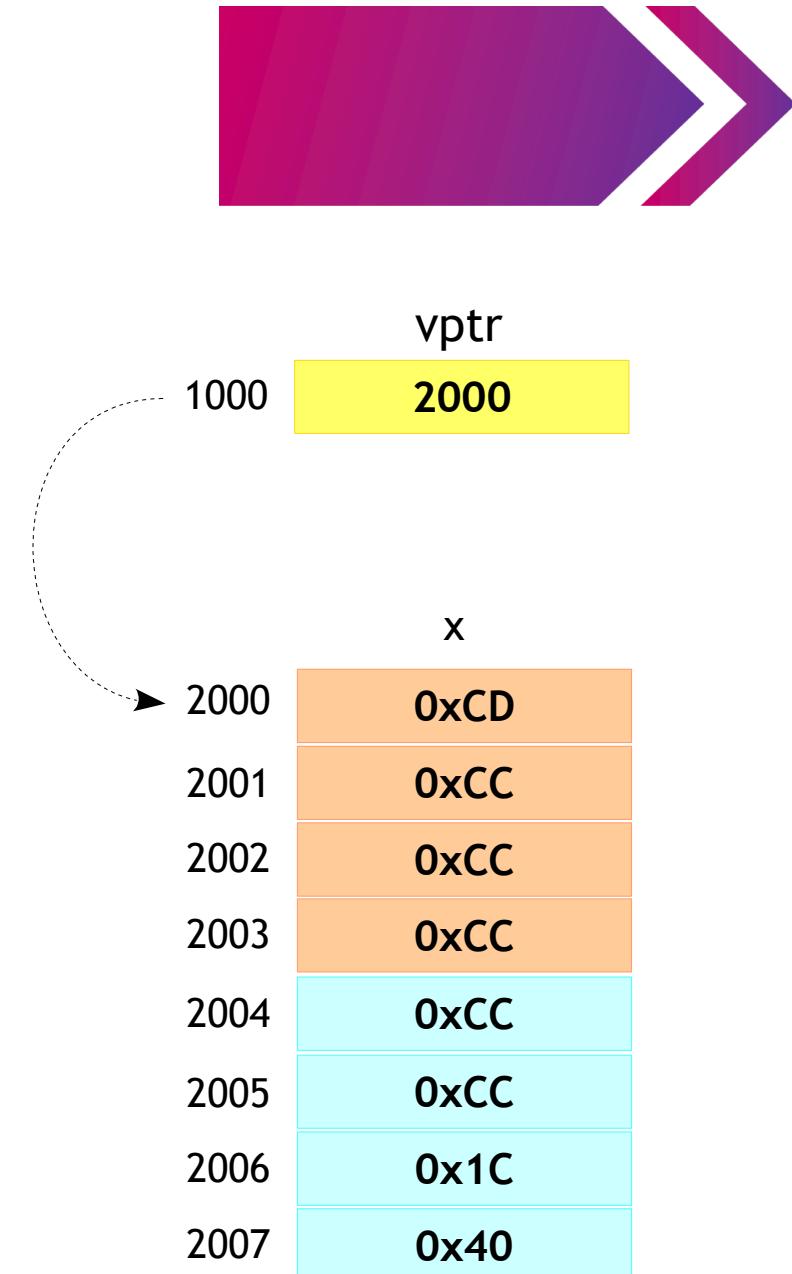
008\_example.c

```
#include <stdio.h>

int main()
{
    double x = 7.2;
    void *vptr = &x;

    printf("%hx\n", *(char *)vptr);
    printf("%hx\n", *(char *)(vptr + 7));
    printf("%hu\n", *(short *)(vptr + 3));
    → printf("%x\n", *(int *)(vptr + 0));

    return 0;
}
```



# Advanced C

## Pointers - Void Pointer

- W.A.P to swap any given data type

# Advanced C

## Pointers - The 7 Rules - Rule 7

- Rule: “Static Allocation vs Dynamic Allocation”

### Example

```
#include <stdio.h>

int main()
{
    char array[5];

    return 0;
}
```

### Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```

# Advanced C

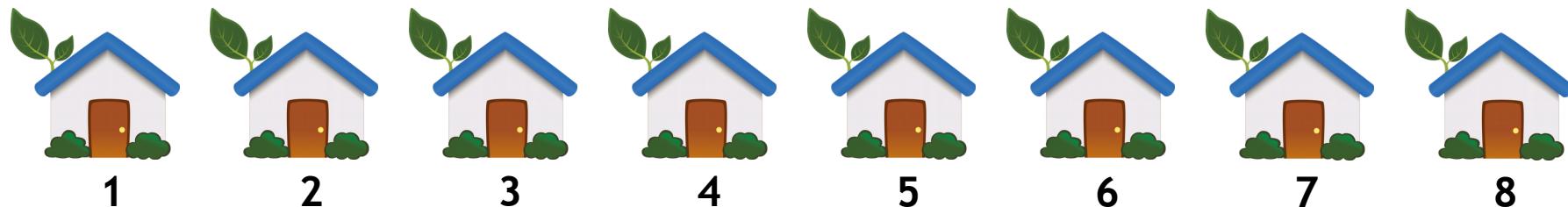
## Pointers - Rule 7 in detail



- Named vs Unnamed Allocation = Named vs Unnamed Houses



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



Ok, House 1, I should go that side ←



# Advanced 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

# Advanced C

## Pointers - Rule 7 in detail

- Static vs Dynamic

### Example

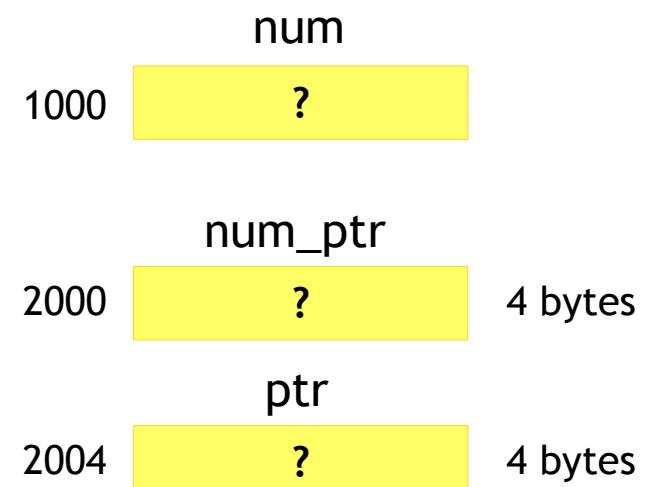
```
#include <stdio.h>

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

    num_ptr = &num;

    ptr = malloc(4);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 in detail

- Static vs Dynamic

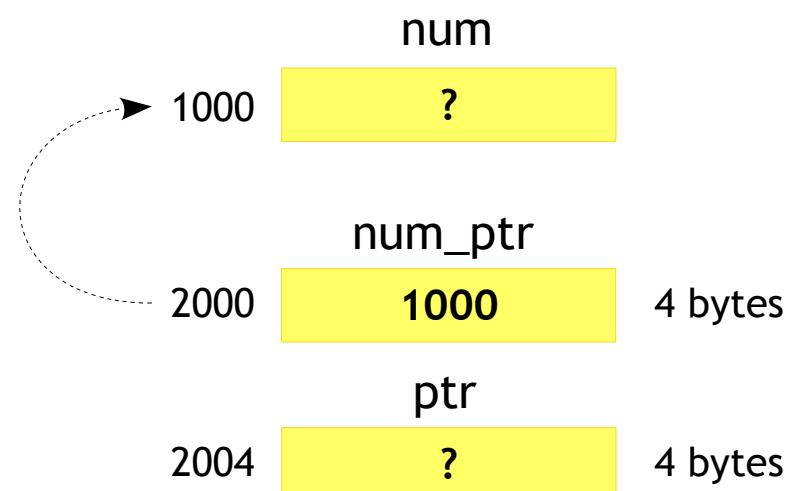
### Example

```
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;
→ num_ptr = &num;

    ptr = malloc(4);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 in detail

- Static vs Dynamic

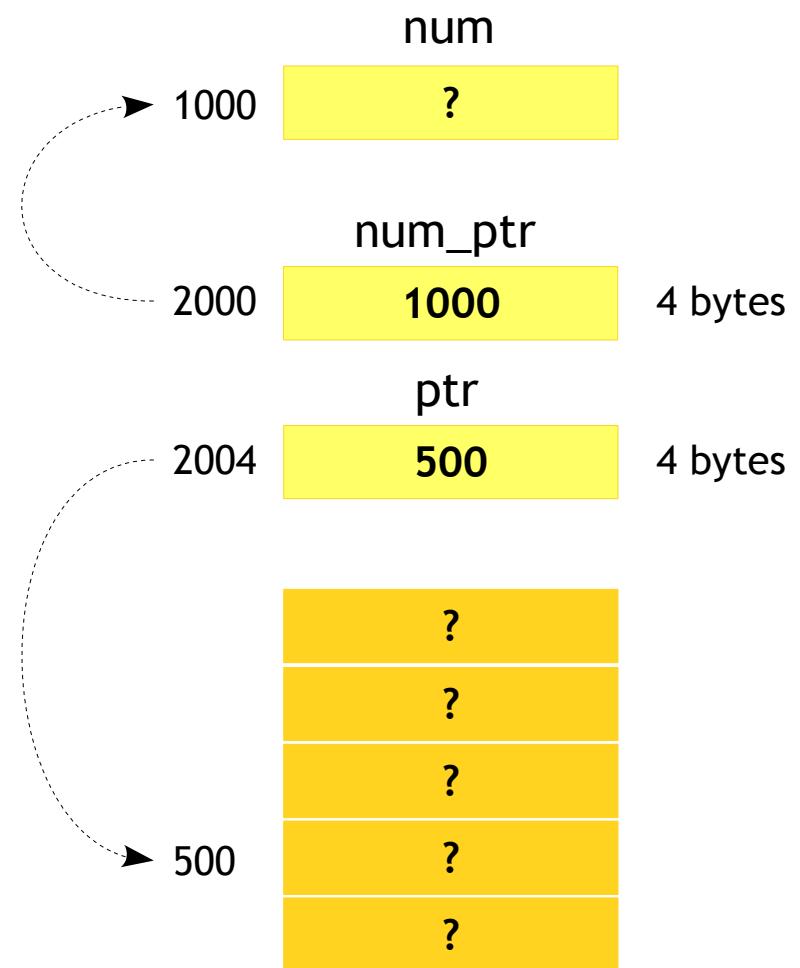
### Example

```
#include <stdio.h>

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

→ptr = malloc(4);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 in detail - Dynamic Allocation

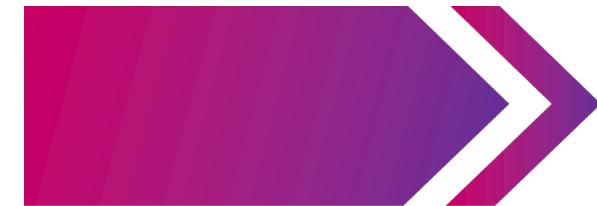


- The need
  - You can decide size of the memory at run time
  - You can resize it whenever required
  - You can decide when to create and destroy it



# Advanced 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

# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - malloc

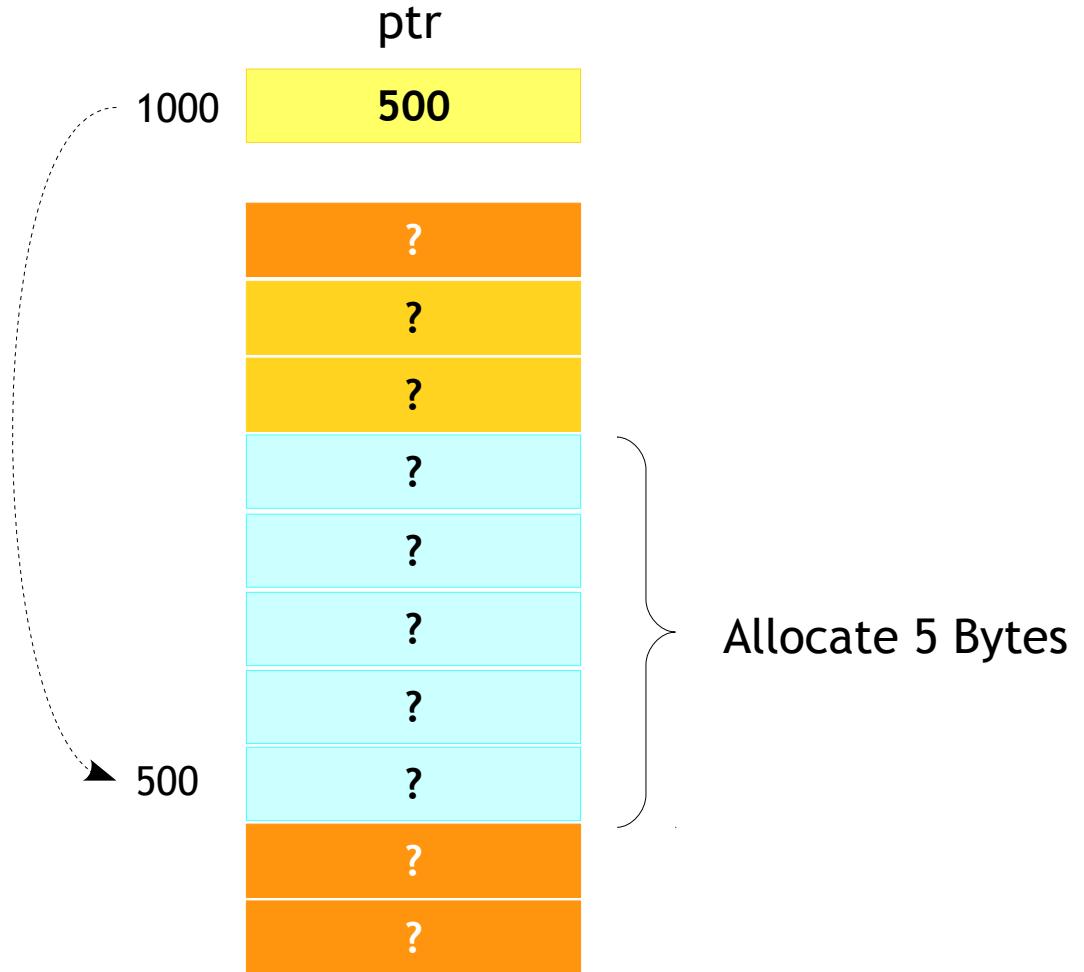
### Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - malloc

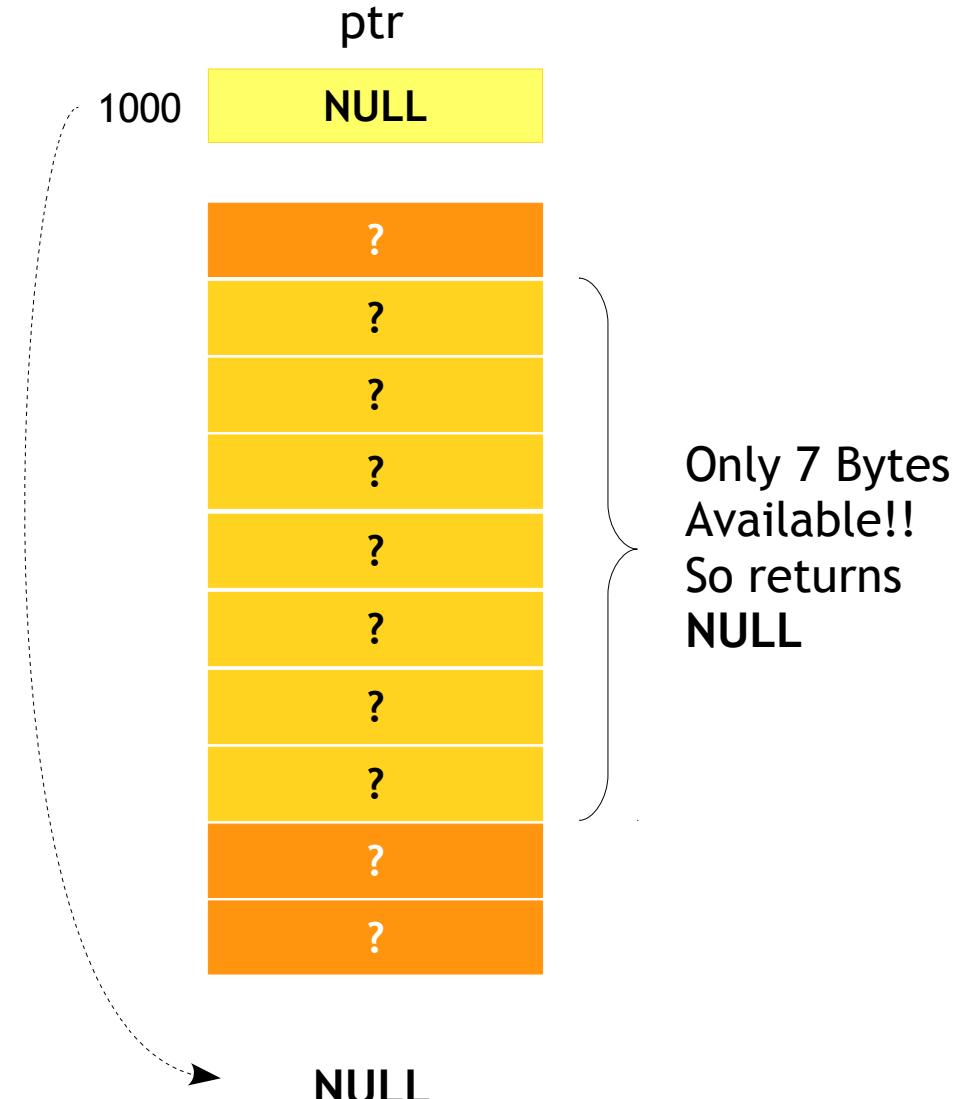
### Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(10);

    return 0;
}
```



# Advanced 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

# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - calloc

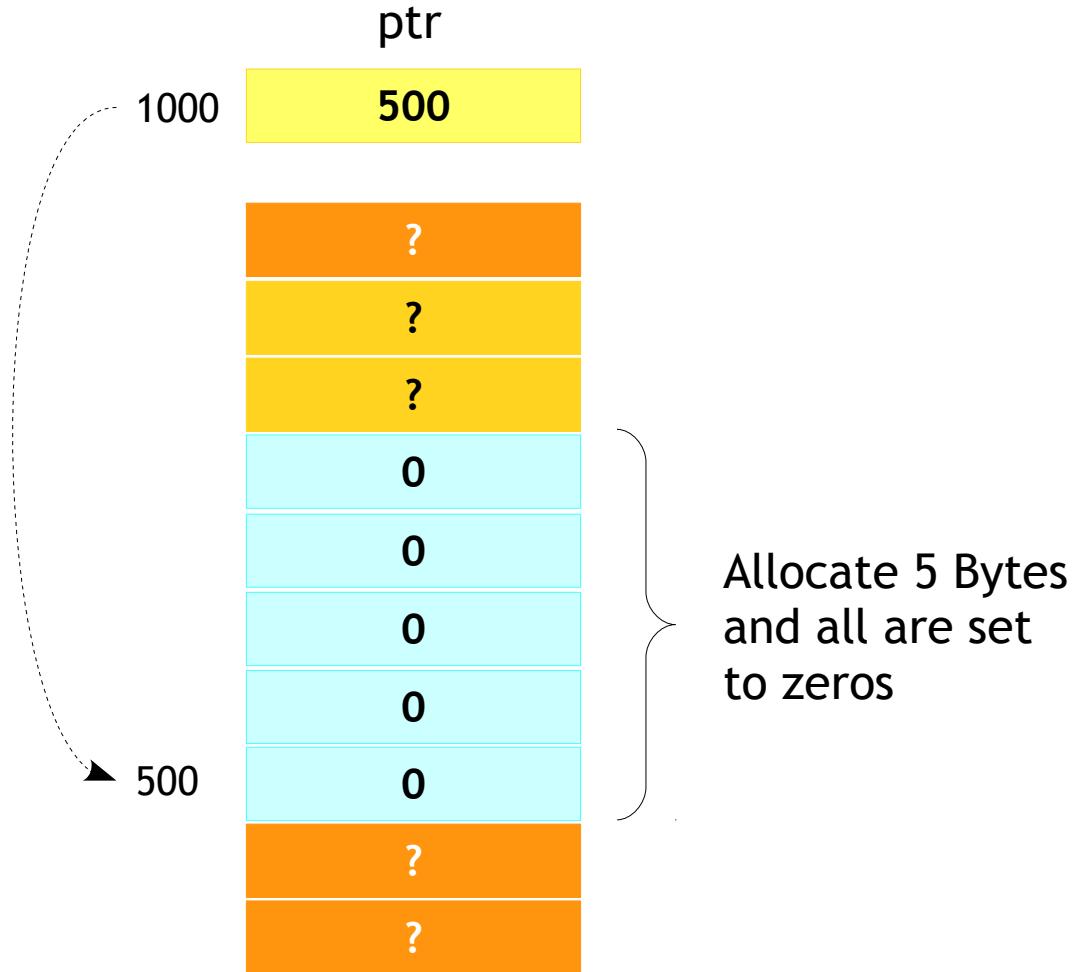
### Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = calloc(5, 1);

    return 0;
}
```



# Advanced 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

# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - realloc

### Example

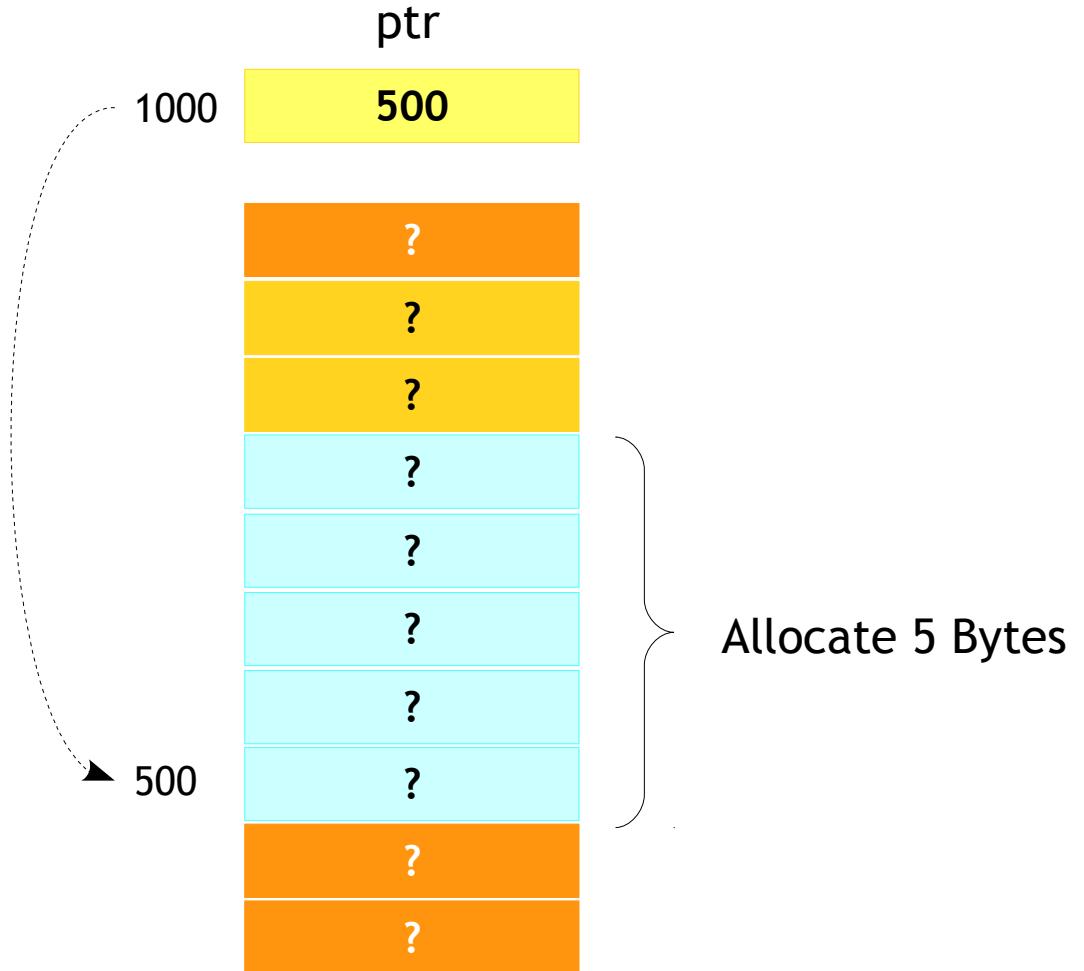
```
#include <stdio.h>

int main()
{
    char *ptr;

→ [ptr = malloc(5);]

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

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - realloc

### Example

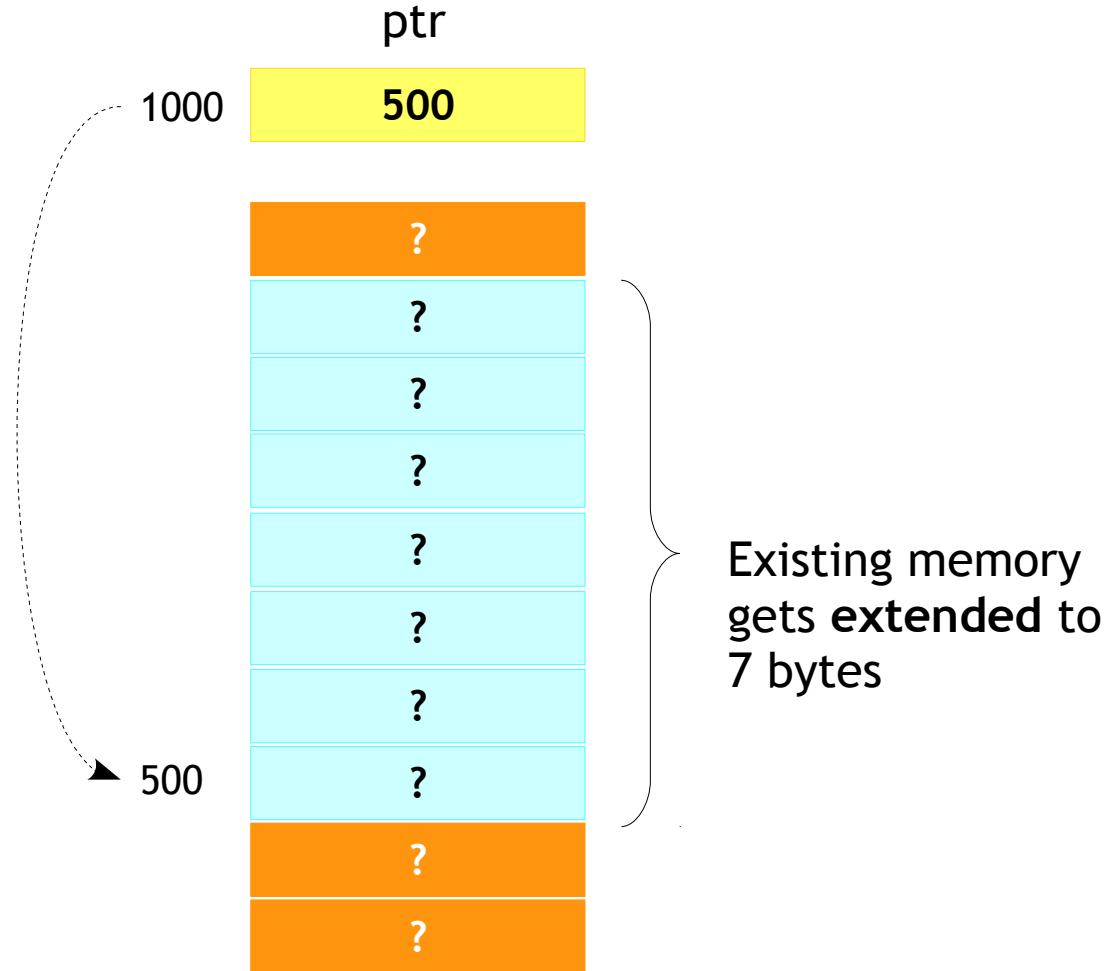
```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

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

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - realloc

### Example

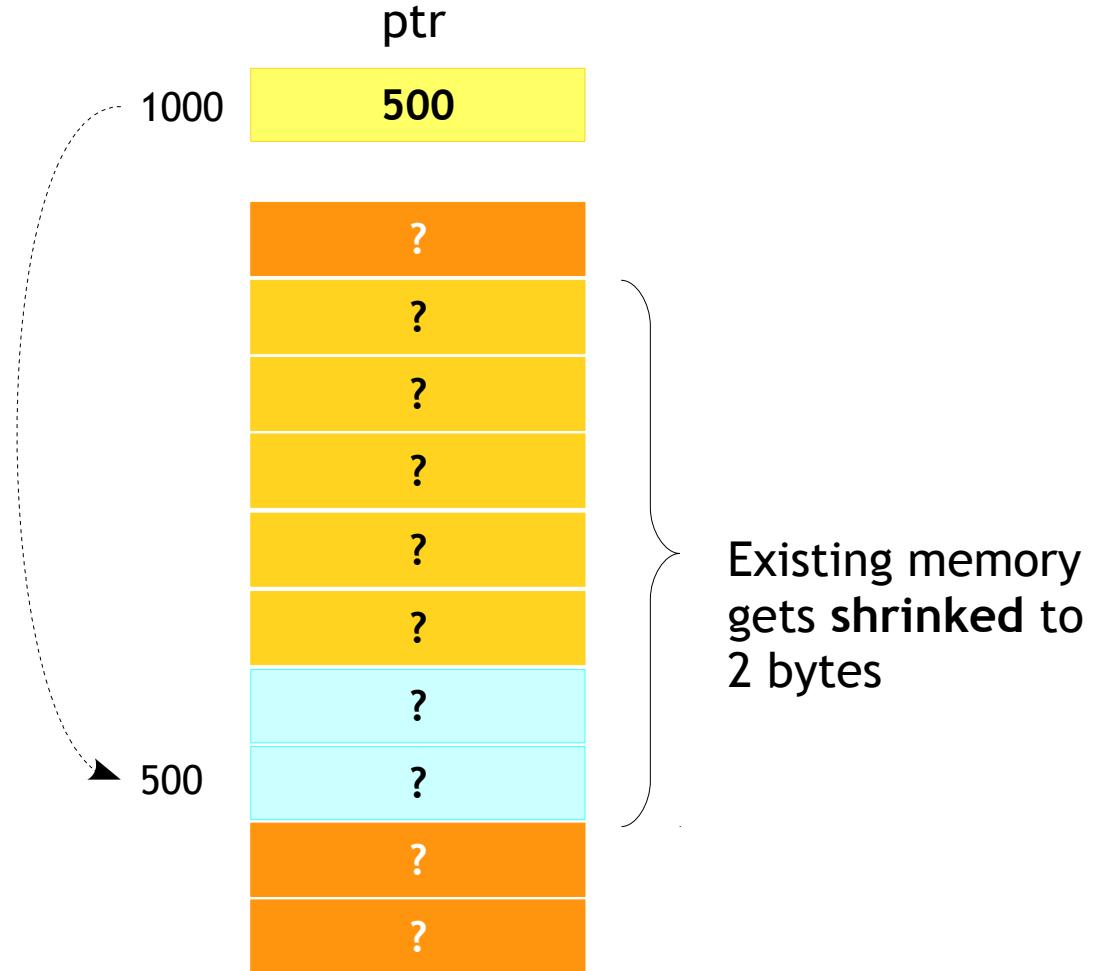
```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

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

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Allocation - realloc

- Points to be noted
  - Reallocating existing memory will be like deallocated 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!
  - If new memory block is allocated then old memory block is automatically freed by realloc function

# Advanced 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

# Advanced C

## Pointers - Rule 7 - Dynamic Deallocation - free

### Example

```
#include <stdio.h>

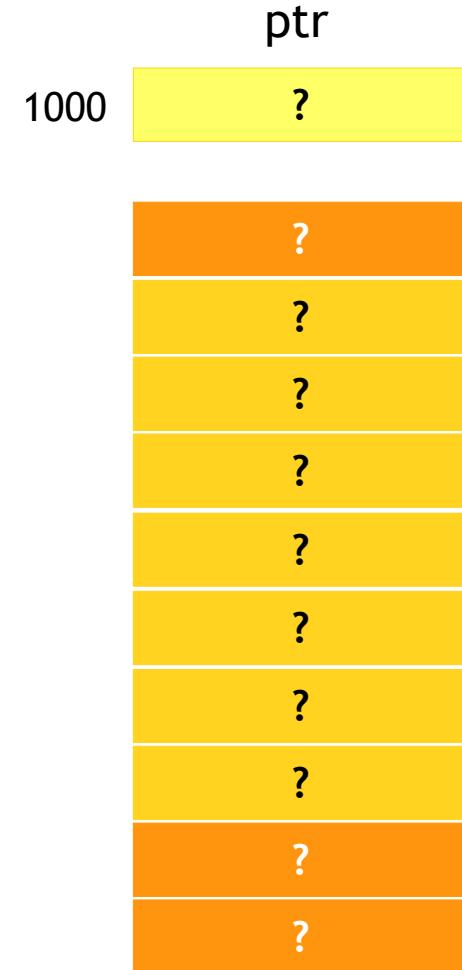
int main()
{
    → char *ptr;
    int iter;

    ptr = malloc(5);

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

    free(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Deallocation - free



### Example

```
#include <stdio.h>

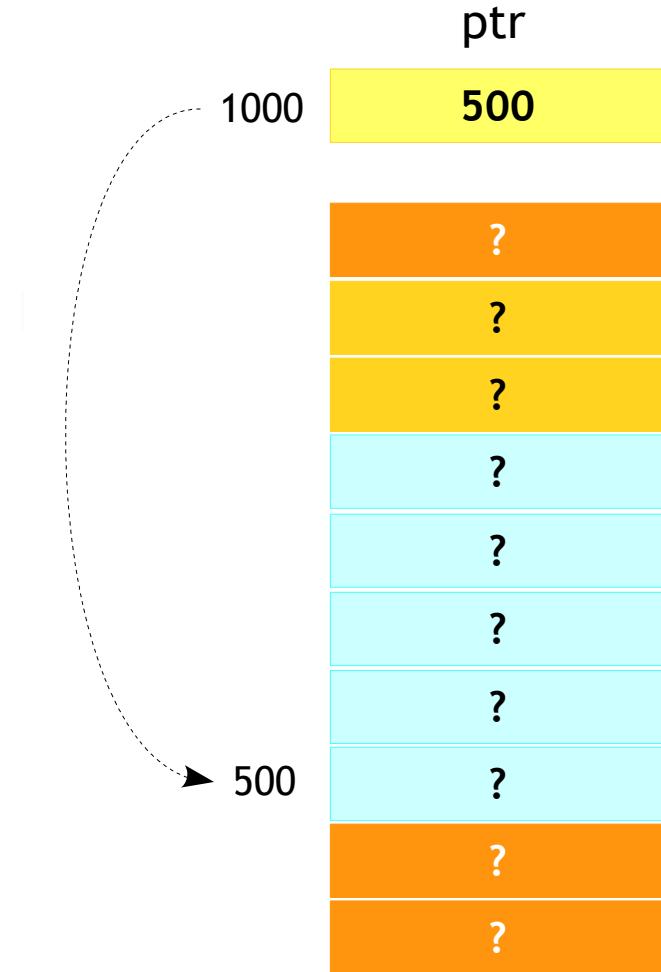
int main()
{
    char *ptr;
    int iter;

→ptr = malloc(5);

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

    free(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Deallocation - free

### Example

```
#include <stdio.h>

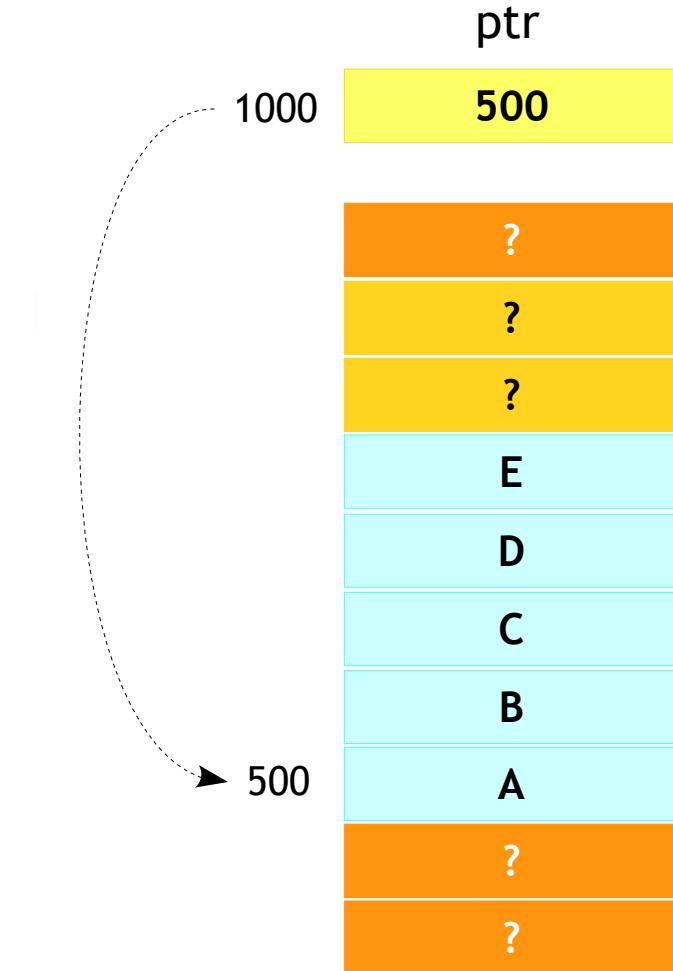
int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

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

    free(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Rule 7 - Dynamic Deallocation - free

### Example

```
#include <stdio.h>

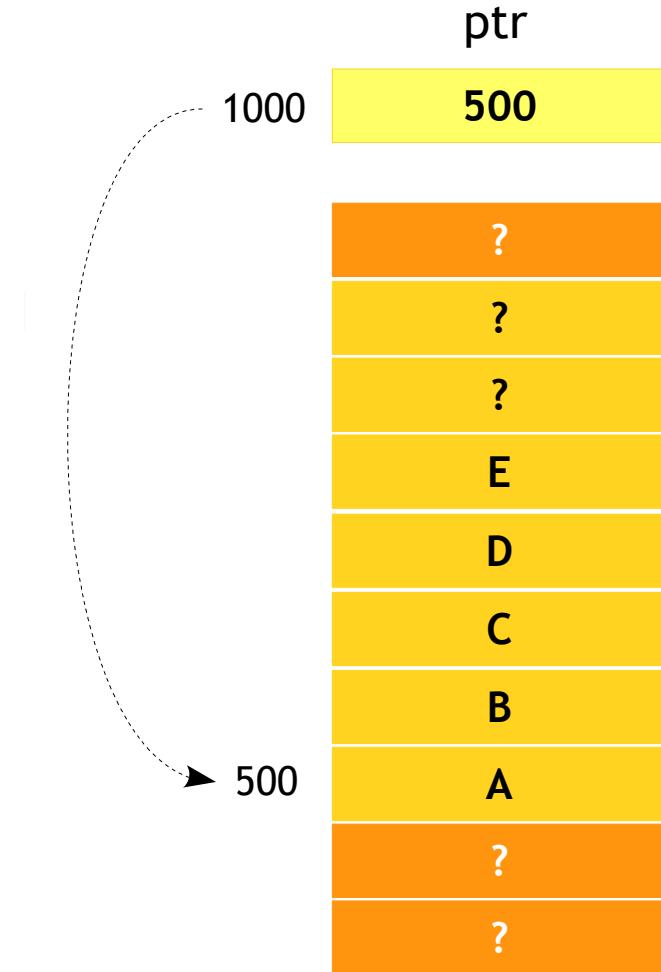
int main()
{
    char *ptr;
    int iter;

    ptr = malloc(5);

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

→ [ free(ptr); ]  ← Dashed box highlights this line

    return 0;
}
```



# Advanced 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

# Advanced C

## Pointers - Rule 7 - DIY

- Implement my\_strdup function

# Advanced C

## Pointers - Const Pointer

### Example

```
#include <stdio.h>

int main()
{
    int const *num = NULL;

    return 0;
}
```

The location, its pointing to is constant

### Example

```
#include <stdio.h>

int main()
{
    int *const num = NULL;

    return 0;
}
```

The pointer is constant

# Advanced C

## Pointers - Const Pointer



### Example

```
#include <stdio.h>

int main()
{
    const int *const num = NULL;

    return 0;
}
```

Both constants



# Advanced C

## Pointers - Const Pointer



### Example

```
#include <stdio.h>

int main()
{
    const int num = 100;
    int *iptr = &num;

    printf("Number is %d\n", *iptr);

    *iptr = 200;

    printf("Number is %d\n", num);

    return 0;
}
```



# Advanced C

## Pointers - Const Pointer



### Example

```
#include <stdio.h>

int main()
{
    int num = 100;
    const int *iptr = &num;

    printf("Number is %d\n", num);

    num = 200;

    printf("Number is %d\n", *iptr);

    return 0;
}
```



# Advanced C

## Pointers - Do's and Dont's



### Example

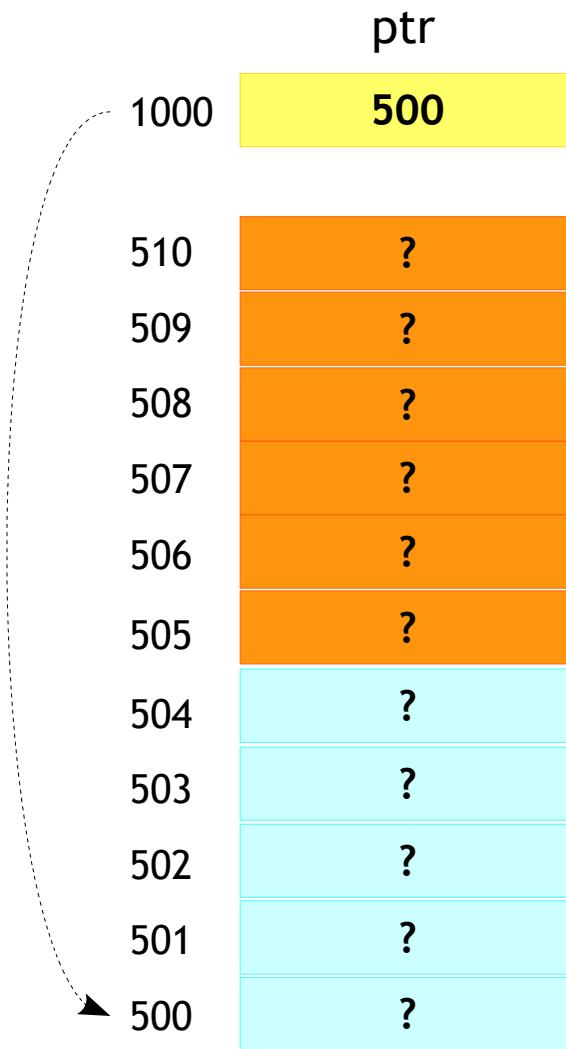
```
#include <stdio.h>

int main()
{
    → char *ptr = malloc(5);

    ptr = ptr + 10; /* Yes */
    ptr = ptr - 10; /* Yes */

    return 0;
}
```

- `malloc(5)` allocates a block of 5 bytes as shown



# Advanced C

## Pointers - Do's and Dont's



### Example

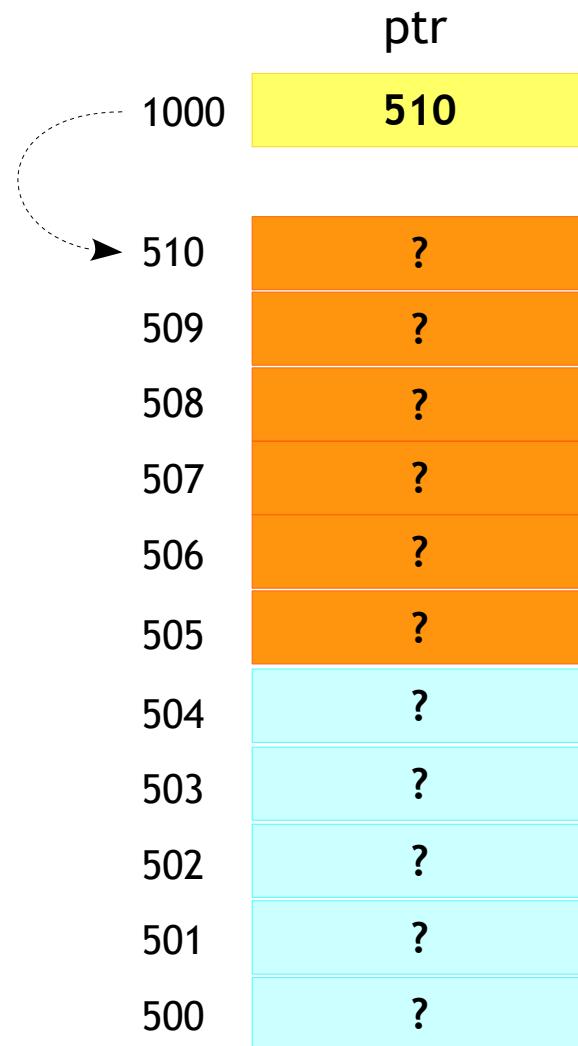
```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

→ [ptr = ptr + 10; /* Yes */
  ptr = ptr - 10; /* Yes */

    return 0;
}
```

- Adding 10 to ptr we will advance 10 bytes from the base address which is illegal but no issue in compilation!!



# Advanced C

## Pointers - Do's and Dont's



### Example

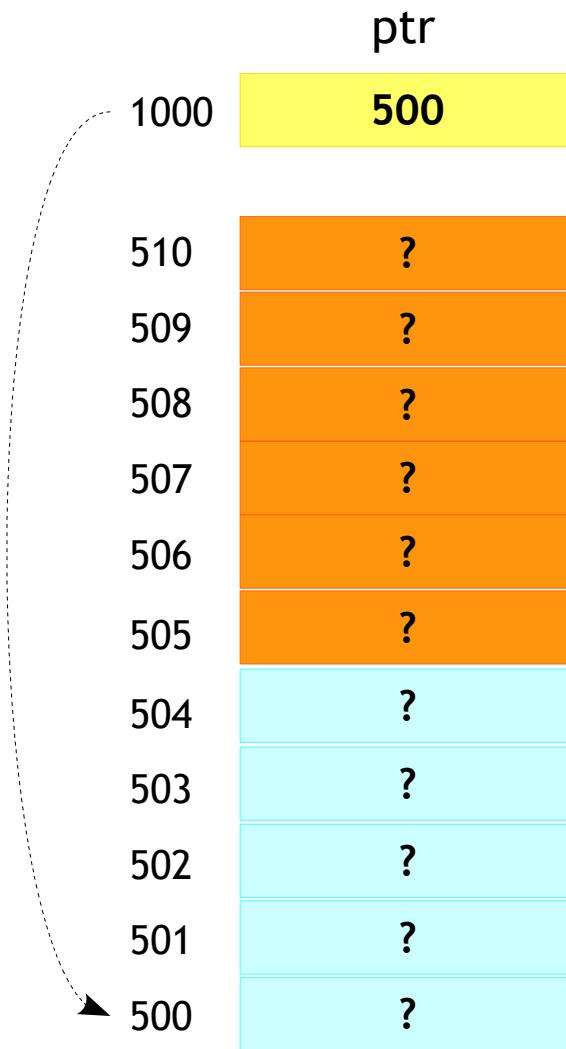
```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    → ptr = ptr + 10; /* Yes */
    [ptr = ptr - 10; /* Yes */]

    return 0;
}
```

- Subtracting 10 from ptr we will retract 10 bytes to the base address which is perfectly fine



# Advanced C

## Pointers - Do's and Dont's



### Example

```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr * 1; /* No */
    ptr = ptr / 1; /* No */

    return 0;
}
```

- All these operation on the ptr will be illegal and would lead to compiler error!!
- In fact most of the binary operator would lead to compilation error

# Advanced C

## Pointers - Do's and Dont's



### Example

```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr + ptr; /* No */
    ptr = ptr * ptr; /* No */
    ptr = ptr / ptr; /* No */

    return 0;
}
```

- All these operation on the ptr will be illegal and would lead to compiler error!!
- In fact most of the binary operator would lead to compilation error

# Advanced C

## Pointers - Do's and Dont's



### Example

```
#include <stdio.h>

int main()
{
    char *ptr = malloc(5);

    ptr = ptr - ptr;

    return 0;
}
```

- What is happening here!?
- Well the value of ptr would be 0, which is nothing but NULL (Most of the architectures) so it is perfectly fine
- The compiler would compile the code with a warning though

# Advanced C

## Pointers - Pitfalls - Segmentation Fault

- A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed.

### Example

```
#include <stdio.h>

int main()
{
    int num = 0;

    printf("Enter the number\n");
    scanf("%d", &num);

    return 0;
}
```

### Example

```
#include <stdio.h>

int main()
{
    int *num = 0;

    printf("The number is %d\n", *num);

    return 0;
}
```

# Advanced C

## Pointers - Pitfalls - Dangling Pointer

- A dangling pointer is something which does not point to a valid location any more.

### Example

```
#include <stdio.h>

int main()
{
    int *num_ptr;

    num_ptr = malloc(4);
    free(num_ptr);

    *num_ptr = 100;

    return 0;
}
```

### Example

```
#include <stdio.h>

int *foo()
{
    int num_ptr;

    return &num_ptr;
}

int main()
{
    int *num_ptr;

    num_ptr = foo();

    return 0;
}
```

# Advanced C

## Pointers - Pitfalls - Wild Pointer

- An uninitialized pointer pointing to a invalid location can be called as an wild pointer.

### Example

```
#include <stdio.h>

int main()
{
    int *num_ptr_1; /* Wild Pointer */
    static int *num_ptr_2; / Not a wild pointer */

    return 0;
}
```

# Advanced C

## Pointers - Pitfall - Memory Leak



- Improper usage of the memory allocation will lead to memory leaks
- Failing to deallocate memory which is no longer needed is one of most common issue.
- Can exhaust available system memory as an application runs longer.



# Advanced C

## Pointers - Pitfall - Memory Leak



### Example

```
#include <stdio.h>

int main()
{
    int *num_array, sum = 0, no_of_elements, iter;

    while (1)
    {
        printf("Enter the number of elements: \n");
        scanf("%d", &no_of_elements);
        num_array = malloc(no_of_elements * sizeof(int));

        sum = 0;
        for (iter = 0; iter < no_of_elements; iter++)
        {
            scanf("%d", &num_array[iter]);
            sum += num_array[iter];
        }

        printf("The sum of array elements are %d\n", sum);
        /* Forgot to free!! */
    }
    return 0;
}
```

# Advanced C

## Pointers - Pitfalls - Bus Error



- A bus error is a fault raised by hardware, notifying an operating system (OS) that, a process is trying to access memory that the CPU cannot physically address: an invalid address for the address bus, hence the name.

### Example

```
#include <stdio.h>

int main()
{
    char array[sizeof(int) + 1];
    int *ptr1, *ptr2;

    ptr1 = &array[0];
    ptr2 = &array[1];

    scanf("%d %d", ptr1, ptr2);

    return 0;
}
```

## Pointers

### Rule1 - Pointer is an integer

- Pointer is a variable that can store an address
- The data present in the pointer will be numeric data and is an address.
- Syntax to declare the pointer is:  
datatype \* pointer\_name;
- Type of the pointer is to store the address of that kind of variable.

Eg: An integer pointer is used to store the address of an integer variable and a character pointer is used to store the address of a character variable etc.

- Size of the pointer depends on the bitness of the system. It is 4 bytes in a 32 bit system and 8 bytes in a 64 bit system. This is because a variable can get the memory allocated anywhere from the memory block. The address varies from 0x00 to 0xFFFFFFFF in a 32 bit system and 0x00 to 0xFFFFFFFFFFFFFF in a 64 bit system. The pointer should be able to store any address in it.

Eg:

```
int main()
{
    int *ptr; //integer pointer meant to hold the address of an integer
    variable
    int num;
}
```



```
int main()
{
    int *ptr;
    int num;
    num = 10;
    ptr = 10;
}
```



- Both num and ptr are storing 10. The difference is 10 in num is a normal integer data and 10 in ptr is treated as an address.
- Rule 2 - Referencing and dereferencing.
- Pointers have 2 operators namely, referencing and dereferencing operator
  - Referencing operator is used to get the address of a variable. It is represented by "&".  
Eg: &ptr is used to get the address of the ptr variable
  - Dereferencing operator is used to get the value from an address. This needs to be applied only on pointer variables. This operator is represented by "\*".  
Eg: int main()

```

{
    int num = 10;
    int *ptr;
    ptr = &num;
    printf("%d\n", *ptr);
    return 0;
}

```



- In the above example, num has 10 at address 1000. &num should give 1000 which is stored in ptr. As ptr is having 1000 stored, doing \*ptr means \*1000 i.e go to address 1000 and fetch the value.

```

Eg2:      int main()
{
    int num = 10;
    int *ptr;
    ptr = &num;
    printf("%d\n", *ptr);
    *ptr = 20;
    printf("%d\n", num);
    return 0;
}

```

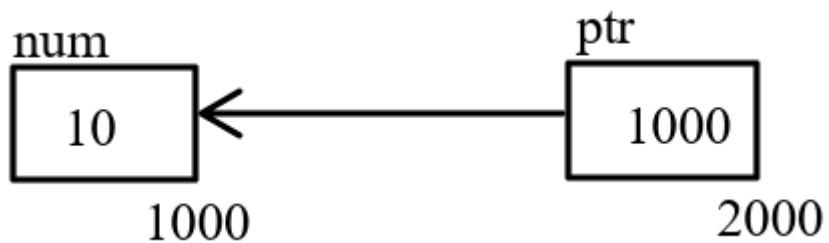
- When \*ptr = 20 is done, it means \*1000 = 20 i.e go to address 1000 and write 20. This inturn changes the value present in num because it is present in address 1000.

### Rule 3 - Pointing means containing

- Whenever there is a pointer variable, it has an address stored inside it. And the pointer will be pointing to that address.

Eg: int main()

```
{  
    Int num = 10;  
    int *ptr;  
    ptr = &num;  
    return 0;  
}
```

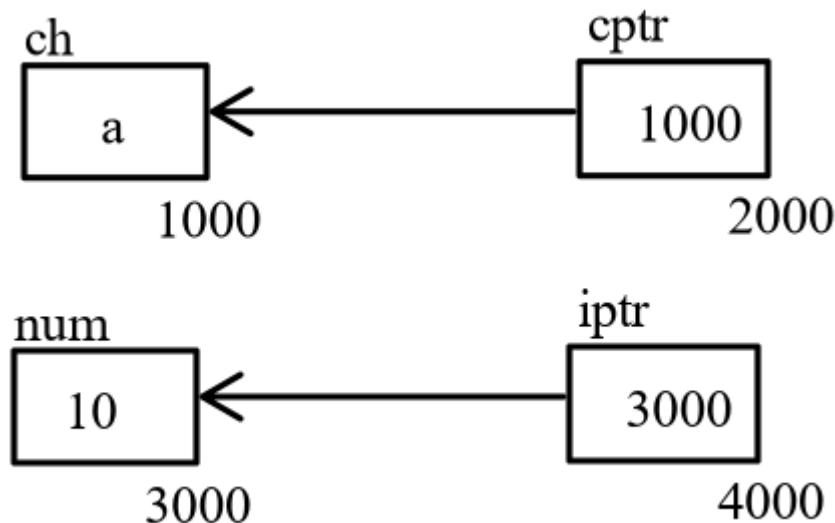


#### Rule 4: Pointer type

- When all the pointers are getting the same bytes of memory allocated, why do we need a type to be specified?
- Let us consider an example. Assume there is a character pointer and an integer pointer.

Eg: int main()

```
{  
    char ch = 'a';  
    int num = 10;  
    char *cptr = &ch;// similar to char *cptr; cptr = &ch;  
    int *iptr = &num; // similar to int *iptr; iptr = &num;  
    return 0;  
}
```



- Generally, can we say that the address of a character variable is 1 byte and the address of integer variable is 4 bytes? No because the size of the address depends on the bitness of the system
- Looking just into the address, can we say what kind of data is present in that address? No.
- Now whenever we dereference the pointer, how many bytes of data does it fetch or write from the given address?
- It all depends on the type of the pointer.

- An integer pointer will fetch sizeof(int) number of bytes from the given address. A character pointer will fetch sizeof(char) number of bytes.
- Hence type of the pointer will say the pointer to fetch sizeof(datatype) number of bytes.

#### Rule 4 in detail: Endianness of a system

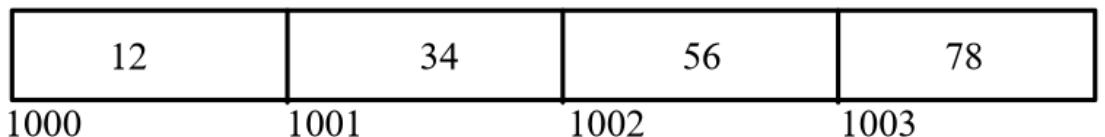
- Byte ordering of the data in the system is called endianness. There are two ways how bytes of data are stored in the system. They are little and big endian systems.
- In the little endian system, the least significant byte is stored in the lowest address and in the big endian system, the most significant byte is stored in the lowest address.

Eg: int num = 0x12345678;

LittleEndian:



BigEndian:



Eg: int main()

```

{
    int num = 0x12345678;
    int *iptr = &num;
    char *cptr = &num;
    *cptr = 0xAB;
    printf("%x", *iptr);
    return 0;
}

```

- In the above example, when char \*cptr = &num is done, it throws a warning as we are trying to store the address of an integer. This can be avoided by typecasting.
- printf gives 123456AB as output in little endian system and AB345678 in big endian system. This is because, when \*cptr = 0xAB is done, cptr is pointing to

address 1000. Hence it changes the data in address 1000. When the data is fetched after that, in little endian system as the least significant byte is changed, we get the output as 0x123456AB and in big endian system as the most significant bit is changed, we get output as 0xAB345678.

#### Rule 5: Pointer arithmetic

- Type of the pointer is also necessary to perform pointer arithmetic.
- Pointer arithmetic always depends on `sizeof(pointer_type)`
- Assume an integer pointer `ptr`. When pointer arithmetic is performed, it depends on `sizeof(int)`.  
Eg:  

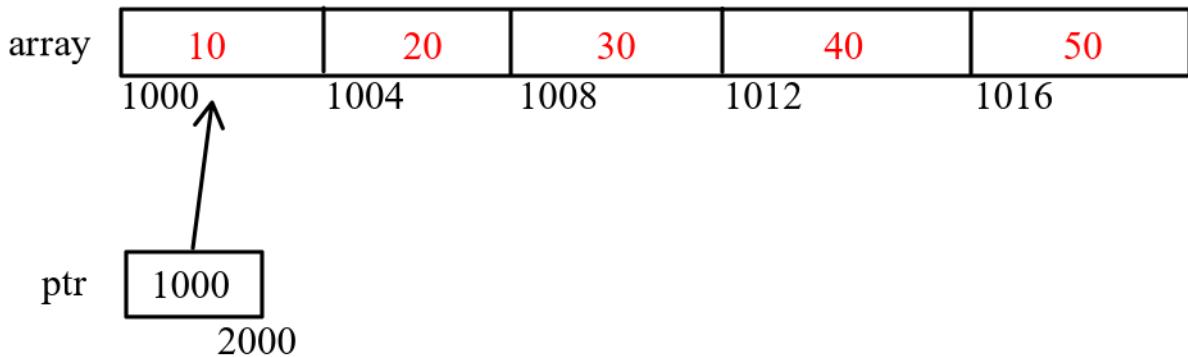
```
int num;
int *ptr = &num;
ptr++;
```
- If `ptr` is having address 1000, when `ptr++` is done, it gets changed to 1004 as shown below  
`ptr++ is  $ptr = ptr + 1$ ;`  
Pointer arithmetic depends on `sizeof(type of the pointer)`. Hence,  
$$ptr = ptr + 1 * sizeof(int)$$
$$ptr = 1000 + 1 * 4$$
$$ptr = 1004.$$

#### Rule 5 in detail: Passing an array to a function

- Name of the array interprets 2 things.
- First, name of the array is interpreted as the whole address of the array when used in `sizeof()` operator and when used after `&` operator.
- Rest all other cases i.e when it is being assigned to a pointer or passed as an argument to another function, it is interpreted as the base address of the array.

Eg:

```
int main()
{
    int array[5] = {10, 20, 30, 40, 50};
    int *ptr = array;
    printf("%d\n", *ptr);
    ptr++;
    printf("%d\n", *(ptr + 2));
    return 0;
}
```



- `printf("%d\n", *ptr);` \*ptr is \*1000. This gives 10 as output.
- `ptr++` is `ptr = ptr + 1 ;`  
`ptr = ptr + 1 * sizeof(type of the pointer);`  
`ptr = 1000 + 1 * sizeof(int);`  
`ptr = 1004`
- `*(ptr + 2)` is `*(ptr + 2 * sizeof(type of the pointer))`  
`*(ptr + 2 * sizeof(int))`  
`*(1004 + 2 * 4)`  
`*1012`  
 Hence `printf("%d\n", *(ptr + 2));` gives 40 as it has to go to address 1012 and fetch an integer.
- Different ways of printing array's elements are,
  - `array[i];` This is interpreted by the compiler as `*(array + i)`. Both array and ptr represent the base address. array represents address 1000. Hence `array + i` is also pointer arithmetic and depends on `sizeof(type of the array)`.
  - `i[array];` // `*(array + i)` is same as `*(i + array)`
  - `ptr[i] ;` // same as `*(ptr + i)`
  - `i[ptr];`

#### Rule 6: Pointing to nothing

- If there is an uninitialized pointer variable, it can be pointing to some random address. When we later try dereferencing, it might give some garbage value or it might also lead to segmentation fault.
- Instead of making the pointer fail silently, good practice is to have a defined behaviour.
- Whenever the pointer is declared, it can be initialised with a null pointer constant called `NULL`.
- `NULL` might represent address 0 (actually `(void *)0`) or other address depending on the operating system. The pointers which have `NULL` stored in them are called Null pointers. These pointers always give a fixed result when dereferenced i.e segmentation fault.

Eg: `int main()`

```

{
    int *ptr = NULL;
    printf("%d\n", *ptr); // gives segmentation fault
}

```

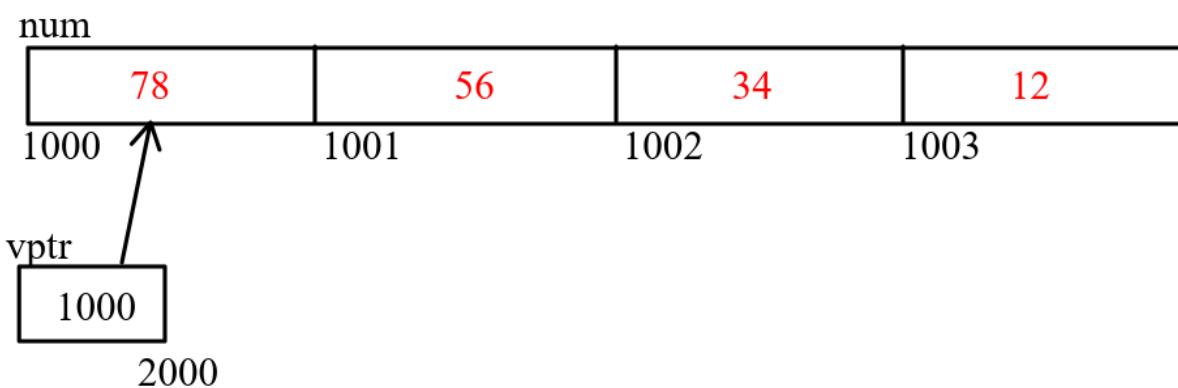
- Uses of Null pointer are to indicate the linked list termination, failure of malloc calloc etc.
- There are generic pointers which can hold the address of any variable. They are called void pointers.
- The syntax to declare void pointer is,  
void \*vptr; // vptr is the name of the pointer
- Void pointers cannot be dereferenced. Because when \*vptr is said, compiler will not know what kind of data should be fetched. Hence typecasting before dereferencing is necessary.
- \*(int \*)vptr is a way of typecasting. This means whatever address vptr is holding should be treated as an address of integer and then dereferenced.  
Now this will dereference 4 bytes of data and gives an integer
- Pointer arithmetic on void pointer is compiler implementation dependent.  
This is allowed in gcc because sizeof(void) is 1.

Eg: int main()

```

{
    int num = 0x12345678;
    void *vptr = &num;
    printf("%x\n", *(int *)vptr);
    printf("%hx\n", *(char *)(vptr + 1));
    printf("%hx\n", *((short *)vptr + 1));
    return 0;
}

```



- \*(int \*)vptr  
\*(int \*)1000 // treat address 1000 as address of integer and dereference  
Hence %x prints 12345678
- \*(char \*)(vptr + 1)  
\*(char \*)(vptr + 1 \* sizeof(void)) // because pointer arithmetic is applied on vptr

```
*(char *)(1000 + 1 * 1)
```

```
*(char *)(1001)
```

Address 1001 is treated as address of the character and 1 byte is fetched. %x is used to print the hexadecimal equivalent of data in 4 bytes. %hx is used to print the hexadecimal equivalent of the data in 2 bytes and %hhx to print the hexadecimal equivalent from 1 byte of data.

Hence this gives output as 56

```
> *((short *)vptr + 1)  
*((short *)vptr + 1 * sizeof(short)) // because typecasting is done first then  
pointer arithmetic  
*((short *) 1000 + 1 * 2)  
*((short *)1002)
```

Output of third printf is 1234 (because of the little endian system)

Rule 7 : Dynamic memory allocation

# Standard I/O



# Advanced C

## Standard I/O - Why should I know this? - DIY

### Screen Shot

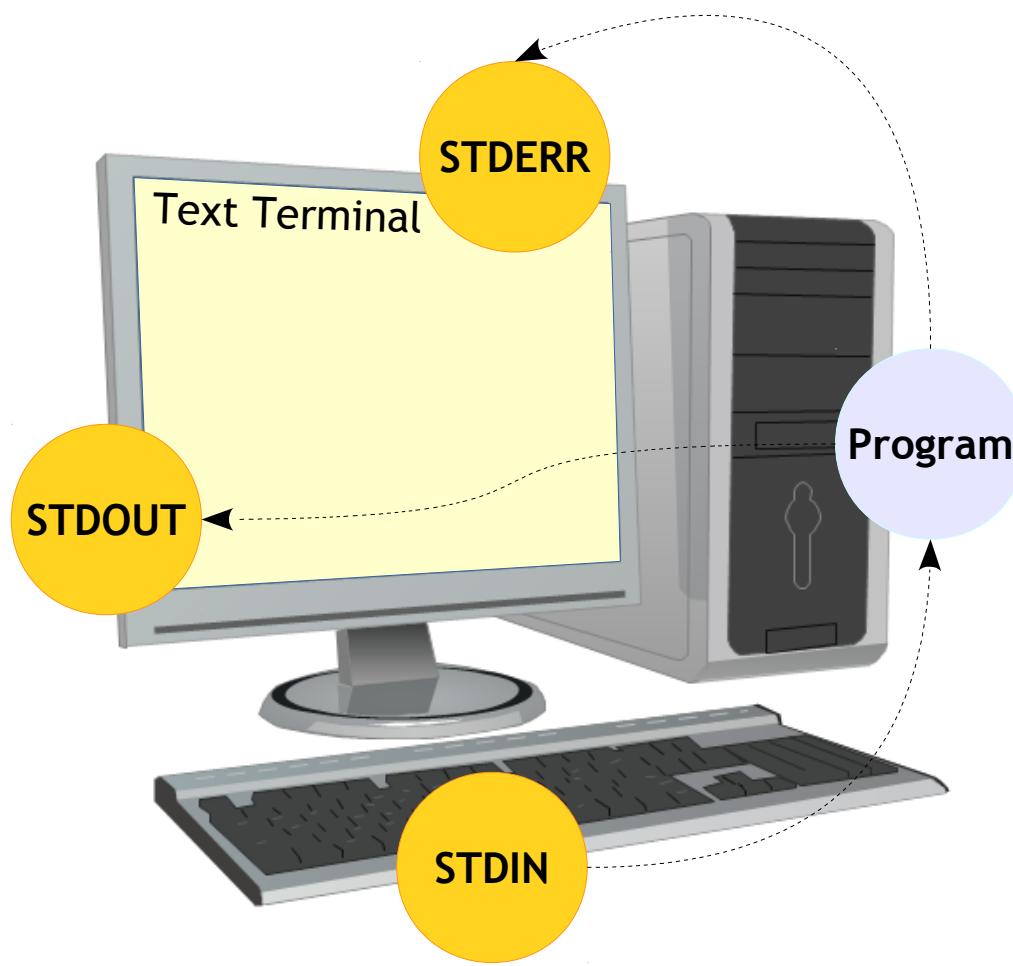
```
user@user:~]
user@user:~]./print_bill.out
Enter the item 1: Kurkure
Enter no of pcs: 2
Enter the cost : 5
Enter the item 2: Everest Paneer Masala
Enter no of pcs: 1
Enter the cost : 25.50
Enter the item 3: India Gate Basmati
Enter no of pcs: 1
Enter the cost : 1050.00
```

S.No	Name	Quantity	Cost	Amount
1.	Kurkure	2	5.00	10.00
2.	Everest Paneer	1	25.50	25.50
2.	India Gate Bas	1	1050.00	1050.00
Total		4	1085.50	

```
user@user:~]
```

# Advanced C

## Standard I/O



# Advanced C

## Standard I/O - The File Descriptors



- OS uses 3 file descriptors to provide standard input output functionalities
  - 0 → stdin
  - 1 → stdout
  - 2 → stderr
- These are sometimes referred as “The Standard Streams”
- The IO access example could be
  - stdin → Keyboard, pipes
  - stdout → Console, Files, Pipes
  - stderr → Console, Files, Pipes



# Advanced C

## Standard I/O - The File Descriptors



- Wait!!, did we see something wrong in previous slide?  
Both stdout and stderr are similar ?
- If yes why should we have 2 different streams?
- The answer is convenience and urgency.
  - Convenience : Diagnostic information can be printed on stderr. Example - we can separate error messages from low priority informative messages
  - Urgency : serious error messages shall be displayed on the screen immediately
- So how the C language help us in the standard IO?



# Advanced C

## Standard I/O - The header file



- You need to refer input/output library function

`#include <stdio.h>`

- When the reference is made with “*<name>*” the search for the files happen in standard path
- Header file Vs Library

# Advanced C

## Standard I/O - Unformatted (Basic)



- Internal binary representation of the data directly between memory and the file
- Basic form of I/O, simple, efficient and compact
- Unformatted I/O is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor
- `getchar()` and `putchar()` are two functions part of standard C library
- Some functions like `getch()`, `getche()`, `putch()` are defined in `conio.h`, which is not a standard C library header and is not supported by the compilers targeting Linux / Unix systems

# Advanced C

## Standard I/O - Unformatted (Basic)



### 001\_example.c

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

int main()
{
    int ch;

    for ( ; (ch = getchar()) != EOF; )
    {
        putchar(toupper(ch));
    }

    puts("EOF Received");

    return 0;
}
```



# Advanced C

## Standard I/O - Unformatted (Basic)

### 002\_example.c

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

int main()
{
    int ch;

    for ( ; (ch = getc(stdin)) != EOF; )
    {
        putc(toupper(ch), stdout);
    }

    puts("EOF Received");

    return 0;
}
```

# Advanced C

## Standard I/O - Unformatted (Basic)



### 003\_example.c

```
#include <stdio.h>

int main()
{
    char str[10];

    puts("Enter the string");
    gets(str);
    puts(str);

    return 0;
}
```



# Advanced C

## Standard I/O - Unformatted (Basic)



### 004\_example.c

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

int main()
{
    char str[10];

    puts("Enter the string");
    fgets(str, 10, stdin);
    puts(str);

    return 0;
}
```



# Advanced C

## Standard I/O - Formatted



- Data is formatted or transformed
- Converts the internal binary representation of the data to ASCII before being stored, manipulated or output
- Portable and human readable, but expensive because of the conversions to be done in between input and output
- The printf() and scanf() functions are examples of formatted output and input

# Advanced C

## Standard I/O - printf()

005\_example.c

```
#include <stdio.h>

int main()
{
    char a[8] = "Emertxe";

    printf(a);

    return 0;
}
```

- What will be the output of the code on left side?
- Is that syntactically OK?
- Lets understand the printf() prototype
- Please type

man printf

on your terminal

# Advanced C

## Standard I/O - printf()



### Prototype

```
int printf(const char *format, ...);  
or  
int printf("format string", [variables]);
```

where format string arguments can be  
%[flags] [width] [.precision] [length] type\_specifier

%typeSpecifier is mandatory and others are optional

- Converts, formats, and prints its arguments on the standard output under control of the format
- Returns the number of characters printed

# Advanced C

## Standard I/O - printf()



### Prototype

```
int printf(const char *format, ...);
```

→ *What is this!?*

What is this!?



# Advanced C

## Standard I/O - printf()



### Prototype

```
int printf(const char *format, ...);
```

What is this!?

- Is called as ellipses
- Means, you can pass any number (i.e 0 or more) of “optional” arguments of any type
- So how to complete the below example?

### Example

```
int printf("%c %d %f", );
```

What should be written here and how many?



# Advanced C

## Standard I/O - printf()

### Example

```
int printf("%c %d %f", arg1, arg2, arg3);
```

Now, how do you decide this!?

Based on the number of format specifiers

- So the **number of arguments** passed to the printf function should exactly **match the number of format specifiers**
- So lets go back the code again

# Advanced C

## Standard I/O - printf()

### Example

```
#include <stdio.h>

int main()
{
    char a[8] = "Emertxe";
    printf(a);
    return 0;
}
```

Isn't this a string?

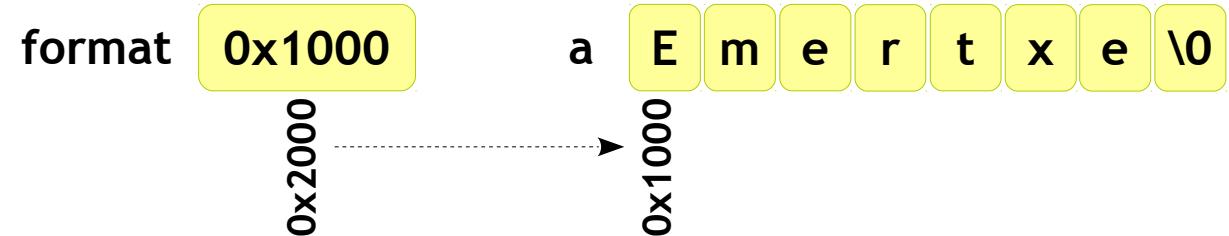
And strings are nothing but array of characters terminated by null

So what get passed, while passing a array to function?

`int printf(const char *format, ...);`

Isn't this a pointer?

So a pointer hold a address, can be drawn as



So the base address of the array gets passed to the pointer, Hence the output

Note: You will get a warning while compiling the above code.  
So this method of passing is not recommended

# Advanced C

## Standard I/O - printf() - Type Specifiers

### 006\_example.c

Specifiers	Example	Expected Output
%c	printf("%c", 'A')	A
%d %i	printf("%d %i", 10, 10)	10 10
%o	printf("%o", 8)	10
%x %X	printf("%x %X %x", 0xA, 0xA, 10)	a A a
%u	printf("%u", 255)	255
%f %F	printf("%f %F", 2.0, 2.0)	2.000000 2.000000
%e %E	printf("%e %E", 1.2, 1.2)	1.200000e+00 1.200000E+00
%a %A	printf("%a", 123.4)	0x1.ed9999999999ap+6
	printf("%A", 123.4)	0X1.ED9999999999AP+6
%g %G	printf("%g %G", 1.21, 1.0)	1.21 1
%s	printf("%s", "Hello")	Hello

# Advanced C

## Standard I/O - printf() - Type Length Specifiers

### 007\_example.c

Length specifier	Example	Example
%[h]X	printf("%hX", 0xFFFFFFFF)	FFFF
%[l]X	printf("%lX", 0xFFFFFFFFL)	FFFFFFF
%[ll]X	printf("%llx", 0xFFFFFFFFFFFFFFFF)	FFFFFFFFFFFFFF
%[L]f	printf("%Lf", 1.23456789L)	1.234568

# Advanced C

## Standard I/O - printf() - Width



### 008\_example.c

Width	Example	Expected Output
%[x]d	<code>printf("%3d %3d", 1, 1)</code> <code>printf("%3d %3d", 10, 10)</code> <code>printf("%3d %3d", 100, 100)</code>	1 1 10 10 100 100
%[x}s	<code>printf("%10s", "Hello")</code> <code>printf("%20s", "Hello")</code>	Hello Hello
%*[specifier]	<code>printf("%*d", 1, 1)</code> <code>printf("%*d", 2, 1)</code> <code>printf("%*d", 3, 1)</code>	1 1 1

# Advanced C

## Standard I/O - printf() - Precision



### 009\_example.c

Precision	Example	Expected Output
%[x].[x]d	printf("%3.1d", 1) printf("%3.2d", 1) printf("%3.3d", 1)	1 01 001
%0.[x]f	printf("%0.3f", 1.0) printf("%0.10f", 1.0)	1.000 1.0000000000
%[x].[x]s	printf("%12.8s", "Hello World")	Hello Wo

# Advanced C

## Standard I/O - printf() - Flags



### 010\_example.c

Flag	Example	Expected Output
%[#]x	<code>printf("%#x %#X %#x", 0xA, 0xA, 10)</code> <code>printf("%#o", 8)</code>	0xa 0XA 0xa 010
%[-x]d	<code>printf("%-3d %-3d", 1, 1)</code> <code>printf("%-3d %-3d", 10, 10)</code> <code>printf("%-3d %-3d", 100, 100)</code>	1 1 10 10 100 100
%[ ]3d	<code>printf("% 3d", 100)</code> <code>printf("% 3d", -100)</code>	100 -100

# Advanced C

## Standard I/O - printf() - Escape Sequence

### 011\_example.c

Escape Sequence	Meaning	Example	Expected Output
\n	New Line	<code>printf("Hello World\n")</code>	Hello World (With a new line)
\r	Carriage Return	<code>printf("Hello\rWorld")</code>	World
\t	Tab	<code>printf("Hello\tWorld")</code>	Hello World
\b	Backspace	<code>printf("Hello\bWorld")</code>	HellWorld
\v	Vertical Tab	<code>printf("Hello\vWorld")</code>	Hello World
\f	Form Feed	<code>printf("Hello World\f")</code>	Might get few extra new line(s)
\e	Escape	<code>printf("Hello\eWorld")</code>	Helloorld
\\"		<code>printf("A\\B\\C")</code>	A\B\C
\"		<code>printf("\\"Hello World\\\"")</code>	"Hello World"

# Advanced C

## Standard I/O - printf()



- So in the previous slides we saw some 80% of printf's format string usage.

What?? Ooh man!!.. Now how to print **80%??**



# Advanced C

## Standard I/O - printf() - Example

012\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string[] = "Hello World";

    printf("%d %c %f %s\n", num1, ch, num2, string);
    printf("%+05d\n", num1);
    printf("%.2f %.5s\n", num2, string);

    return 0;
}
```

# Advanced C

## Standard I/O - printf() - Return

013\_example.c

```
#include <stdio.h>

int main()
{
    int ret;
    char string[] = "Hello World";

    ret = printf("%s\n", string);

    printf("The previous printf() printed %d chars\n", ret);

    return 0;
}
```

# Advanced C

## Standard I/O - sprintf() - Printing to string



### Prototype

```
int sprintf(char *str, const char *format, ...);
```

- Similar to printf() but prints to the buffer instead of stdout
- Formats the arguments in arg1, arg2, etc., according to format specifier
- buffer must be big enough to receive the result

# Advanced C

## Standard I/O - sprintf() - Example

014\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string1[] = "sprintf() Test";
    char string2[100];

    sprintf(string2, "%d %c %f %s\n", num1, ch, num2, string1);
    printf("%s", string2);

    return 0;
}
```

# Advanced C

## Standard I/O - Formatted Input - scanf()



### Prototype

```
int scanf(char *format, ...);  
or  
int scanf("string", [variables]);
```

- Reads characters from the standard input, interprets them according to the format specifier, and stores the results through the remaining arguments.
- Almost all the format specifiers are similar to printf() except changes in few
- Each “optional” argument must be a **pointer**

# Advanced C

## Standard I/O - Formatted Input - scanf()

- It returns as its value the number of successfully matched and assigned input items.
- On the end of file, EOF is returned. Note that this is different from 0, which means that the next input character does not match the first specification in the format string.
- The next call to scanf() resumes searching immediately after the last character already converted.

# Advanced C

## Standard I/O - scanf() - Example

015\_example.c

```
#include <stdio.h>

int main()
{
    int num1;
    char ch;
    float num2;
    char string[10];

    scanf("%d %c %f %s", &num1, &ch, &num2, string);
    printf("%d %c %f %s\n", num1, ch, num2, string);

    return 0;
}
```

# Advanced C

## Standard I/O - scanf() - Format Specifier



### 016\_example.c

Flag	Examples	Expected Output
%*[specifier]	<code>scanf("%d%*c%d%*c%d", &amp;h, &amp;m, &amp;s)</code>	User Input → HH:MM:SS Scanned Input → HHMMSS
		User Input → 5+4+3 Scanned Input → 543

# Advanced C

## Standard I/O - scanf() - Format Specifier

### 017\_example.c

Flag	Examples	Expected Output
%[]	<code>scanf("%[a-z A-Z]", name)</code>	User Input → Emertxe Scanned Input → Emertxe
	<code>scanf("%[0-9]", id)</code>	User Input → Emx123 Scanned Input → Emx
		User Input → 123 Scanned Input → 123
		User Input → 123XYZ Scanned Input → 123

# Advanced C

## Standard I/O - scanf() - Return

### 018\_example.c

```
#include <stdio.h>

int main()
{
    int num = 100, ret;

    printf("The enter a number [is 100 now]: ");
    ret = scanf("%d", &num);

    if (ret != 1)
    {
        printf("Invalid input. The number is still %d\n", num);
        return 1;
    }
    else
    {
        printf("Number is modified with %d\n", num);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - sscanf() - Reading from string



### Prototype

```
int sscanf(const char *string, const char *format, ...);
```

- Similar to scanf() but read from string instead of stdin
- Formats the arguments in arg1, arg2, etc., according to format

# Advanced C

## Standard I/O - sscanf() - Example

019\_example.c

```
#include <stdio.h>

int main()
{
    int age;
    char array_1[10];
    char array_2[10];

    sscanf("I am 30 years old", "%s %s %d", array_1, array_2, &age);
    sscanf("I am 30 years old", "%*s %*s %d", &age);
    printf("OK you are %d years old\n", age);

    return 0;
}
```

# Advanced C

## Standard I/O - DIY

### Screen Shot

```
user@user:~]
user@user:~]./print_bill.out
Enter the item 1: Kurkure
Enter no of pcs: 2
Enter the cost : 5
Enter the item 2: Everest Paneer Masala
Enter no of pcs: 1
Enter the cost : 25.50
Enter the item 3: India Gate Basmati
Enter no of pcs: 1
Enter the cost : 1050.00
```

S.No	Name	Quantity	Cost	Amount
1.	Kurkure	2	5.00	10.00
2.	Everest Paneer	1	25.50	25.50
3.	India Gate Bas	1	1050.00	1050.00
Total		4	1085.50	

```
user@user:~]
```

# Advanced C

## Standard I/O - DIY



### Screen Shot

```
user@user:~]
user@user:~]./progress_bar.out
Loading [-----] 50%
user@user:~]
```

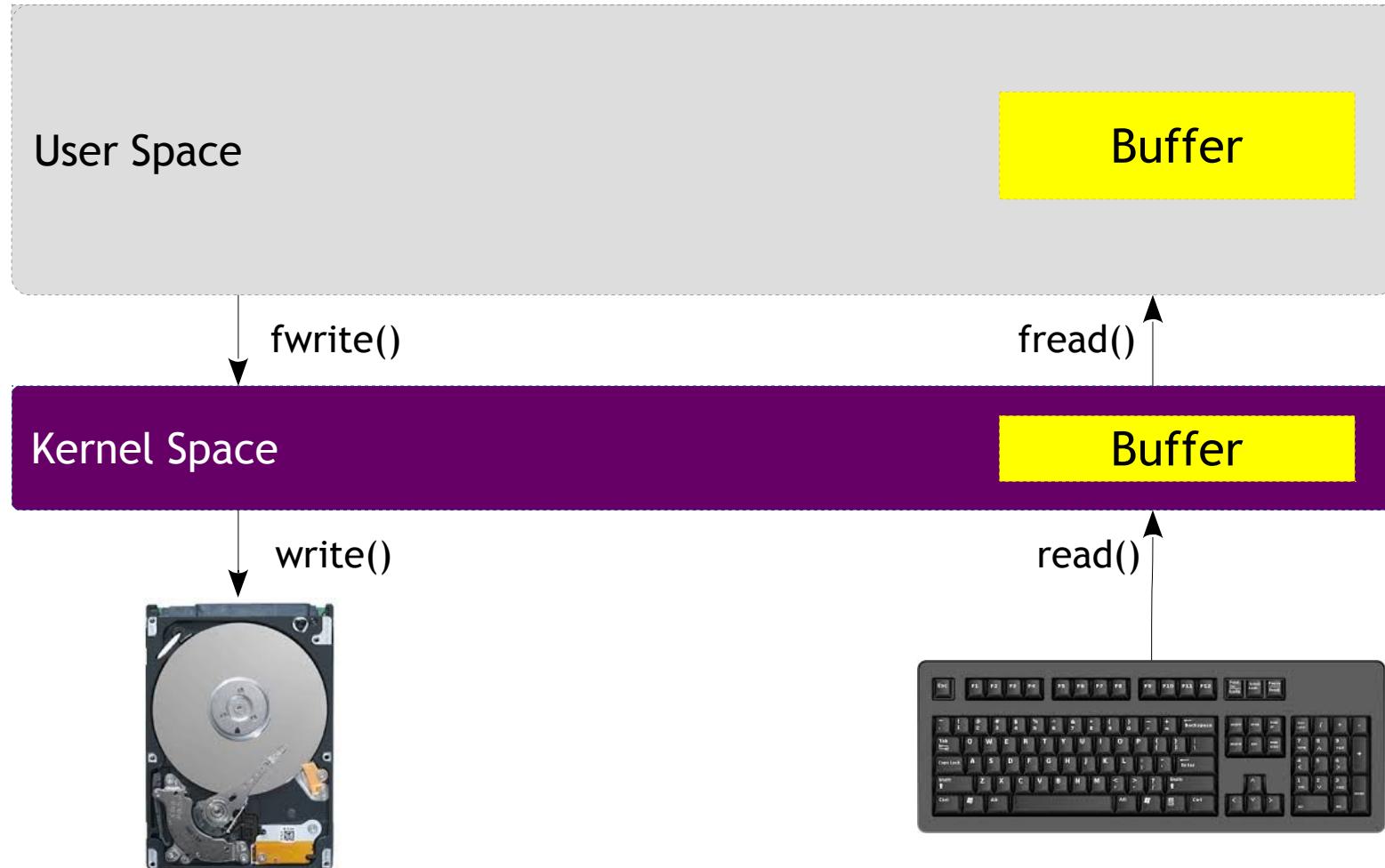
# Advanced C

## Standard I/O - Buffering



# Advanced C

## Standard I/O - Buffering



# Advanced C

## Standard I/O - User Space Buffering



- Refers to the technique of temporarily storing the results of an I/O operation in user-space before transmitting it to the kernel (in the case of writes) or before providing it to your process (in the case of reads)
- This technique minimizes the number of system calls (between user and kernel space) which may improve the performance of your application



# Advanced C

## Standard I/O - User Space Buffering



- For example, consider a process that writes one character at a time to a file. This is obviously inefficient: Each write operation corresponds to a `write()` system call
- Similarly, imagine a process that reads one character at a time from a file into memory!! This leads to `read()` system call
- I/O buffers are temporary memory area(s) to moderate the number of transfers in/out of memory by assembling data into batches

# Advanced C

## Standard I/O - Buffering - stdout



- The output buffer get flushed out due to the following reasons
  - Normal Program Termination
  - '\n' in a printf
  - Read
  - fflush call
  - Buffer Full

# Advanced C

## Standard I/O - Buffering - stdout

020\_example.c

```
#include <stdio.h>

int main()
{
    printf("Hello");

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdout



### 021\_example.c

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

int main()
{
    while (1)
    {
        printf("Hello");
        sleep(1);
    }

    return 0;
}
```

### Solution

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

int main()
{
    while (1)
    {
        printf("Hello\n");
        sleep(1);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdout



### 022\_example.c

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

int main()
{
    int num;

    while (1)
    {
        printf("Enter a number: ");
        scanf("%d", &num);
    }

    return 0;
}
```



# Advanced C

## Standard I/O - Buffering - stdout



### 023\_example.c

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

int main()
{
    while (1)
    {
        printf("Hello");
        fflush(stdout);
        sleep(1);
    }

    return 0;
}
```



# Advanced C

## Standard I/O - Buffering - stdout

### 024\_example.c

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

int main()
{
    char str[BUFSIZ] = "1";

    while (1)
    {
        printf("%s", str);
        sleep(1);
    }

    return 0;
}
```

### Solution

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

int main()
{
    char str[BUFSIZ] = "1";

    setbuf(stdout, NULL);

    while (1)
    {
        printf("%s", str);
        sleep(1);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdin



- The input buffer generally gets filled till the user presses and enter key or end of file.
- The complete buffer would be read till a '\n' or and EOF is received.



# Advanced C

## Standard I/O - Buffering - stdin



### 025\_example.c

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != '\n')
    {
        scanf("%c", &ch);

        printf("%c", ch);
    }

    return 0;
}
```



# Advanced C

## Standard I/O - Buffering - stdin

### Solution 1

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != '\n')
    {
        scanf("%c", &ch);
        __fpurge(stdin);
        printf("%c", ch);
    }

    return 0;
}
```

### Solution 2

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != '\n')
    {
        scanf("%c", &ch);
        while (getchar() != '\n');
        printf("%c", ch);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stderr

- The stderr file stream is unbuffered.

### 026\_example.c

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

int main()
{
    while (1)
    {
        fprintf(stdout, "Hello");
        fprintf(stderr, "World");

        sleep(1);
    }

    return 0;
}
```

# Strings



# Advanced C

S      s - Fill in the blanks please ;)

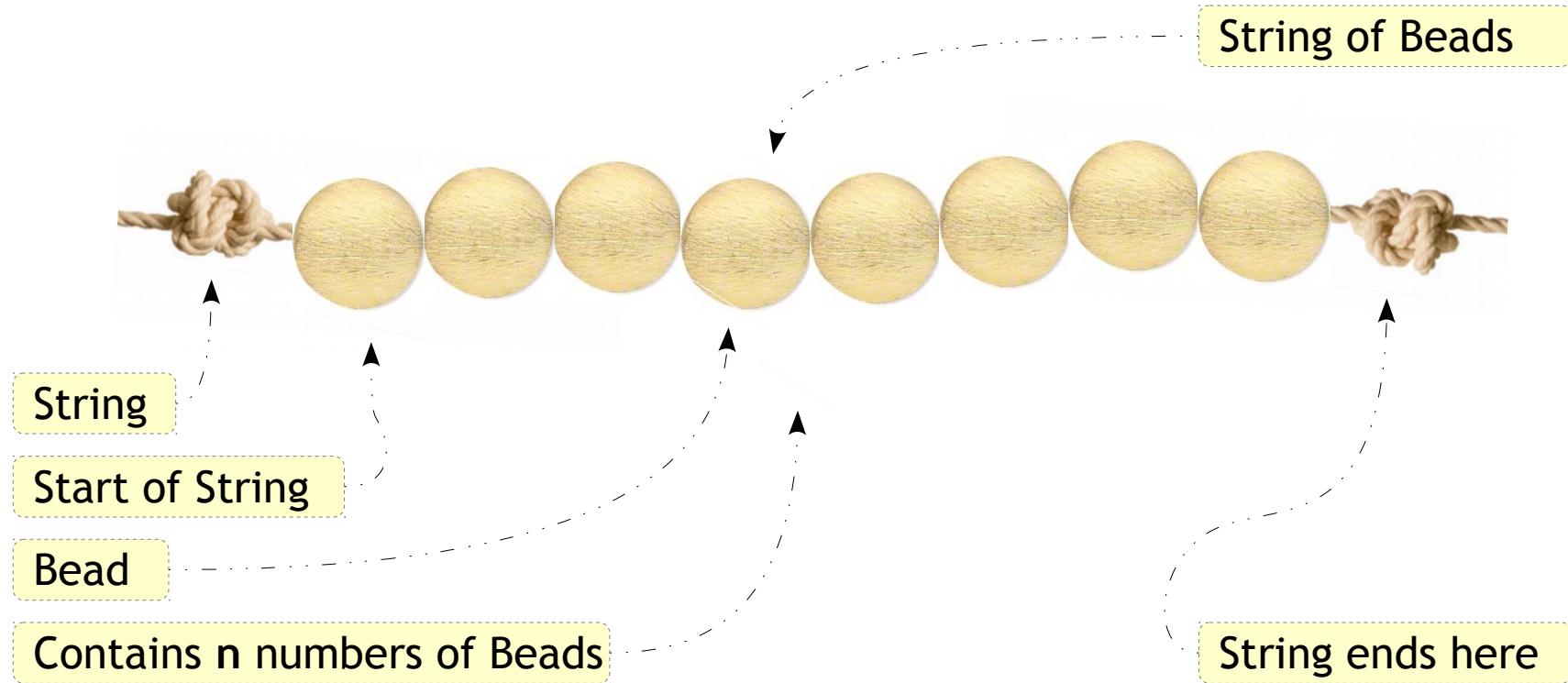


# Advanced C Strings



A set of things tied or threaded together on a thin cord

Source: Google



# Advanced C

## Strings



- Contiguous sequence of characters
- Stores printable ASCII characters and its extensions
- End of the string is marked with a special character, the null character '\0'
- '\0' is implicit in strings enclosed with “”
- Example

“You know, now this is what a string is!”



# Advanced C

## Strings



- Constant string
  - Also known as string literal
  - Such strings are read only
  - Usually, stored in read only (code or text segment) area
  - String literals are shared
- Modifiable String
  - Strings that can be modified at run time
  - Usually, such strings are stored in modifiable memory area (data segment, stack or heap)
  - Such strings are not shared

# Advanced C

## Strings - Initialization

### 001\_example.c

```
char char_array[5] = {'H', 'E', 'L', 'L', 'O'}; ← Character Array
```

```
char str1[6] = {'H', 'E', 'L', 'L', 'O', '\0'}; ← String
```

```
char str2[] = {'H', 'E', 'L', 'L', 'O', '\0'}; ← Valid
```

```
char str3[6] = {"H", "E", "L", "L", "O"}; ← Invalid
```

```
char str4[6] = {"H" "E" "L" "L" "O"}; ← Valid
```

```
char str5[6] = {"HELLO"}; ← Valid
```

```
char str6[6] = "HELLO"; ← Valid
```

```
char str7[] = "HELLO"; ← Valid
```

```
char *str8 = "HELLO"; ← Valid
```

# Advanced C

## Strings - Memory Allocation



### Example

```
char str1[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```

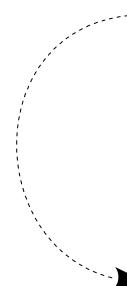
```
char *str2 = "Hello";
```

str1

‘H’	1000
‘E’	1001
‘L’	1002
‘L’	1003
‘O’	1004
‘\0’	1005

str2

1000	996
?	997
?	998
?	999
‘H’	1000
‘E’	1001
‘L’	1002
‘L’	1003
‘O’	1004
‘\0’	1005



# Advanced C

## Strings - Size



### 002\_example.c

```
#include <stdio.h>

int main()
{
    char char_array_1[5] = {'H', 'E', 'L', 'L', 'O'};
    char char_array_2[] = "Hello";

    sizeof(char_array_1);
    sizeof(char_array_2);

    return 0;
}
```

The size of the array is calculated so,

5, 6

### 003\_example.c

```
int main()
{
    char *str = "Hello";

    sizeof(str);

    return 0;
}
```

The size of pointer is always constant so,  
4 (32 Bit Sys)



# Advanced C

## Strings - Size



### 004\_example.c

```
#include <stdio.h>

int main()
{
    if (sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
    {
        printf("WoW\n");
    }
    else
    {
        printf("HuH\n");
    }

    return 0;
}
```



# Advanced C

## Strings - Manipulations



## 005\_example.c

```
#include <stdio.h>

int main()
{
    char str1[6] = "Hello";
    char str2[6];

    str2 = "World";

    char *str3 = "Hello";
    char *str4;

    str4 = "World";

    str1[0] = 'h';
    str3[0] = 'w';

    printf("%s\n", str1);
    printf("%s\n", str2);

    return 0;
}
```

**Not possible to assign a string to a  
array since its a constant pointer**

Possible to assign a string to a pointer since its variable

**Valid.** str1 contains “hello”

**Invalid. str3 might be stored in  
read only section.  
Undefined behaviour**



# Advanced C

## Strings - Sharing



### 006\_example.c

```
#include <stdio.h>

int main()
{
    char *str1 = "Hello";
    char *str2 = "Hello";

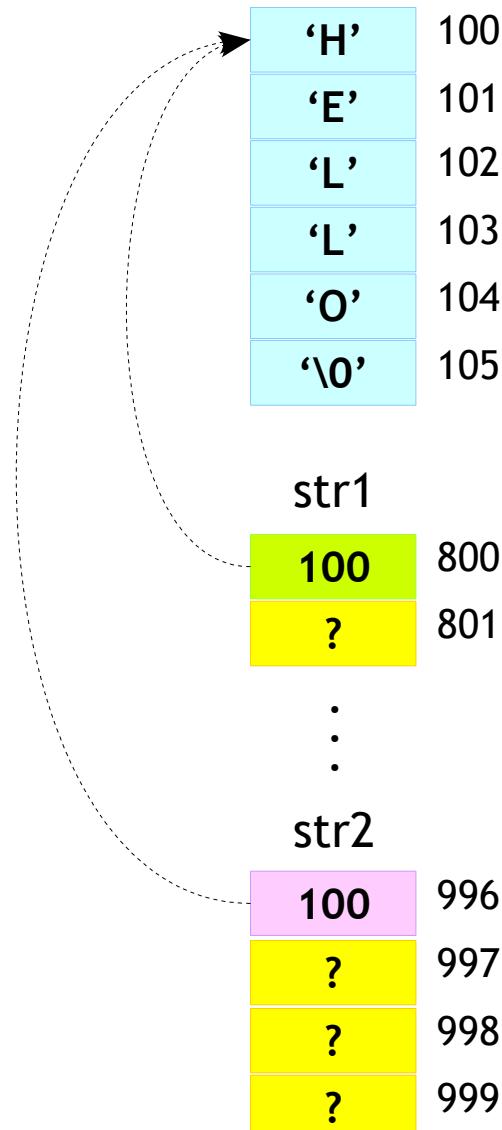
    if (str1 == str2)
    {
        printf("Hoo. They share same space\n");
    }
    else
    {
        printf("No. They are in different space\n");
    }

    return 0;
}
```



# Advanced C

## Strings - Sharing



# Advanced C

## Strings - Empty String

### 007\_example.c

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

int main()
{
    char *str = "";
    int ret;

    ret = strlen(str);
    printf("%d\n", ret);

    return 0;
}
```

# Advanced C

## Strings - Passing to Function

### 008\_example.c

```
#include <stdio.h>

void print(const char *str)
{
    while (*str)
    {
        putchar(*str++);
    }
}

int main()
{
    char *str = "Hello World";

    print(str);

    return 0;
}
```

# Advanced C

## Strings - Reading

### 009\_example.c

```
#include <stdio.h>

int main()
{
    char str[6];

    gets(str);
    printf("The string is: %s\n", str);

    return 0;
}
```

- The above method is not recommended by the gcc. Will issue warning while compilation
- Might lead to stack smashing if the input length is greater than array size!!

# Advanced C

## Strings - Reading



### 010\_example.c

```
#include <stdio.h>

int main()
{
    char str[6];

    fgets(str, 6, stdin);
    printf("The string is: %s\n", str);

    scanf("%5[^\\n]", str);
    printf("The string is: %s\n", str);

    return 0;
}
```

- fgets() function or selective scan with width are recommended to read string from the user



# Advanced C

## Strings - DIY



- WAP to calculate length of the string
- WAP to copy a string
- WAP to compare two strings
- WAP to compare two strings ignoring case
- WAP to check a given string is palindrome or not



# Advanced C

## Strings - Library Functions



Purpose	Prototype	Return Values
Length	<code>size_t strlen(const char *str)</code>	String Length
Compare	<code>int strcmp(const char *str1, const char *str2)</code>	<code>str1 &lt; str2 → &lt; 0</code> <code>str1 &gt; str2 → &gt; 0</code> <code>str1 = str2 → = 0</code>
Copy	<code>char *strcpy(char *dest, const char *src)</code>	Pointer to dest
Check String	<code>char *strstr(const char *haystack, const char *needle)</code>	Pointer to the beginning of substring
Check Character	<code>char *strchr(const char *s, int c)</code>	Pointer to the matched char else NULL
Merge	<code>char *strcat(char *dest, const char *src)</code>	Pointer to dest

# Advanced C

## Strings - Quiz



- Can we copy 2 strings like, str1 = str2?
- Why don't we pass the size of the string to string functions?
- What will happen if you overwrite the '\0' (null character) of string? Will you still call it a string?
- What is the difference between char \*s and char s[]?



# Advanced C

## Strings - DIY



- WAP to reverse a string
- WAP to compare string2 with string1 up to n characters
- WAP to concatenate two strings



# Advanced C

## Strings - DIY



- Use the standard string functions like
  - `strlen`
  - `strcpy`
  - `strcmp`
  - `strcat`
  - `strstr`
  - `strtok`



# Advanced C

## Strings - DIY

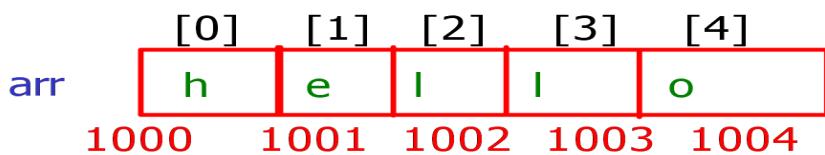


- WAP to print user information -
  - Read : Name, Age, ID, Mobile number
  - Print the information on monitor
  - Print error “Invalid Mobile Number” if length of mobile number is not 10
- WAP to read user name and password and compare with stored fields. Present a puzzle to fill in the banks
- Use strtok to separate words from string  
“www.emertxe.com/bangalore”



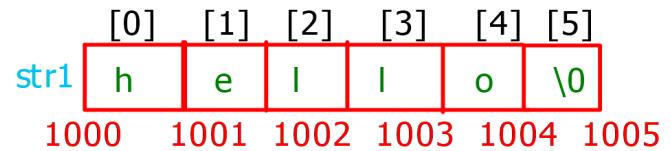
# Strings

- String is a sequence of characters terminated by null character '\0' or array of characters terminated by null character '\0'.
- Strings are used to store the text/message information in the application like name, address, company name, institute name etc.
- Normally array of character is declared as,
  - char arr[5] = {'h', 'e', 'l', 'l', 'o'};
  - Memory will be allocated 5 bytes in stack like below



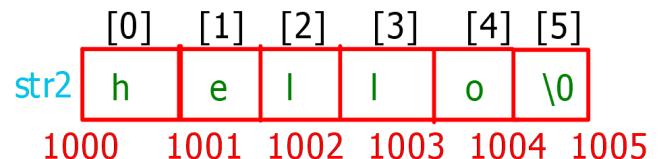
- But above array is not string, it has to be end by null character like below example
- Below are the ways of declaring the string in C language

1. `char str1[6] = {'h', 'e', 'l', 'l', 'o', '\0'};`



2. `char str2[] = {"h", "e", "l", "l", "o", "\0"};`

- As, you learned in the arrays, size can be skipped if array is initialised
- But see to it that array is terminated by '\0'

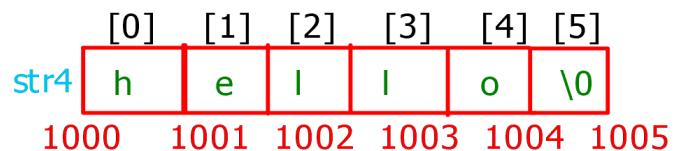
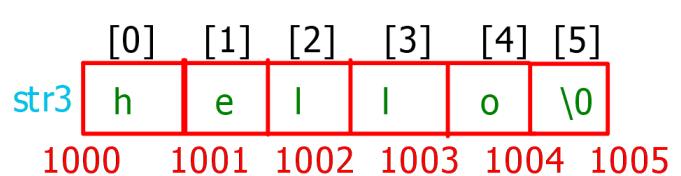


3. `char str3[] = {"h" "e" "l" "l" "o" "\0"};`

- Here, each character is treated as string and concatenate with the each like,

$$\begin{aligned} h\backslash 0 + e\backslash 0 + l\backslash 0 + l\backslash 0 + o\backslash 0 + \backslash 0 &= \\ \text{hello}\backslash 0 + \backslash 0 &= \text{hello}\backslash 0 \end{aligned}$$

- So, whenever “ “ is used, the compiler will add a null character at the end.



4. `char str5[6] = {"Hello"};`

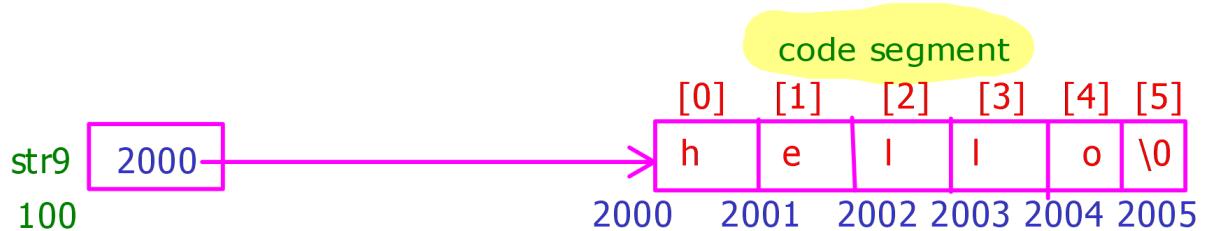
- a. Here, because of double quotes, the compiler will add null character at the end explicitly.

5. Below are the other ways of creating strings.

```
char str6[] = {"Hello"};
char str7[6] = "Hello";
char str8[] = "Hello";
```

6. `char *str9 = "hello";`

- Pointer to an characters
- These kind of initialisation is called as string literals
- Memory for the string literals are allocated in code/text segment

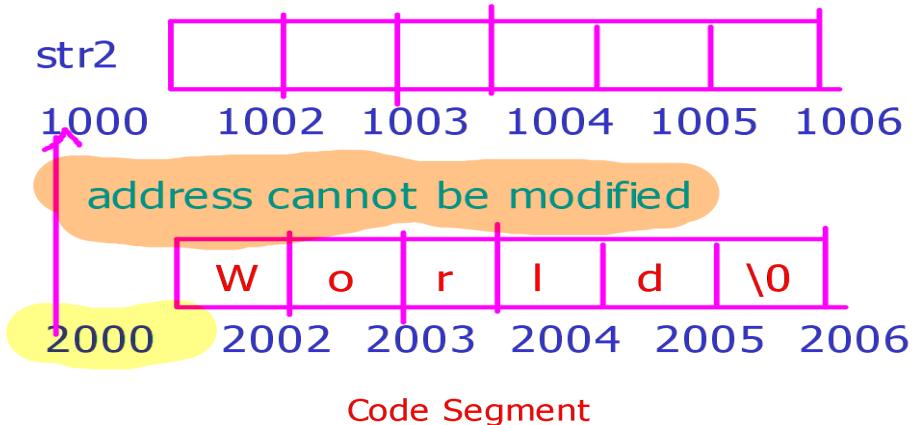


- Code segment is read only memory
- So, if you try to modify the string literals which will result in segmentation fault error
- `Str[2] = 'E'` is not allowed

Invalid ways of creating strings

1. `char str3[] = {"h", "e", "l", "l", "o", "\0"};`
  - invalid creation/initialization of a string - error
  - This will be treated as an array of strings, to create an array of string 2D array is used, which will be learned later in the module.
2. `char str[ ];`
  - Invalid, because the size of the array is mandatory.
3. `char str2[6];`  
`Str2 = "World";`
  - It's a compile time error because array address will be fixed
  - This declaration is creating string in code segment and returning the address of that string, Array address cannot be modified as shown below

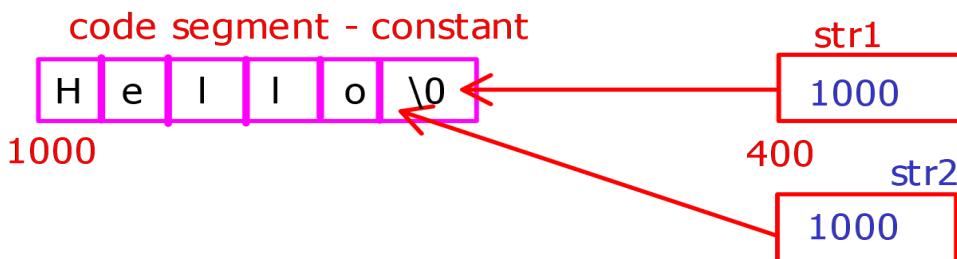
- So, it will result in a compile time error.



- But with a pointer it's possible because a pointer is capable of holding the address of any memory segments.
- So, the below declaration holds true for pointers.
  - `char *str;`  
`Str = "World";`

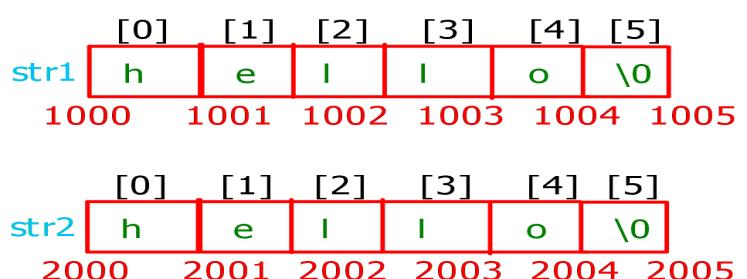
### Shareable Strings

- Consider below example
  - `char str1 = "Hello";`  
`char str2 = "Hello";`



- If the string literals are having same collection of characters (case sensitive), then because of read only memory instead of allocating new memory compiler will make pointer to point to the same memory location
- This is known as shareable memory.
- Sharing is only valid in case of string literals and pointers. With array the memory layout looks like below:

`str1[6] = "hello";`  
`str2[6] = "hello";`



## String Methods

### 1. strlen(str)

- a. Strlen function returns the length of the string
- b. Difference between strlen and sizeof is that sizeof returns the number of bytes including the null character but strlen returns length excluding null character.
- c. Strlen returns an unsigned integer value.

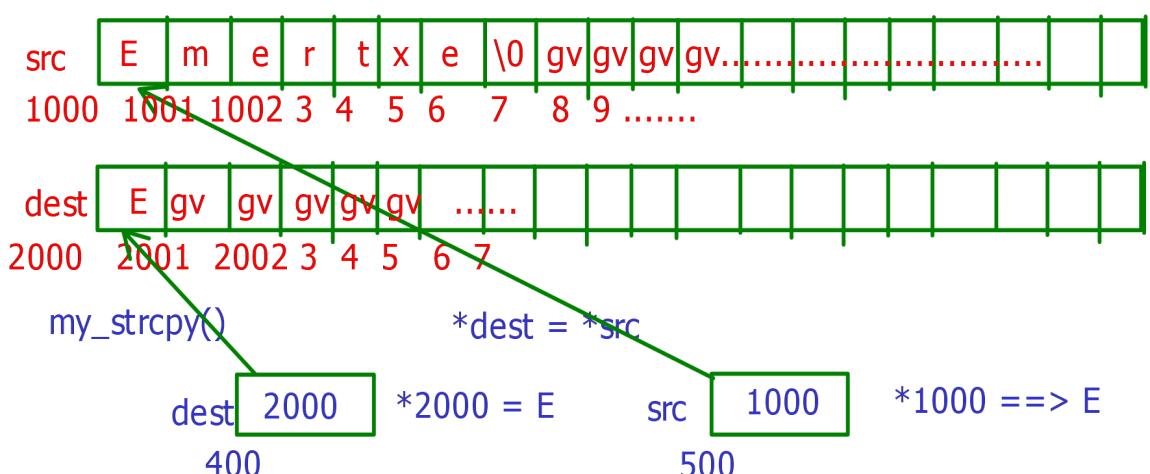
```
size_t my_strlen(char *str)           1.. *(1000+0++) - *1000 - h count=1
{
    int count =                         2.. *(1000+1++) - *1001 - i count=2
    while(*(str+count++));             => 3.. *(1000+2++) - *1002 - \0 count = 3
    return count-1;
}
```

### 2. String copy - char \*strcpy(char \*dest, const char \*src);

- a. strcpy is used to copy a string from one to another.
- b. strcpy will copy character by character like below

```
void my_strcpy(char *dest, char *src)
{
    while(*src)
    {
        *dest = *src;
        src++;
        dest++;
    }
    *dest = *src; //to copy null character at the end
}
```

### main()



3. String Compare - **int strcmp(const char \*s1, const char \*s2);**

- a. Strcmp is used to compare the 2 strings.
- b. Strcmp will return 0 → if both the strings are equal
  - i. Returns < 0 if first string has lesser ASCII value than second string
  - ii. Returns > 0 if first string has greater ASCII than second string

For example

Case 1: string1 = "Ram"  
string2 = "Ram"

'R' is compared with 'R' using ASCII equivalent -> 'R' - 'R'  
-> 82 - 82 = 0

'a' is compared with 'a' using ASCII equivalent -> 'a' - 'a'  
-> 97 - 97 = 0

'm' is compared with 'a' using ASCII equivalent -> 'm' - 'm'  
-> 109 - 109 = 0

It's returning 0 at the end so, strings are equal.

Case 2:

string 1 = "Ram"  
string 2 = "Rama"

'R' is compared with 'R' using ASCII equivalent -> 'R' - 'R'  
-> 82 - 82 = 0

'a' is compared with 'a' using ASCII equivalent -> 'a' - 'a'  
-> 97 - 97 = 0

'm' is compared with 'a' using ASCII equivalent -> 'm' - 'm'  
-> 109 - 109 = 0

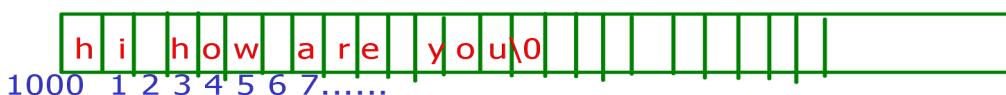
'\0' is compared with 'a' using ASCII equivalent -> '\0' - 'a'  
-> 0 - 97 = - 97

Here, strcmp return -97, so strings are not equal

- strcmp will compare character by character using ASCII equivalent value
- Once the character is mismatched, it will stop comparing
- If any one of the string reached to null character stop comparing

4. Substring search - **char \*strstr(const char \*haystack, const char \*needle);**

haystack



needle



- a. Strstr will search for the substring in the given string
  - b. Here, needle is the substring and haystack is the string.
  - c. once there is a matching, it will return the starting address of how which is 1003 in the above example.
  - d. returns the base address of matching character
  - e. if matching needle is not found then it will return the NULL address
5. String concatenation - **char \*strcat(char \*restrict dest, const char \*restrict src);**
- a. Strcat will merge one string with another
  - b. It will concat the destination string with source
6. String token - **char \*strtok(char \*restrict str, const char \*restrict delim);**
- a. Strtok will search for the token present in the string
  - b. Once the token is found, replace that token with '\0'.
  - c. For example,  
 char str = "hi;:how are\you?/:bangalore"  
 char token = ";"
  - d. In the above example strtok will search for the ";" in the string whenever it finds the token it stops searching and returns the starting address from where it started searching for.
  - e. To continue the search use loop and next time pass the Null as a first argument.
- Ex,
- ```

Res = strtok(str, token);
while(*str)
{
    printf("%s\n", res);
    strtok(NULL, token);
}

```
7. ASCII to integer - **int atoi(const char \*nptr);**
- a. Atoi function converts given string to integer
  - b. Takes input as string and produce integer as output
  - c. For example - input : "123"
   
Output: 123

Input: "123abc"  
 Output: 123

Input: abc123  
 Output: 0

C

Come let's see how deep it is!!  
- Storage Classes

Team Emertxe



# Advanced C

## Memory Segments

### Linux OS



The Linux OS is divided into two major sections

- User Space
- Kernel Space

The user programs cannot access the kernel space. If done will lead to segmentation violation

Let us concentrate on the user space section here

# Advanced C

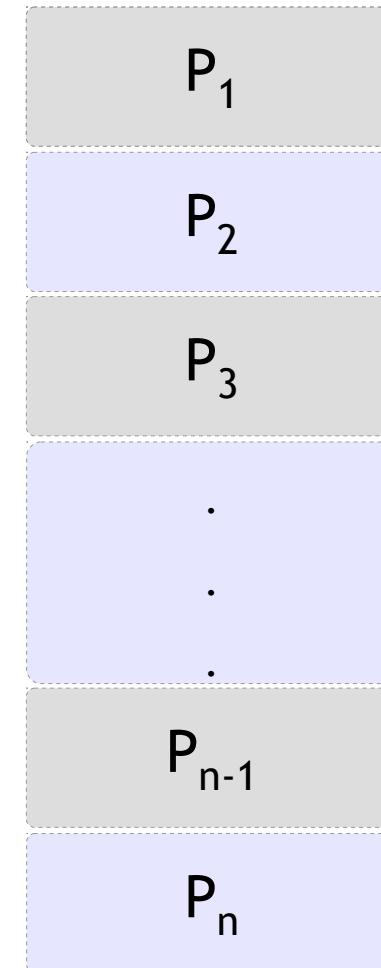
## Memory Segments



Linux OS



User Space



The User space contains many processes

Every process will be scheduled by the kernel

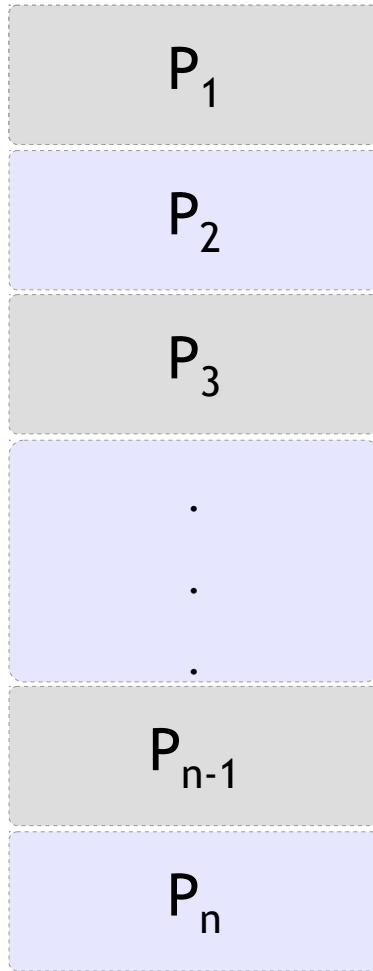
Each process will have its memory layout discussed in next slide

# Advanced C

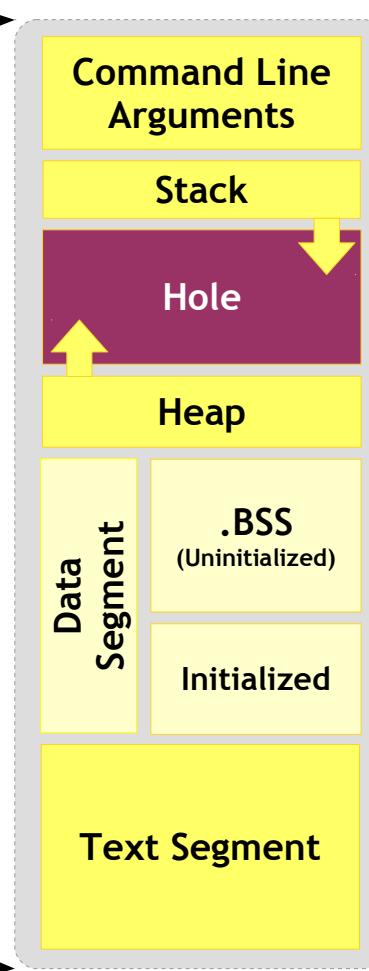
## Memory Segments



### User Space



### Memory Segments



The memory segment of a program contains four major areas.

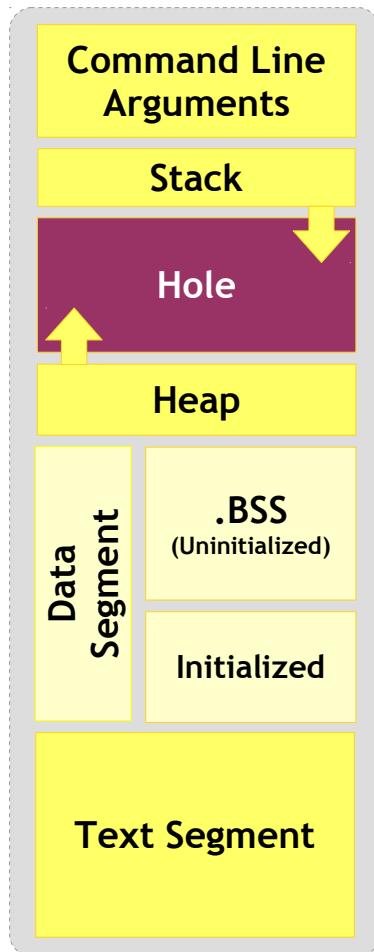
- Text Segment
- Stack
- Data Segment
- Heap

# Advanced C

## Memory Segments - Text Segment



### Memory Segments



Also referred as Code Segment

Holds one of the section of program in object file or memory

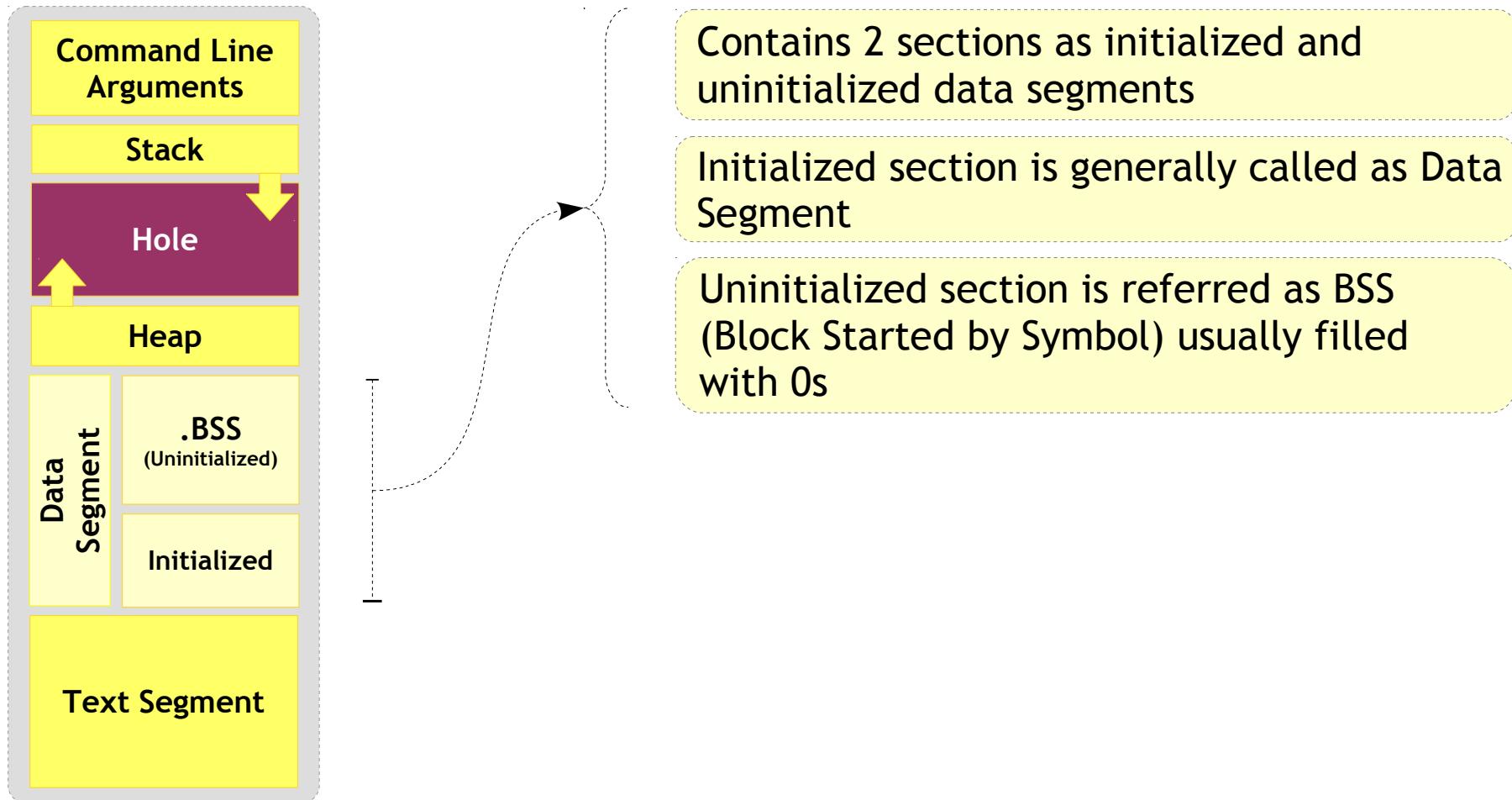
In memory, this is place below the heap or stack to prevent getting over written

Is a read only section and size is fixed

# Advanced C

## Memory Segments - Data Segment

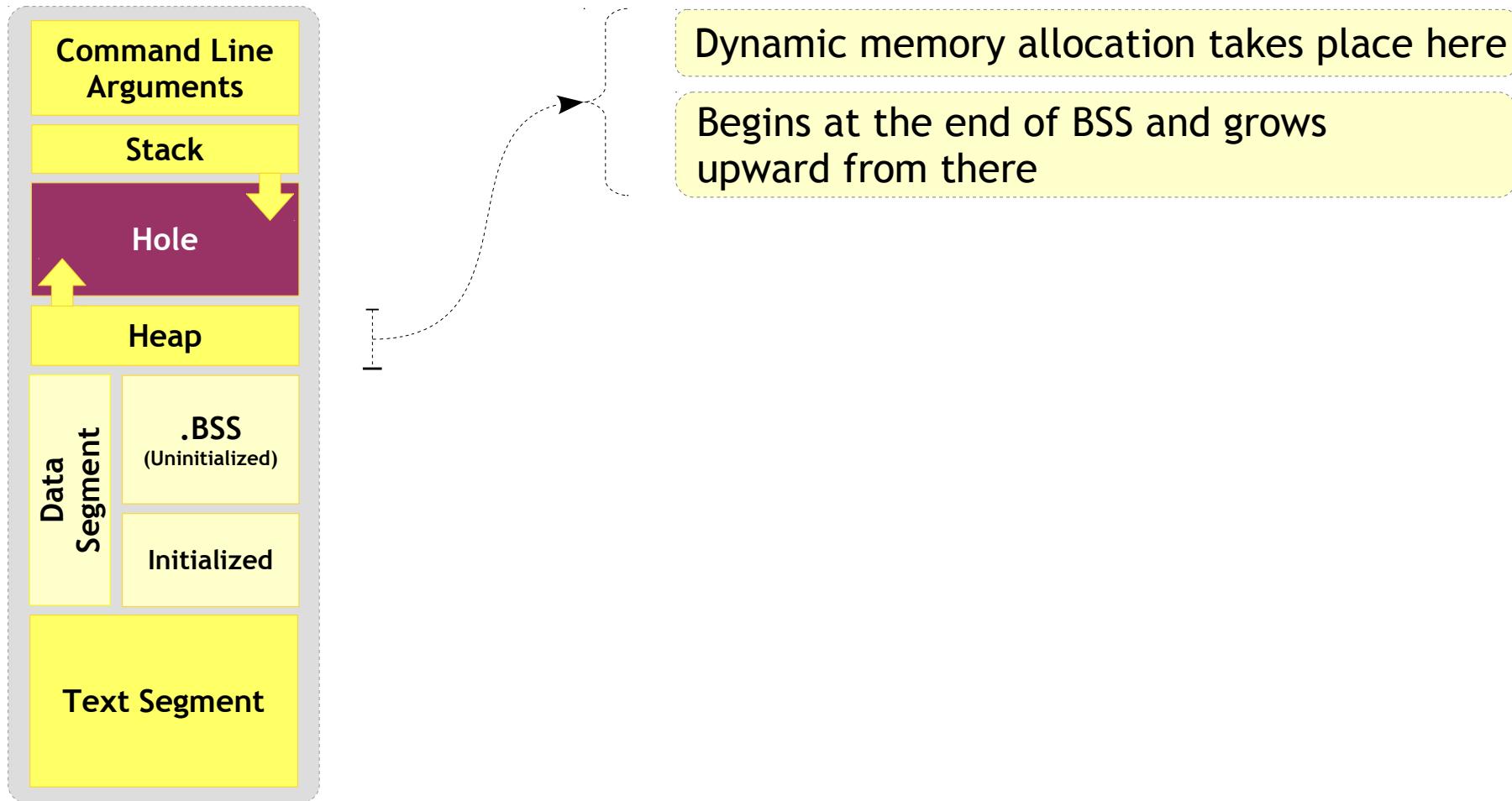
### Memory Segments



# Advanced C

## Memory Segments - Data Segment

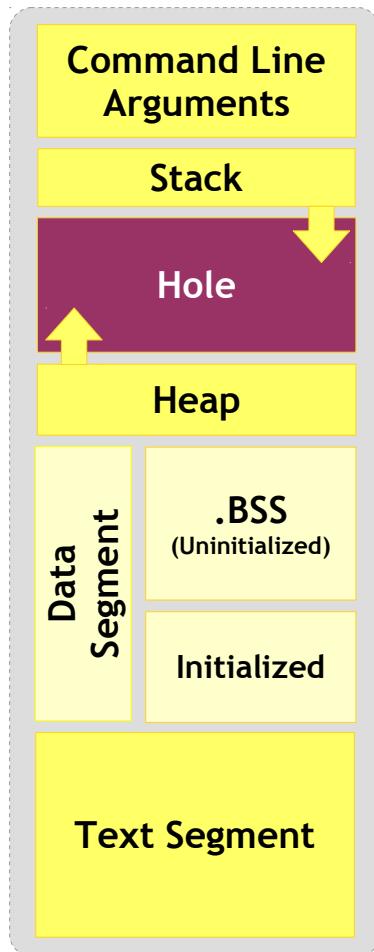
### Memory Segments



# Advanced C

## Memory Segments - Stack Segment

### Memory Segments



Adjoins the heap area and grow in opposite area of heap when stack and heap pointer meet (Memory Exhausted)

Typically loaded at the higher part of memory

A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack

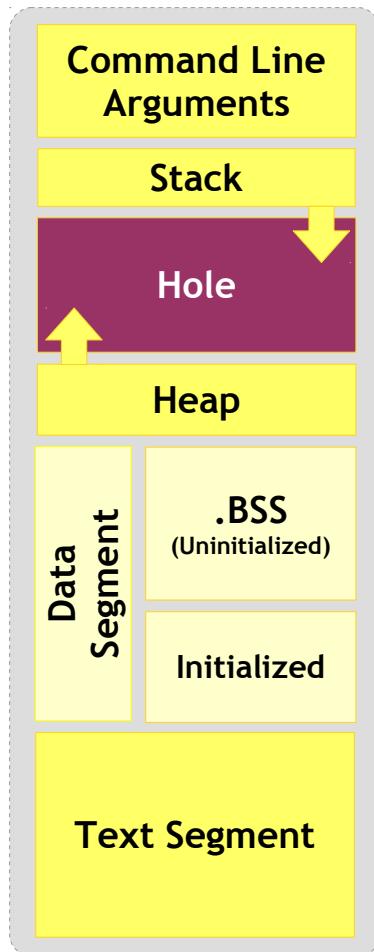
The set of values pushed for one function call is termed a “stack frame”

# Advanced C

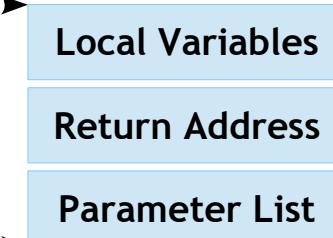
## Memory Segments - Stack Segment



### Memory Segments



### Stack Frame



A stack frame contain at least of a return address

# Advanced C

## Memory Segments - Stack Frame

```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```
int add_numbers(int n1, int n2)
{
    int s = 0;

    s = n1 + n2;

    return s;
}
```

### Stack Frame

num1 = 10  
num2 = 20  
sum = 0

Return Address to the caller

s = 0

Return Address to the main()

n1 = 10  
n2 = 20

main()

add\_numbers()

# Advanced C

## Memory Segments - Runtime



- **Text Segment:** The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions
- **Initialized Data Segment:** This segment contains global variables which are initialized by the programmer
- **Uninitialized Data Segment:** Also named “BSS” (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL (for pointers) before the program begins to execute

# Advanced C

## Memory Segments - Runtime



- **The Stack:** The stack is a collection of stack frames. When a new frame needs to be added (as a result of a newly called function), the stack grows downward
- **The Heap:** Most dynamic memory, whether requested via C's `malloc()`. The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows upward



# Advanced C

## Storage Classes



| Variable           | Storage Class | Scope    | Lifetime                 | Memory Allocation | Linkage             |
|--------------------|---------------|----------|--------------------------|-------------------|---------------------|
| Local              | auto          | Block    | B/W block entry and exit | Stack             | None                |
|                    | register      | Block    | B/W block entry and exit | Register / Stack  | None                |
|                    | static        | Block    | B/W program start & end  | Data Segment      | None                |
| Global             | static        | File     | B/W program start & end  | Data segment      | Internal            |
|                    | extern        | Program  | B/W program start & end  | Data segment      | Internal / External |
| Function Parameter | register      | Function | Function entry and exit  | Stack             | None                |

\*Block : function body or smaller block with in function

# Advanced C

## Declaration

```
extern int num1; ←  
extern int num1; ←  
  
int main(); ←  
  
int main()  
{  
    int num1, num2;  
    char short_opt;  
    ↑  
    ...  
}
```

Declaration specifies type to the variables

Its like an announcement and hence can be made 1 or more times

Declaration about num1

Declaration about num1 yet again!!

Declaration about main function



# Advanced C

## Declaration

```
extern int num1; ←  
extern int num1; ←  
  
int main(); ←  
  
int main()  
{  
    int num1, num2;  
    char short_opt;  
    ←  
    ...  
}
```

Declaration specifies type to the variables

Its like an announcement and hence can be made 1 or more times

Declaration about num1

Declaration about num1 yet again!!

Declaration about main function



\*\*\* One Definition Rule for variables \*\*\*

“In a given scope, there can be only one definition of a variable”

# Advanced C

## Storage Classes - Auto

### 001\_example.c

```
#include <stdio.h>

int main()
{
    int i = 0;

    printf("i %d\n", i);

    return 0;
}
```

# Advanced C

## Storage Classes - Auto



### 002\_example.c

```
#include <stdio.h>

int foo()
{
    int i = 0;

    printf("i %d\n", i);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```



# Advanced C

## Storage Classes - Auto



### 003\_example.c

```
#include <stdio.h>

int *foo()
{
    int i = 10;
    int *j = &i;

    return j;
}

int main()
{
    int *i;

    i = foo();
    printf("%d\n", *i);

    return 0;
}
```



# Advanced C

## Storage Classes - Auto

### 004\_example.c

```
#include <stdio.h>

char *foo()
{
    char ca[12] = "Hello World";

    return ca;
}

int main()
{
    char *ca;

    ca = foo();
    printf("ca is %s\n", ca);

    return 0;
}
```

# Advanced C

## Storage Classes - Auto

### 005\_example.c

```
#include <stdio.h>

int book_ticket()
{
    int ticket_sold = 0;

    ticket_sold++;

    return ticket_sold;
}

int main()
{
    int count;

    count = book_ticket();
    count = book_ticket();

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

    return 0;
}
```

# Advanced C

## Storage Classes - Auto

### 006\_example.c

```
#include <stdio.h>

int main()
{
    int i = 0;

    {
        int j = 0;

        printf("i %d\n", i);
    }

    printf("j %d\n", j);

    return 0;
}
```

# Advanced C

## Storage Classes - Auto

### 007\_example.c

```
#include <stdio.h>

int main()
{
    int j = 10;

    {
        int j = 0;

        printf("j %d\n", j);
    }

    printf("j %d\n", j);

    return 0;
}
```

# Advanced C

## Storage Classes - Auto

### 008\_example.c

```
#include <stdio.h>

int main()
{
    int i = 10;
    int i = 20;

    {
        printf("i %d\n", i);
    }

    printf("i %d\n", i);

    return 0;
}
```

# Advanced C

## Storage Classes - Register

### 009\_example.c

```
#include <stdio.h>

int main()
{
    register int i = 0;

    scanf("%d", &i);
    printf("i %d\n", i);

    return 0;
}
```

# Advanced C

## Storage Classes - Register

### 010\_example.c

```
#include <stdio.h>

int main()
{
    register int i = 10;
    register int *j = &i;

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

    return 0;
}
```

# Advanced C

## Storage Classes - Register

### 011\_example.c

```
#include <stdio.h>

int main()
{
    int i = 10;
    register int *j = &i;

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

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local

### 012\_example.c

```
#include <stdio.h>

int *foo()
{
    static int i = 10;
    int *j = &i;

    return j;
}

int main()
{
    int *i;

    i = foo();
    printf("i %d\n", *i);

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local



### 013\_example.c

```
#include <stdio.h>

char *foo()
{
    static char ca[12] = "Hello World";

    return ca;
}

int main()
{
    char *ca;

    ca = foo();
    printf("ca is %s\n", ca);

    return 0;
}
```



# Advanced C

## Storage Classes - Static Local

### 014\_example.c

```
#include <stdio.h>

int book_ticket()
{
    static int ticket_sold = 0;

    ticket_sold++;

    return ticket_sold;
}

int main()
{
    int count;

    count = book_ticket();
    count = book_ticket();

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

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local

015\_example.c

```
#include <stdio.h>

int main()
{
    static int i = 5;

    if (--i)
    {
        main();
    }

    printf("i %d\n", i);

    return 0;
}
```

016\_example.c

```
#include <stdio.h>

int main()
{
    static int i = 5;

    if (--i)
    {
        return main();
    }

    printf("i %d\n", i);

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local

### 017\_example.c

```
#include <stdio.h>

int foo()
{
    static int i;

    return i;
}

int main()
{
    static int x = foo();

    printf("x %d\n", x);

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local

### 018\_example.c

```
#include <stdio.h>

int *foo()
{
    static int i = 10;
    int *j = &i;

    return j;
}

int main()
{
    int *i;

    i = foo();
    printf("i %d\n", *i);

    return 0;
}
```

# Advanced C

## Storage Classes - Static Local

### 019\_example.c

```
#include <stdio.h>

int *foo()
{
    int i = 10;
    static int *j = &i;

    return j;
}

int main()
{
    int *i;

    i = foo();
    printf("i %d\n", *i);

    return 0;
}
```

# Advanced C

## Storage Classes - Global



### 020\_example.c

```
#include <stdio.h>

int x;

int foo()
{
    printf("x %d\n", x);

    return ++x;
}

int main()
{
    foo();

    printf("x %d\n", x);

    return 0;
}
```



# Advanced C

## Storage Classes - Global

### 021\_example.c

```
#include <stdio.h>

auto int x;

int foo()
{
    printf("x %d\n", x);

    return ++x;
}

int main()
{
    foo();

    printf("x %d\n", x);

    return 0;
}
```

# Advanced C

## Storage Classes - Global

### 022\_example.c

```
#include <stdio.h>

register int x;

int foo()
{
    printf("x %d\n", x);

    return ++x;
}

int main()
{
    foo();

    printf("x %d\n", x);

    return 0;
}
```



# Advanced C

## Storage Classes - Global

### 023\_example.c

```
#include <stdio.h>

int x = 10;

int foo()
{
    printf("x %d\n", x);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```



# Advanced C

## Storage Classes - Global



### 024\_example.c

```
#include <stdio.h>

int x = 10;
int x;

int foo()
{
    printf("x %d\n", x);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```

- Will there be any compilation error?

# Advanced C

## Storage Classes - Global



### 025\_example.c

```
#include <stdio.h>

int x = 10;
int x = 20;

int foo()
{
    printf("x %d\n", x);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```

- Will there be any compilation error?

# Advanced C

## Storage Classes - Global



### Example

```
#include <stdio.h>

1 int x = 10;           // Definition
2 int x;                // Tentative definition
3 extern int x;          // not a definition either
4 extern int x = 20;    // Definition
5 static int x;          // Tentative definition
```

- Declaration Vs definition
  - All the above are declarations
  - Definitions as mentioned in the comment

# Advanced C

## Storage Classes - Global



- Complying with one definition rule
  - All the tentative definitions and extern declarations are eliminated and mapped to definition by linking method
  - If there exists only tentative definitions then all of them are eliminated except one tentative definition which is changed to definition by assigning zero to the variable

- Compilation error if there are more than one definitions
- Compilation error if no definition or tentative definition exists

# Advanced C

## Storage Classes - Global



- **Translation unit** - A source file + header file(s) forms a translation unit
- Compiler translates source code file written in ‘C’ language to machine language
- A program is constituted by the set of translation units and libraries
- An identifier may be declared in a scope but used in different scopes (within a translation unit or in other translation units or libraries)
- **Linkage** - An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called linkage
- There are three type of linkages - **internal, external and none**



# Advanced C

## Storage Classes - Global



- **External Linkage** - A global variable declared without storage class has “external” linkage
- **Internal Linkage** - A global variable declared with static storage class has “internal” linkage
- **None Linkage** - Local variables have “none” linkage
- Variable declaration with “extern” - in such case, linkage to be determined by referring to **“previous visible declaration”**
  - If there is no “previous visible declaration” then external linkage
  - If “previous visible declaration” is local variable then external linkage
  - If “previous visible declaration” is global variable then linkage is same as of global variable

“Compilation error if there is linkage disagreement among definition and tentative definitions”



# Advanced C

## Storage Classes - Static Global

### 026\_example.c

```
#include <stdio.h>

static int x = 10;

int foo()
{
    printf("x %d\n", x);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```

# Advanced C

## Storage Classes - Static Global

### 027\_example.c

```
#include <stdio.h>

static int x = 10;
int x;

int foo()
{
    printf("x %d\n", x);

    return 0;
}

int main()
{
    foo();

    return 0;
}
```

# Advanced C

## Storage Classes - External

### 028\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
        func_2();
        sleep(1);
    }

    return 0;
}
```

### 029\_file2.c

```
#include <stdio.h>

extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

### 030\_file3.c

```
#include <stdio.h>

extern int num;

int func_2()
{
    printf("num is %d from file3\n", num);

    return 0;
}
```

# Advanced C

## Storage Classes - External

031\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

032\_file2.c

```
#include <stdio.h>

extern int num;
extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

# Advanced C

## Storage Classes - External

033\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

034\_file2.c

```
#include <stdio.h>

static int num;
extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

# Advanced C

## Storage Classes - External

035\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

036\_file2.c

```
#include <stdio.h>

extern char num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

# Advanced C

## Storage Classes - External

### 037\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

### 038\_file2.c

```
#include <stdio.h>

extern int num;
extern char num;
```

[ ] → Conflicting types

```
int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

Conflicting types

# Advanced C

## Storage Classes - External

### 039\_example.c

```
#include <stdio.h>

int main()
{
    int x;

    {
        int x = 10;
        {
            extern int x;
            printf("x %d\n", x);
        }
        printf("x %d\n", x);
    }
    printf("x %d\n", x);

    return 0;
}

int x = 20;
```

# Advanced C

## Storage Classes - External



### 040\_example.c

```
#include <stdio.h>

int main()
{
    extern char x;

    printf("x %c\n", x);

    return 0;
}

int x = 0x31;
```



# Advanced C

## Storage Classes - External

### 041\_example.c

```
#include <stdio.h>

int main()
{
    int x;

    {
        int x = 10;
        {
            extern int x = 20;
            printf("x %d\n", x);
        }
        printf("x %d\n", x);
    }
    printf("x %d\n", x);

    return 0;
}

int x;
```

Invalid, extern and initializer  
not permitted in block scope

# Advanced C

## Storage Classes - Static Function

042\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
    }

    return 0;
}
```

043\_file2.c

```
#include <stdio.h>

extern int num;

static int func_2()
{
    printf("num is %d from file2\n", num);

    return 0;
}

int func_1()
{
    func_2();
}
```

# Advanced C

## Storage Classes - Static Function

044\_file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_2();
    }

    return 0;
}
```

045\_file2.c

```
#include <stdio.h>

extern int num;

static int func_2()
{
    printf("num is %d from file2\n", num);

    return 0;
}

int func_1()
{
    func_2();
}
```

## Extra Examples



# Advanced C

## Storage Classes - External

file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

file2.c

```
#include <stdio.h>

extern int num;
static int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

- Compiler error due to linkage disagreement

# Advanced C

## Storage Classes - External

file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

file2.c

```
#include <stdio.h>

int num;
static int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

- Compiler error due to linkage disagreement

# Advanced C

## Storage Classes - External

file1.c

```
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
    }

    return 0;
}
```

file2.c

```
#include <stdio.h>

static int num;
int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

- Compiler error due to linkage disagreement

# Advanced C

## Storage Classes - External

### Example

```
#include <stdio.h>

int main()
{
    int x;

    {
        int x = 10;
        {
            extern int x;
            printf("x %d\n", x);
        }
        printf("x %d\n", x);
    }
    printf("x %d\n", x);

    return 0;
}

static int x = 20;
```

- Compiler error due to linkage disagreement

# Advanced C

## Storage Classes - External

### Example

```
#include <stdio.h>

static int x = 20;

int main()
{
    int x;

    {
        int x = 10;
        {
            extern int x;
            printf("x %d\n", x);
        }
        printf("x %d\n", x);
    }
    printf("x %d\n", x);

    return 0;
}
```

- Compiler error due to linkage disagreement

# Advanced C

## Storage Classes - External

### Example

```
#include <stdio.h>

int x = 20;

int main()
{
    int x;

    {
        int x = 10;
        {
            extern int x;
            printf("x %d\n", x);
        }
        printf("x %d\n", x);
    }
    printf("x %d\n", x);

    return 0;
}
```

- Should be a fine since the compiler refers the same variable **x** and prints 20

# Storage Classes

- Storage class is a predefined keyword which specifies where the memory is allocated for different variables.
- A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable.
  - Following are the different memory segment of a process:
    - Stack
    - Heap
    - Data Segment
      - BSS (Block started By Symbol)
      - Initialised
    - Text or Code segment
  - Following are the storage classes in C:
    - Auto
    - Register
    - Static local and global
    - Extern

## 1. Auto

- a. Auto variables are the one which is declared within the function. Memory for auto local variables will be in the stack segment. Scope of the auto is within the function.
- b. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.
- c. Scope of the auto variable is limited within the scope of block only.
- d. Once the block completes the execution, the auto variables are destroyed. This means only the block in which the auto variable is declared can access it.
- e. For example,

```
Int main()
{
    int num1;
    auto int num2;
}
```
- f. Auto variables cannot be declared in the global space.

## 2. Register

- a. Register variable is also a local variable where memory will be in the register.
- b. Register variables are used for faster access, because the register is very near to the processor.
- c. You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables.
- d. For example, “delay” or “counter” variables are a good example to be stored in the register.
- e. Scope of the variable is within the function.
- f. Cannot declare register in the global scope.
- g. For example,

```
int main()
{
    register int i;
    for(i = 0; i < 1000; i++); //delay loop
}
```
- h. Addresses of register variables are inaccessible but a register can hold the address of another variable through a pointer.

## 3. Static

- a. Static variables are declared with the keyword static and memory will be in the data segment as described below:
  - If the static variable is uninitialised then the memory will be in the BSS of the data segment and initialised with 0.
  - If the static is initialised then memory will be in the initialised block of the data segment.
- b. Scope of the static variables will be:
  - If declared within the function then scope is within function and life time will be until program execution
  - If declared outside the function then scope is within the file and lifetime is until the program execution.
- c. For example,

```
static int gnum; //global static
int main()
{
    static int lnum; //local static
}
```
- d. Static variables are best suited when the single instance of the variable is shared in the whole program, like the reservation system; ticket count is the best example for static.

- Extern is used to share the variables from one file to another. Memory will be in the data segment and scope is until program execution.
- Auto and register variables cannot be global, else it will result in compile time error.
- Addresses of the register variables are inaccessible. If you do then it will result in compile time error
- If static variables are global then the scope of the variable will be within the file and cannot be used with extern.
- Extern usage:

File1.c

```
int x = 10;
int main()
{
    foo();
    return 0;
}
```

File2.c

```
int foo()
{
    printf("x in foo %d\n",x);
    return 0;
}
```

- Execute the above program like:
  - gcc file1.c file2.c
  - There will be warning because of missing function prototype, so add function prototype in the file1.c
  - To use x it needs to be externed into that particular file, so in file2.c add extern int x in the global space.
  - Now run the program again. The output will be displayed as
    - X is in foo 10
- Extern always searches for previous visible declarations in the program.
- Last but not the least text or code segment is a read only segment in the memory. Whatever the code is stored in text segments like function code or string literals, those are read only data.

# Multilevel Pointers



# Advanced C

## Pointers - Multilevel



- A pointer, pointing to another pointer which can be pointing to others pointers and so on is known as multilevel pointers.
- We can have any level of pointers.
- As the depth of the level increases we have to be careful while dealing with it.

# Advanced C

## Pointers - Multilevel



### 001\_example.c

```
#include <stdio.h>

int main()
{
    →int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

| 1000 | num |
|------|-----|
| 10   |     |

# Advanced C

## Pointers - Multilevel



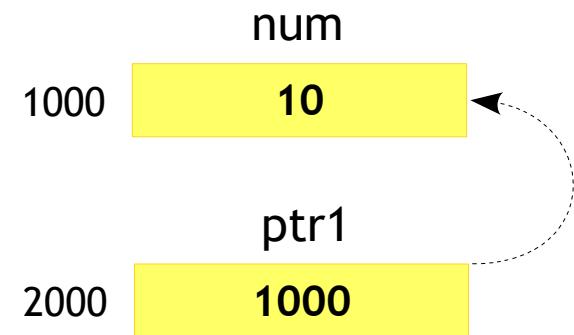
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    → int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



# Advanced C

## Pointers - Multilevel



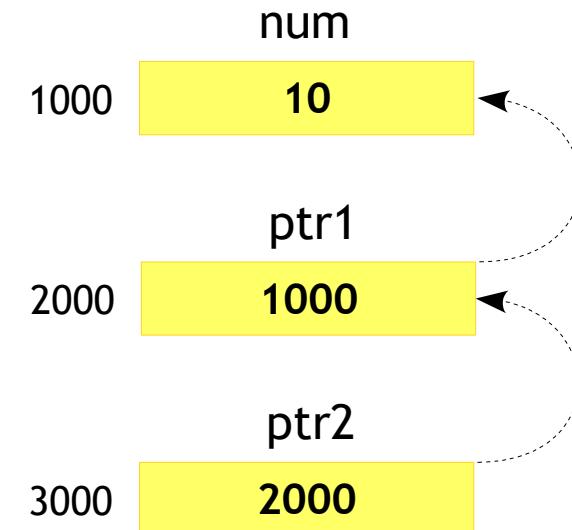
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
→  int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



# Advanced C

## Pointers - Multilevel



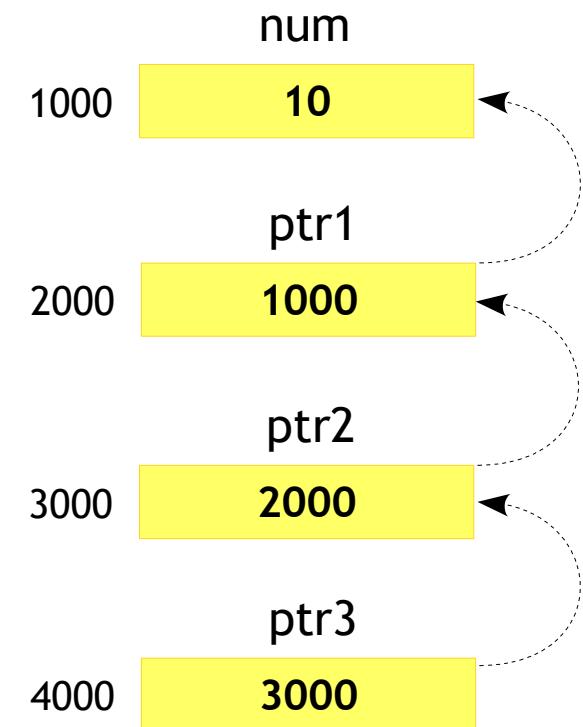
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    →int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



# Advanced C

## Pointers - Multilevel



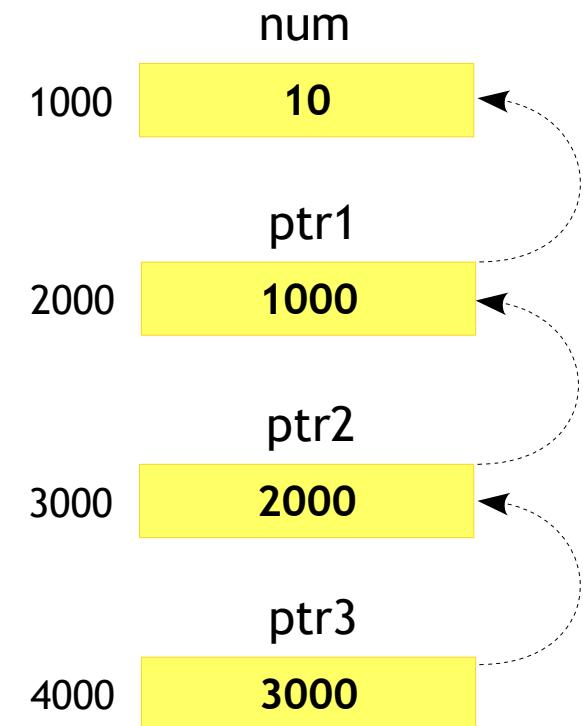
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

→ printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



Output → 3000

# Advanced C

## Pointers - Multilevel



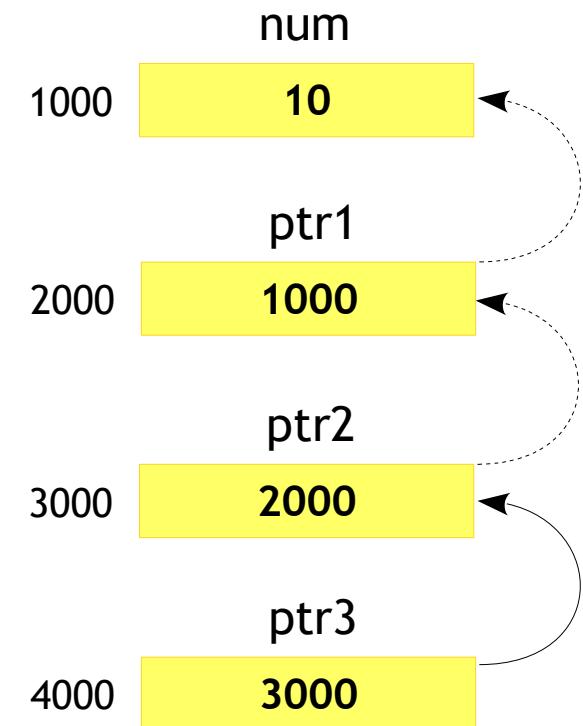
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    →printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



Output → 2000

# Advanced C

## Pointers - Multilevel



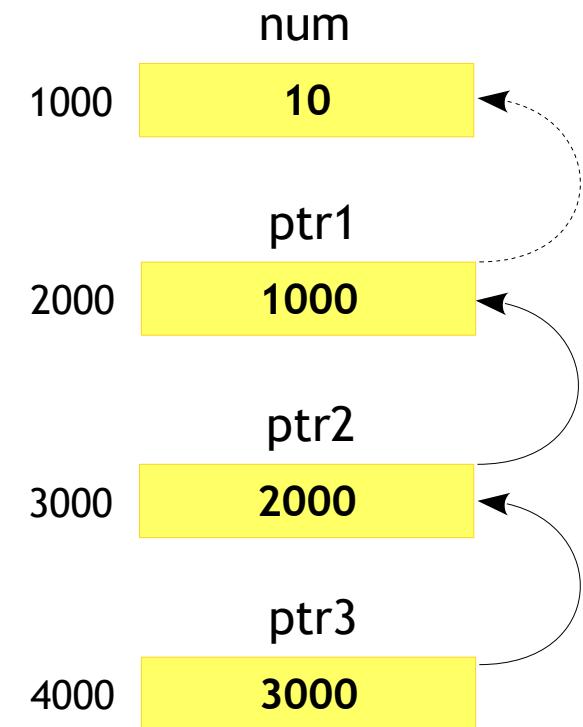
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    →printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



Output → 1000

# Advanced C

## Pointers - Multilevel



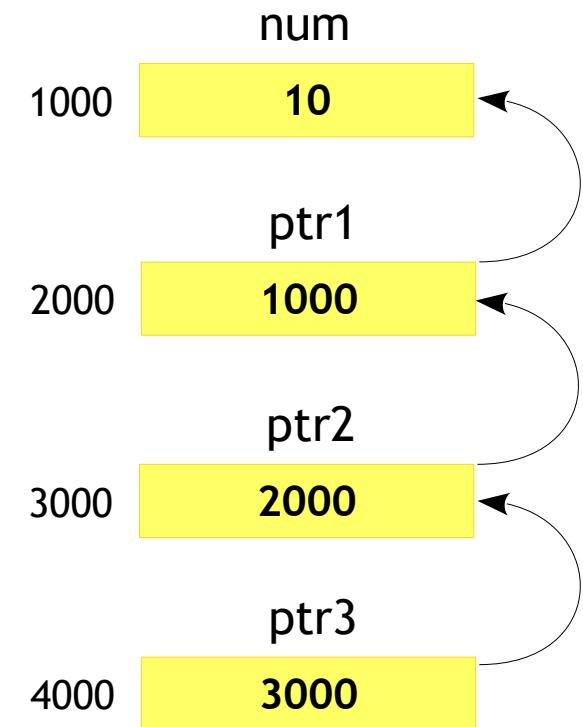
### 001\_example.c

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    →printf("%d", ***ptr3);

    return 0;
}
```



Output → 10

# Advanced C

## Arrays - Interpretations

### Example

```
#include <stdio.h>

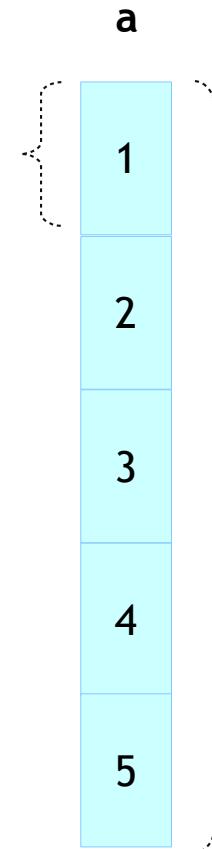
int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    return 0;
}
```

Pointer to  
the first  
small variable

While  
assigning to  
pointer

While  
passing to  
functions



One big  
variable

While  
using with  
sizeof()

While  
pointer  
arithmetic  
on &array

# Advanced C

## Arrays - Interpretations

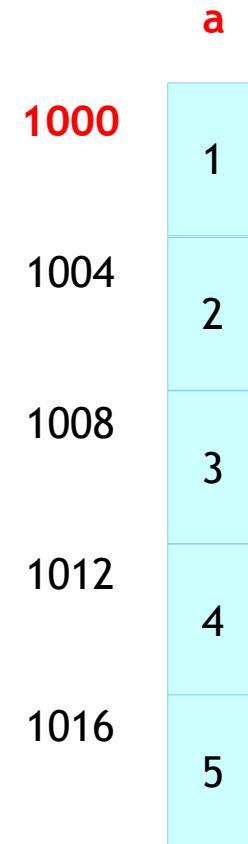
### 002\_example.c

```
#include <stdio.h>

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

→ printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations



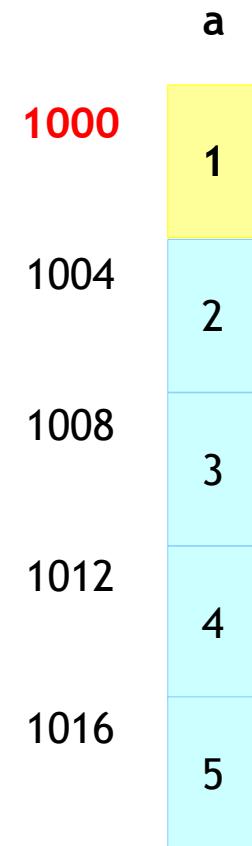
### 002\_example.c

```
#include <stdio.h>

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

    printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations

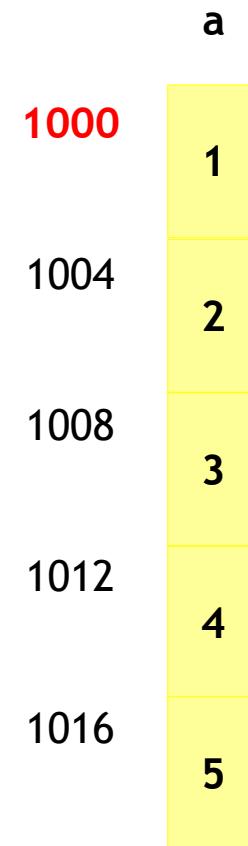
### 002\_example.c

```
#include <stdio.h>

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

    printf("%p\n", a);
    printf("%p\n", &a[0]);
→ printf("%p\n", &a);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations

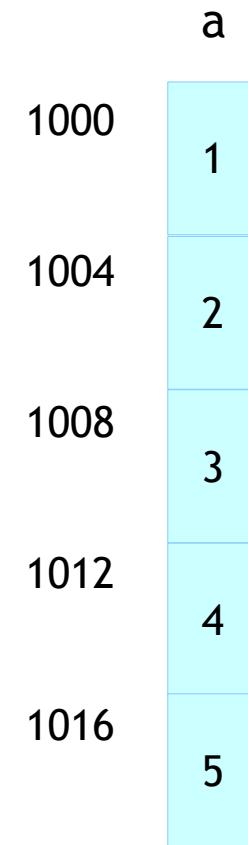
### 003\_example.c

```
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations

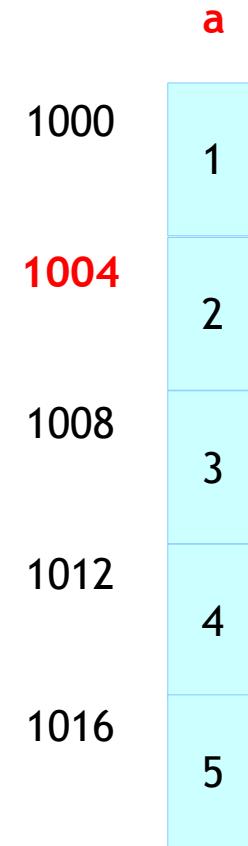
### 003\_example.c

```
#include <stdio.h>

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

→ printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations

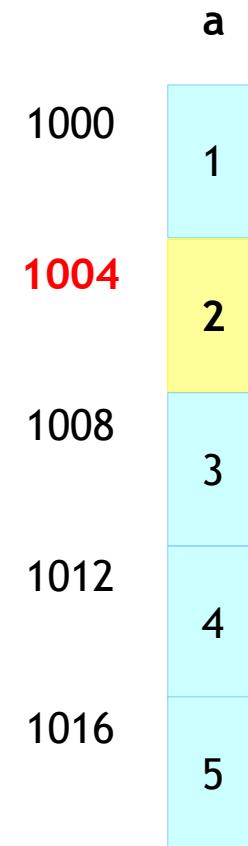
### 003\_example.c

```
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
    printf("%p\n", &a + 1);

    return 0;
}
```



# Advanced C

## Arrays - Interpretations

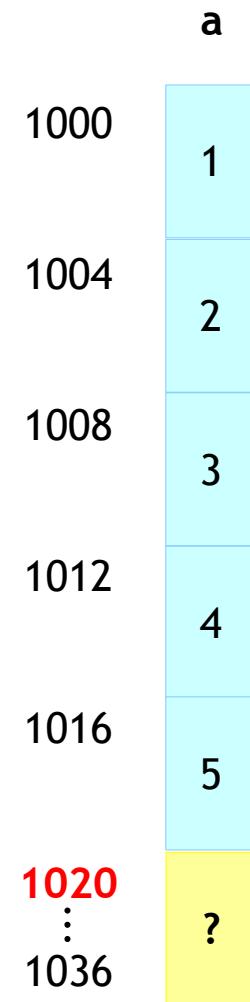
### 003\_example.c

```
#include <stdio.h>

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

    printf("%p\n", a + 1);
    printf("%p\n", &a[0] + 1);
→ printf("%p\n", &a + 1);

    return 0;
}
```



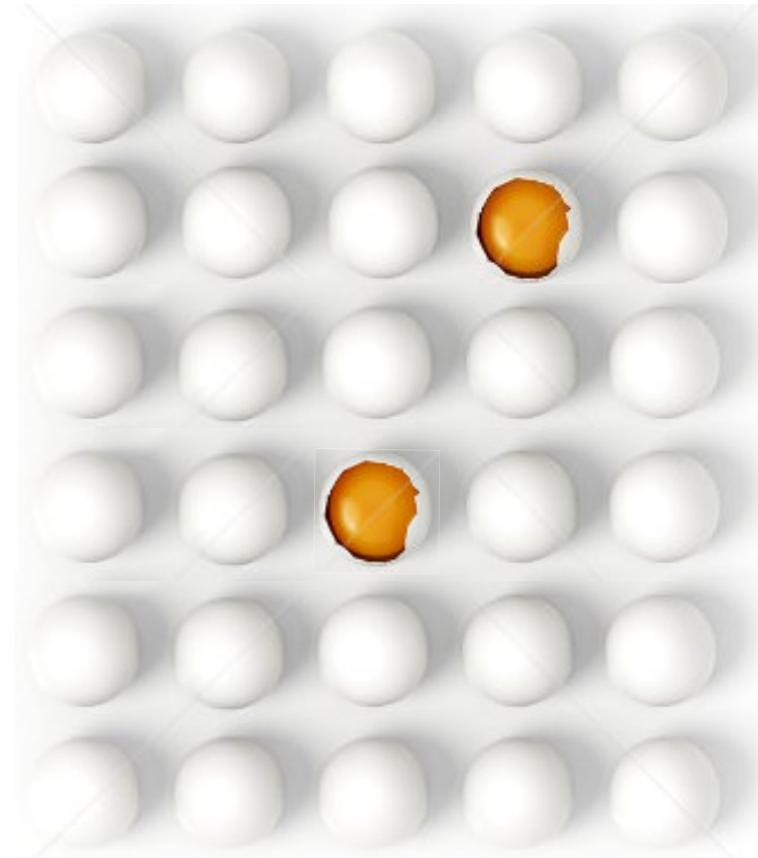
# Advanced C

## Arrays - Interpretations



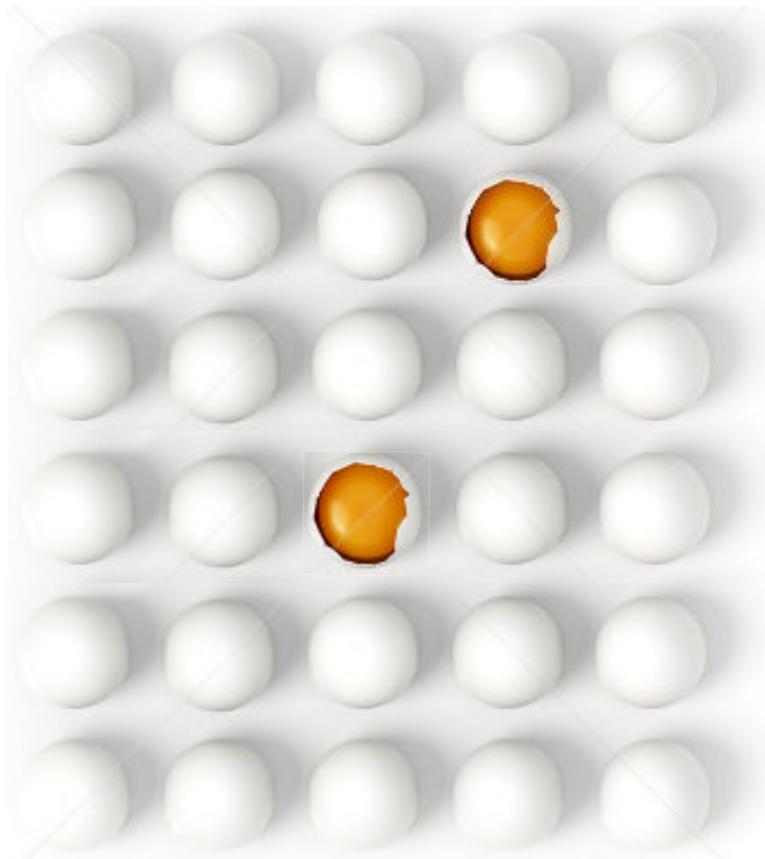
- So in summary, if we try to print the address of `a[]`
  - `a` - prints the value of the constant pointer
  - `&a[0]` - prints the address of the first element pointed by `a`
  - `&a` - prints the address of the whole array which pointed by `a`
- Hence all the lines will print `1000` as output
- These concepts plays a very important role in multi dimension arrays

# Advanced C Arrays



# Advanced C

## Arrays - 2D



- Find the broken eggs!



- Hmm, how should I proceed with count??

# Advanced C

## Arrays - 2D



|    | C1 | C2 | C3 | C4 | C5 |
|----|----|----|----|----|----|
| R1 |    |    |    |    |    |
| R2 |    |    |    | ●  |    |
| R3 |    |    |    |    |    |
| R4 |    | ●  |    |    |    |
| R5 |    |    |    |    |    |
| R6 |    |    |    |    |    |

- Now is it better to tell which one broken??

# Advanced C

## Arrays - 2D



|    | C1 | C2 | C3 | C4 | C5 |
|----|----|----|----|----|----|
| R1 |    |    |    |    |    |
| R2 |    |    |    |    |    |
| R3 |    |    |    |    |    |
| R4 |    |    |    |    |    |
| R5 |    |    |    |    |    |
| R6 |    |    |    |    |    |

- So in matrix method it becomes bit easy to locate items
- In other terms we can reference the location with easy indexing
- In this case we can say the broken eggs are at R2-C4 and R4-C3  
or  
C4-R2 and C3-R4

# Advanced C

## Arrays - 2D



- The matrix in computer memory is a bit tricky!!
- Why?. Since its a sequence of memory
- So pragmatically, it is a concept of dimensions is generally referred
- The next slide illustrates the expectation and the reality of the memory layout of the data in a system



# Advanced C

## Arrays - 2D



Concept Illustration

|    | C0  | C1  | C2  | C3  |
|----|-----|-----|-----|-----|
| R0 | 123 | 9   | 234 | 39  |
| R1 | 23  | 155 | 33  | 2   |
| R2 | 100 | 88  | 8   | 111 |
| R3 | 201 | 101 | 187 | 22  |

System Memory

|      |     |
|------|-----|
| 1001 | 123 |
| 1002 | 9   |
| 1003 | 234 |
| 1004 | 39  |
| 1005 | 23  |
| 1006 | 155 |
| 1007 | 33  |
| 1008 | 2   |
| 1009 | 100 |
| 1010 | 88  |
| 1011 | 8   |
| 1012 | 111 |
| 1013 | 201 |
| 1014 | 101 |
| 1015 | 187 |
| 1016 | 22  |



# Advanced C

## Arrays - 2D



### Syntax

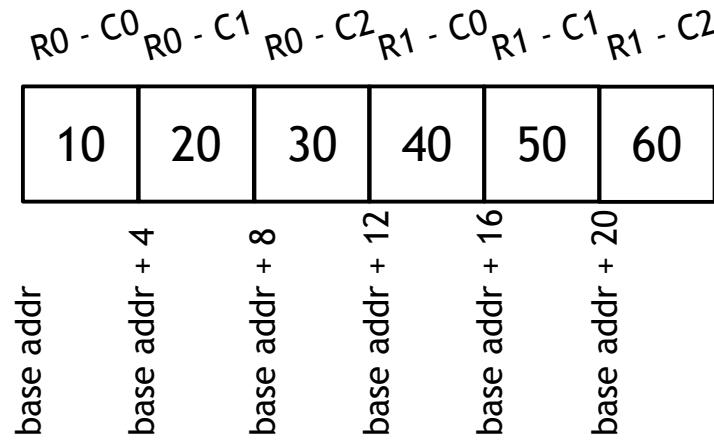
```
data_type name[ROW][COL];
```

Where ROW \* COL represents number of elements

Memory occupied by array = (number of elements \* size of an element)  
= (ROW \* COL \* <size of data\_type>)

### Example

```
int a[2][3] = {{10, 20, 30}, {40, 50, 60}};
```

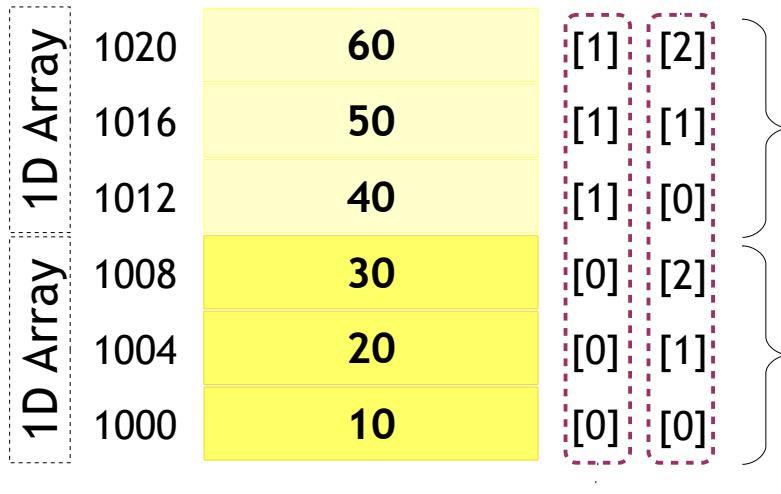


# Advanced C

## Arrays - 2D - Referencing



2 \* 1D array linearly placed in memory



Index to access the  
1D array

2<sup>nd</sup> 1D Array with base address 1012  
 $a[1] = \&a[1][0] = a + 1 \rightarrow 1012$

1<sup>st</sup> 1D Array with base address 1000  
 $a[0] = \&a[0][0] = a + 0 \rightarrow 1000$

# Advanced C

## Arrays - 2D - Dereferencing



### Core Principle

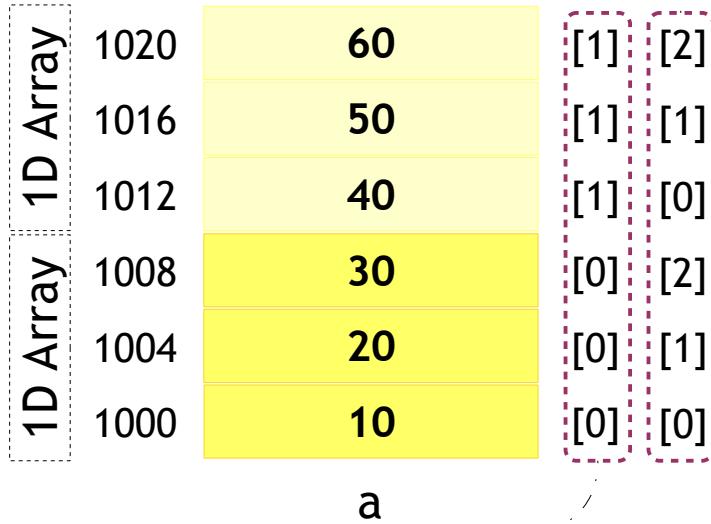
- Dereferencing  $n^{\text{th}}$  - dimensional array will return  $(n - 1)^{\text{th}}$  -dimensional array
  - Example : dereferencing 2D array will return 1D array
- Dereferencing 1D array will return 'data element'
  - Example : Dereferencing 1D integer array will return integer

| Array               | Dimension |
|---------------------|-----------|
| <code>&amp;a</code> | $n + 1$   |
| <code>a</code>      | $n$       |
| <code>*a</code>     | $n - 1$   |

# Advanced C

## Arrays - 2D - Dereferencing

2 \* 1D array linearly placed in memory



Index to access the 1D array

Example 1: Say  $a[0][1]$  is to be accessed, then decomposition happens like,

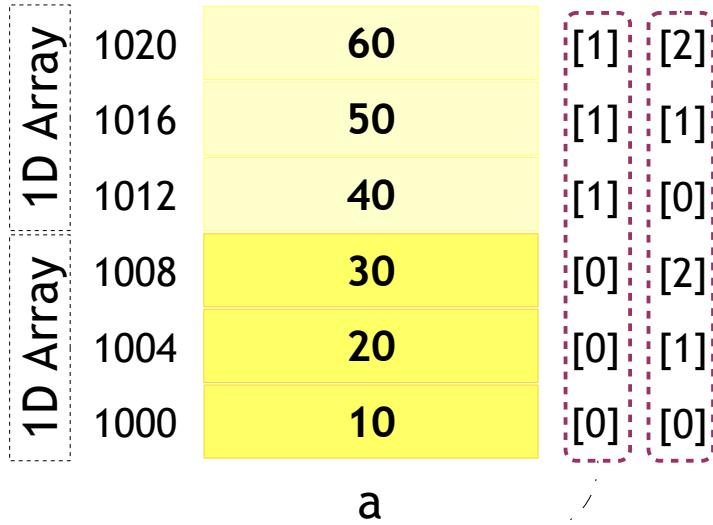
$a[0][1] =$

$$\begin{aligned} &= *(a[0] + (1 * \text{sizeof(type)})) \\ &= *(*a + (0 * \text{sizeof(1D array)})) + (1 * \text{sizeof(type)}) \\ &= *(*a + (0 * 12)) + (1 * 4) \\ &= *(*a + 0) + 4 \\ &= *(a + 0) + 4 \\ &= *(a + 4) \\ &= *(1000 + 4) \\ &= *(1004) \\ &= 20 \end{aligned}$$

# Advanced C

## Arrays - 2D - Dereferencing

2 \* 1D array linearly placed in memory



Index to access the  
1D array

**Example 1:** Say `a[1][1]` is to be accessed, then decomposition happens like,

`a[1][1] =`

$$\begin{aligned} &= *(a[1] + (1 * \text{sizeof(type)})) \\ &= *(*(\text{a} + (1 * \text{sizeof(1D array)})) + (1 * \text{sizeof(type)})) \\ &= *(*(\text{a} + (1 * 12)) + (1 * 4)) \\ &= *(*(\text{a} + 12) + 4) \\ &= *(*\text{a} + 12 + 4) \\ &= *(*\text{a} + 16) \\ &= *(1000 + 16) \\ &= *(1016) \\ &= 50 \end{aligned}$$

Address of `a[r][c] = value(a) + r * sizeof(1D array) + c * sizeof(type)`

# Advanced C

## Arrays - 2D - DIY

- WAP to find the MIN and MAX of a 2D array

# Advanced C

## Pointers - Array of pointers



### Syntax

```
datatype *ptr_name[SIZE]
```

### 004\_example.c

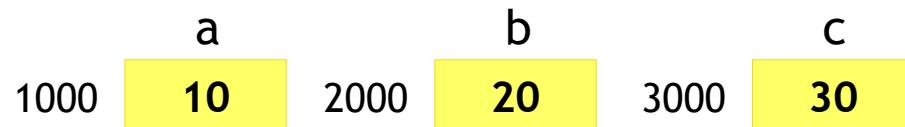
```
#include <stdio.h>
```

```
int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
```

```
    int *ptr[3];

    ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers



### Syntax

```
datatype *ptr_name[SIZE]
```

### 004\_example.c

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

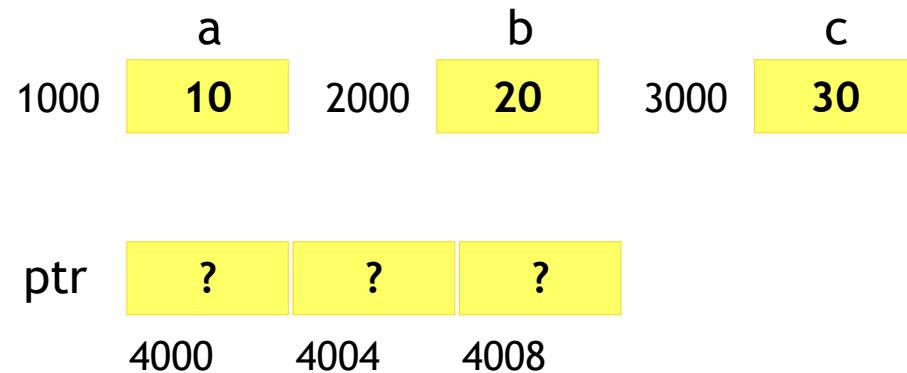
```
int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
```

```
→ [ int *ptr[3]; ]
```

```
ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
```

```
return 0;
```

```
}
```



# Advanced C

## Pointers - Array of pointers



### Syntax

```
datatype *ptr_name[SIZE]
```

### 004\_example.c

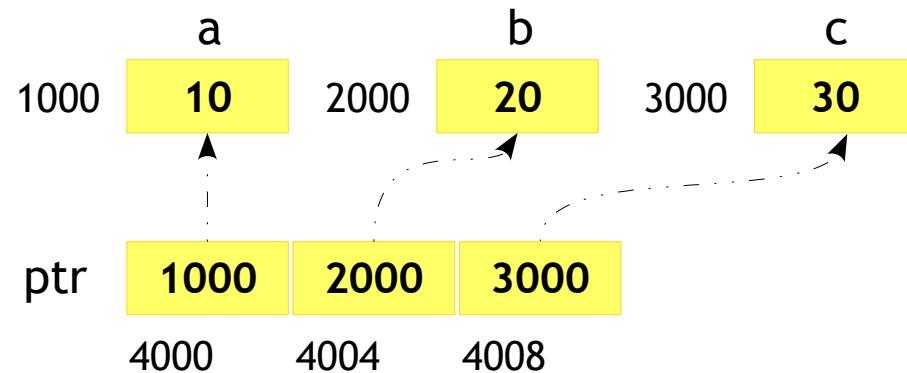
```
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;

    int *ptr[3];

    → ptr[0] = &a;
    ptr[1] = &b;
    ptr[2] = &c;

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers



### Syntax

```
datatype *ptr_name[SIZE]
```

### 005\_example.c

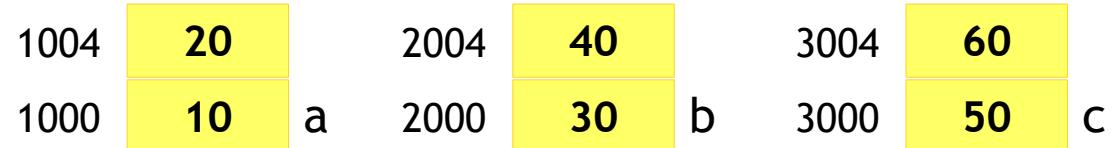
```
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

    int *ptr[3];

    ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers

### Syntax

```
datatype *ptr_name[SIZE]
```

### 005\_example.c

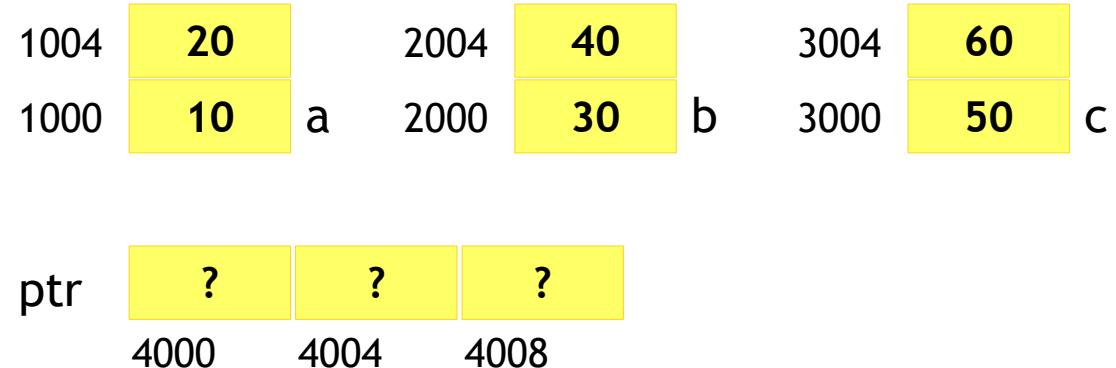
```
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

→ [ int *ptr[3];

    ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers

### Syntax

```
datatype *ptr_name[SIZE]
```

### 005\_example.c

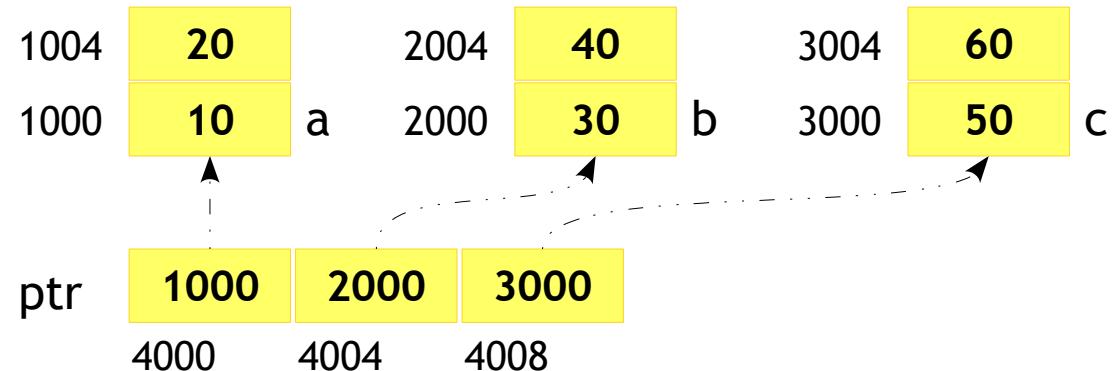
```
#include <stdio.h>

int main()
{
    int a[2] = {10, 20};
    int b[2] = {30, 40};
    int c[2] = {50, 60};

    int *ptr[3];

    → ptr[0] = a;
    ptr[1] = b;
    ptr[2] = c;

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers



### 006\_example.c

```
#include <stdio.h>

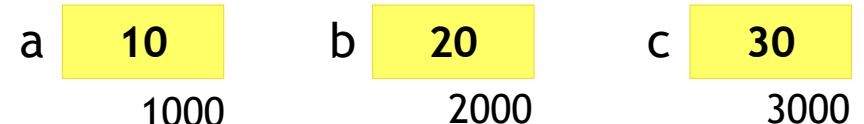
void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    → int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers

### 006\_example.c

```
#include <stdio.h>

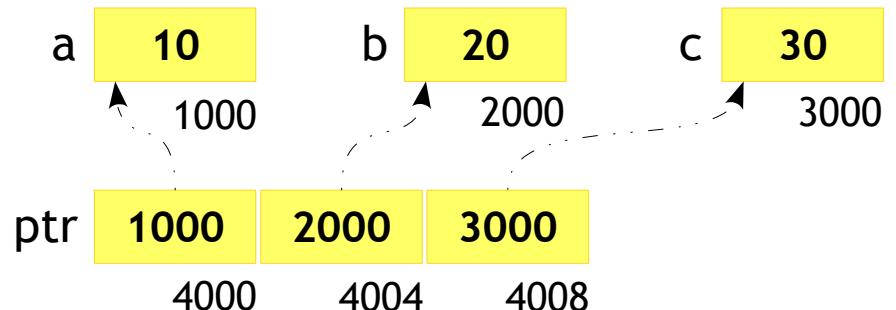
void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
→ int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers



### 006\_example.c

```
#include <stdio.h>

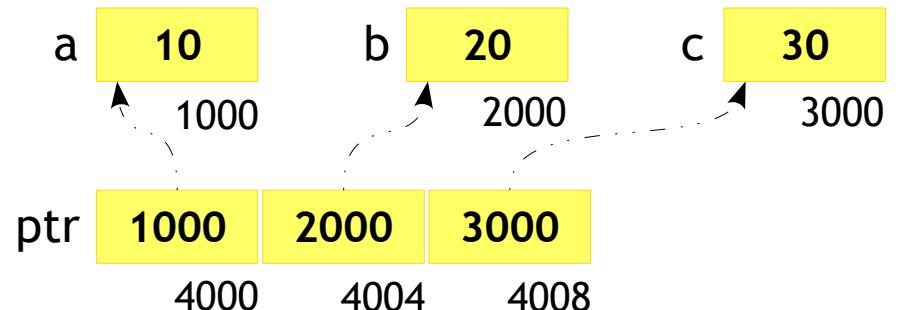
void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

→ print_array(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Array of pointers

### 006\_example.c

```
#include <stdio.h>

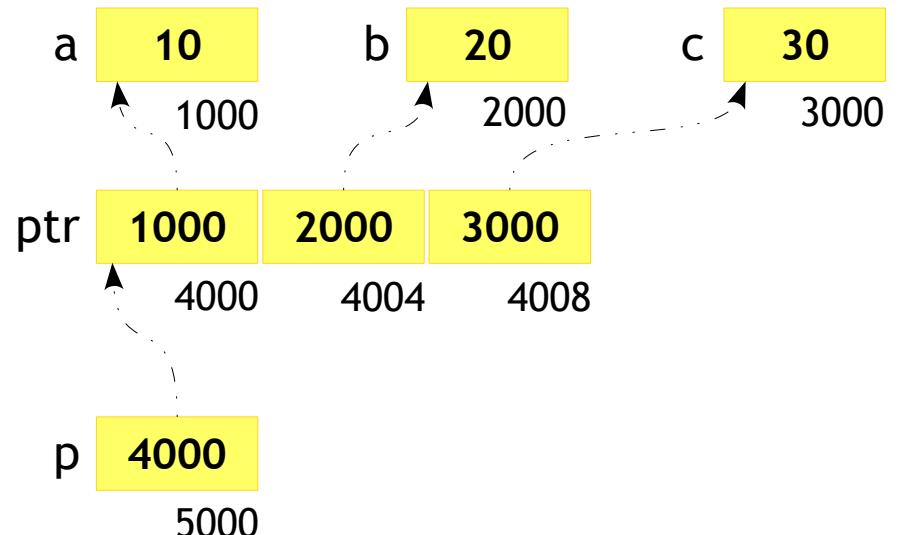
void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



# Advanced C

## Pointers - Array of strings

### 007\_example.c

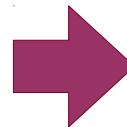
```
#include <stdio.h>

int main()
{
    →char s[3][8] = {
        "Array",
        "of",
        "Strings"
    }

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```

|      | s    |      | s    |
|------|------|------|------|
| 1000 | 'A'  | 1000 | 0x41 |
| 1001 | 'r'  | 1001 | 0x72 |
| 1002 | 'r'  | 1002 | 0x72 |
| 1003 | 'a'  | 1003 | 0x61 |
| 1004 | 'y'  | 1004 | 0x79 |
| 1005 | '\0' | 1005 | 0x00 |
| 1006 | ?    | 1006 | ?    |
| 1007 | ?    | 1007 | ?    |
| 1008 | 'o'  | 1008 | 0x6F |
| 1009 | 'f'  | 1009 | 0x66 |
| 1010 | '\0' | 1010 | 0x00 |
| 1011 | ?    | 1011 | ?    |
| 1012 | ?    | 1012 | ?    |
| 1013 | ?    | 1013 | ?    |
| 1014 | ?    | 1014 | ?    |
| 1015 | ?    | 1015 | ?    |
| 1016 | 'S'  | 1016 | 0x53 |
| 1017 | 't'  | 1017 | 0x74 |
| 1018 | 'r'  | 1018 | 0x72 |
| 1019 | 'i'  | 1019 | 0x69 |
| 1020 | 'n'  | 1020 | 0x6E |
| 1021 | 'g'  | 1021 | 0x67 |
| 1022 | 's'  | 1022 | 0x73 |
| 1023 | '\0' | 1023 | 0x00 |



# Advanced C

## Pointers - Array of strings



### 008\_example.c

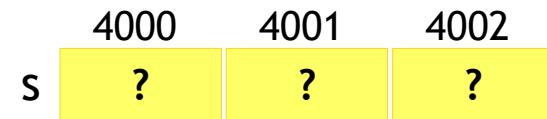
```
#include <stdio.h>

int main()
{
    ➔ char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



# Advanced C

## Pointers - Array of strings

### 008\_example.c

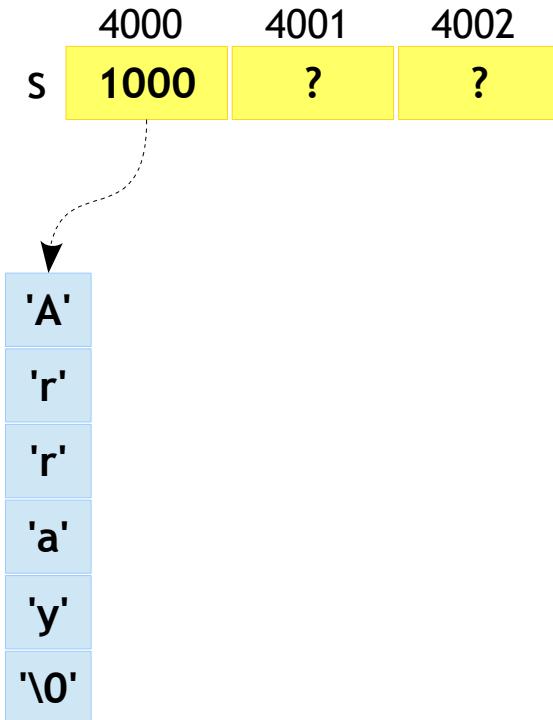
```
#include <stdio.h>

int main()
{
    char *s[3];

    → s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



# Advanced C

## Pointers - Array of strings

### 008\_example.c

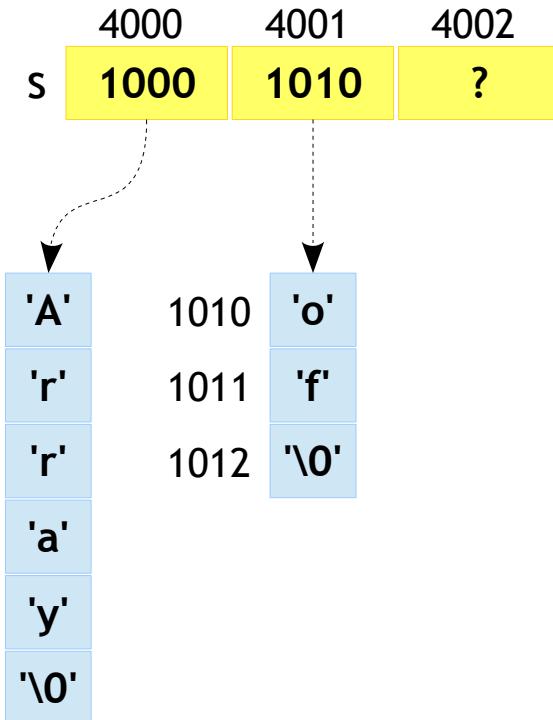
```
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    → s[1] = "of";
    s[2] = "Strings";

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



# Advanced C

## Pointers - Array of strings

### 008\_example.c

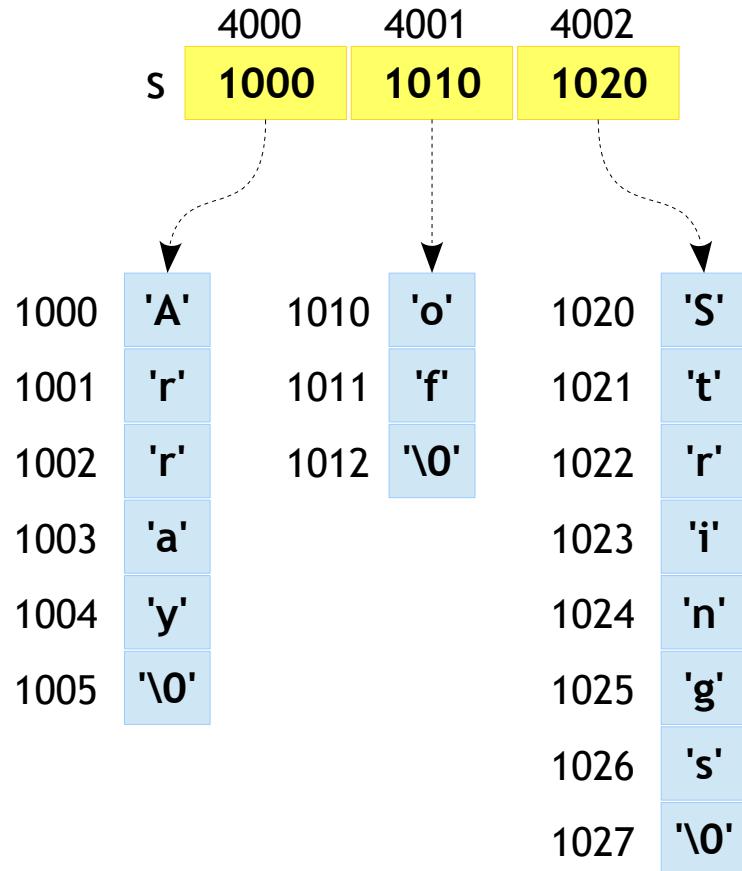
```
#include <stdio.h>

int main()
{
    char *s[3];

    s[0] = "Array";
    s[1] = "of";
    s[2] = "Strings"; → [s[2] = "Strings";]

    printf("%s %s %s\n", s[0], s[1], s[2]);

    return 0;
}
```



# Advanced C

## Pointers - Array of strings



- W.A.P to print menu and select an option
  - Menu options { File, Edit, View, Insert, Help }
- The prototype of print\_menu function
  - void print\_menu (char \*\*menu);

### Screen Shot

```
user@user:~]
user@user:~]./a.out
1. File
2. Edit
3. View
4. Insert
5. Help
Select your option: 2
You have selected Edit Menu
user@user:~]
```

# Advanced C

## Pointers - Array of strings

- Command line arguments
  - Refer to PPT “11\_functions\_part2”

# Advanced C

## Pointers - Pointer to an Array



### Syntax

```
datatype (*ptr_name) [SIZE];
```

### 009\_example.c

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

    ptr = array;

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

    return 0;
}
```

- Pointer to an array!!, why is the syntax so weird??
- Isn't the code shown left is an example for pointer to an array?
- Should the code print as 1 in output?
- Yes, everything is fine here except the dimension of the array.
- This is perfect code for 1D array

# Advanced C

## Pointers - Pointer to an Array

### Syntax

```
datatype (*ptr_name) [SIZE];
```

### 010\_example.c

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

    ptr = array;

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

    return 0;
}
```

- So in order to point to 2D array we would prefer the given syntax
- Okay, Isn't a 2D array linearly arranged in the memory?

So can I write the code as shown?
- Hmm!, Yes but the compiler would warn you on the assignment statement
- Then how should I write?

# Advanced C

## Pointers - Pointer to an Array

### Syntax

```
datatype (*ptr_name) [SIZE];
```

### 011\_example.c

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

    ptr = &array;

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

    return 0;
}
```

- Hhoho, isn't **array** is equal to **&array**?? what is the difference?
- Well the difference lies in the compiler interpretation while pointer arithmetic and hence
- Please see the difference in the next slides

# Advanced C

## Pointers - Pointer to an Array



### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```

|      | array |
|------|-------|
| 1000 | 1     |
| 1004 | 2     |
| 1008 | 3     |
| 1012 | ?     |
| 1016 | ?     |
| 1020 | ?     |
| 1024 | ?     |
| 1028 | ?     |
| 1032 | ?     |
| 1036 | ?     |

# Advanced C

## Pointers - Pointer to an Array

### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    → int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```

p1

?

2000

array

|      |   |
|------|---|
| 1000 | 1 |
| 1004 | 2 |
| 1008 | 3 |
| 1012 | ? |
| 1016 | ? |
| 1020 | ? |
| 1024 | ? |
| 1028 | ? |
| 1032 | ? |
| 1036 | ? |

# Advanced C

## Pointers - Pointer to an Array

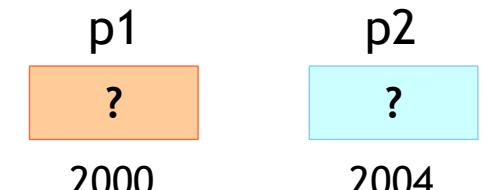
012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
→ int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



| array |   |
|-------|---|
| 1000  | 1 |
| 1004  | 2 |
| 1008  | 3 |
| 1012  | ? |
| 1016  | ? |
| 1020  | ? |
| 1024  | ? |
| 1028  | ? |
| 1032  | ? |
| 1036  | ? |

# Advanced C

## Pointers - Pointer to an Array

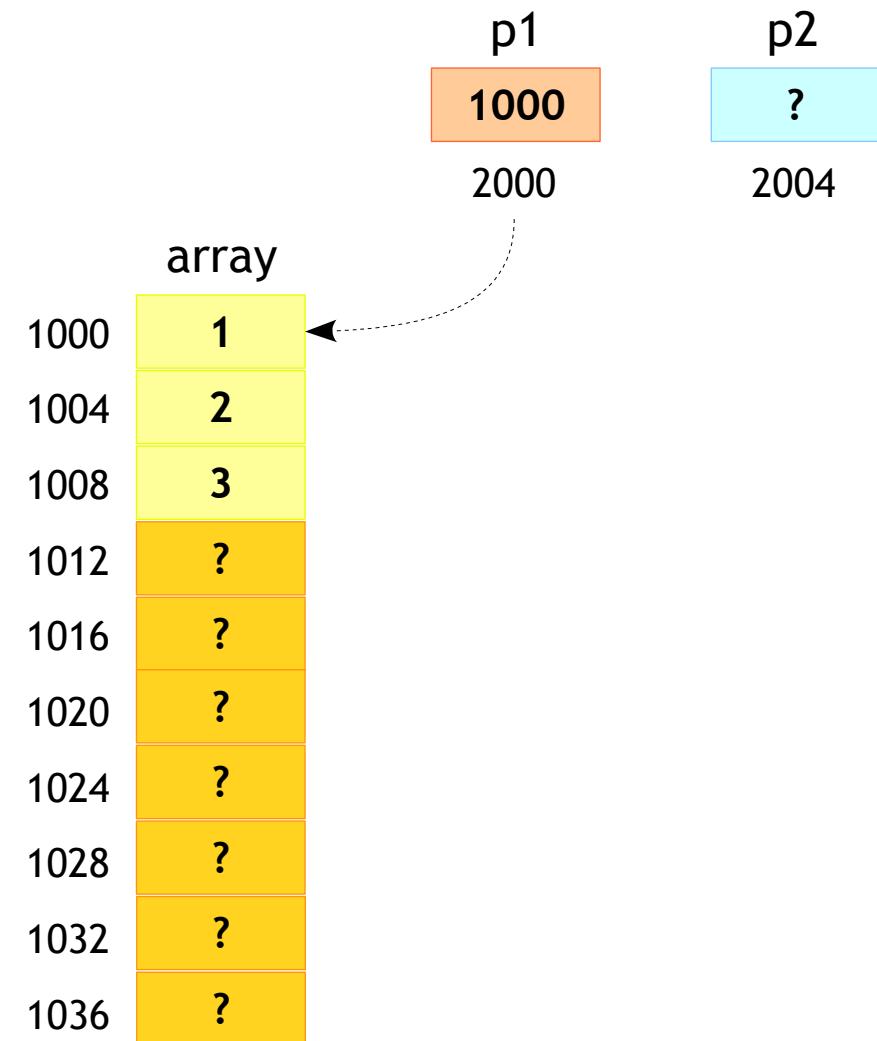
### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    → p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

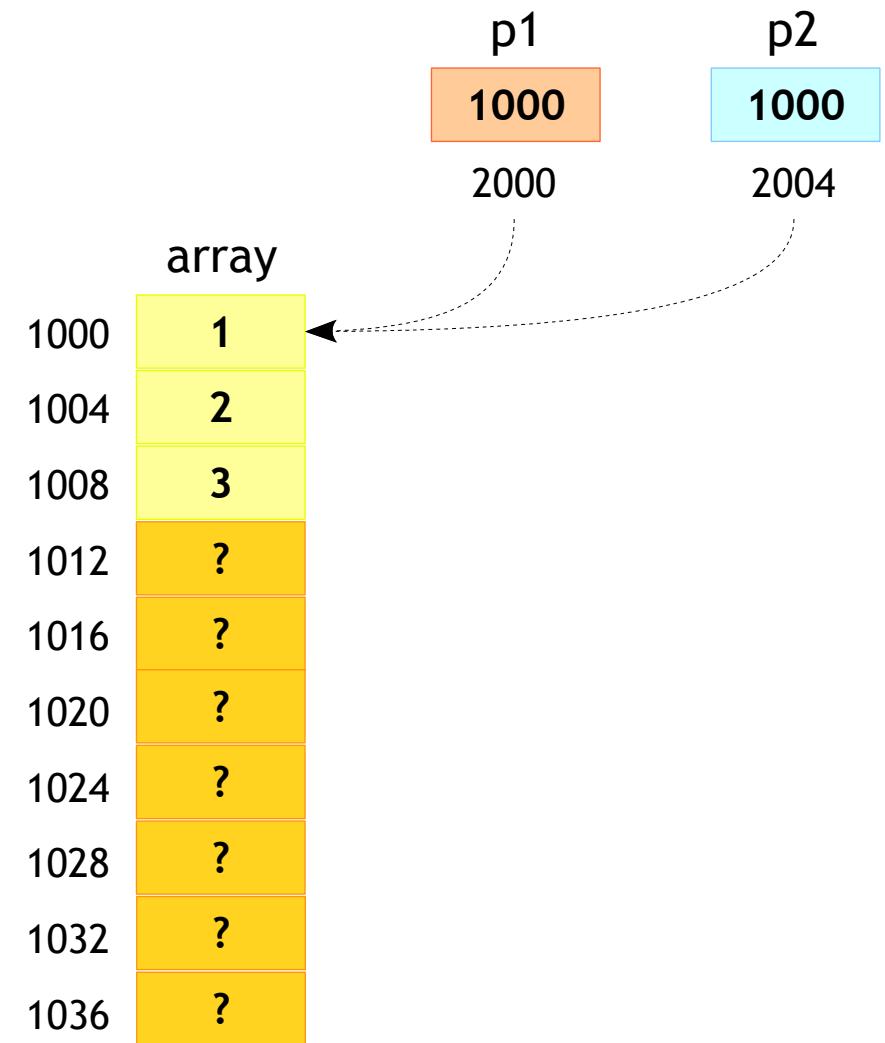
### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    →p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

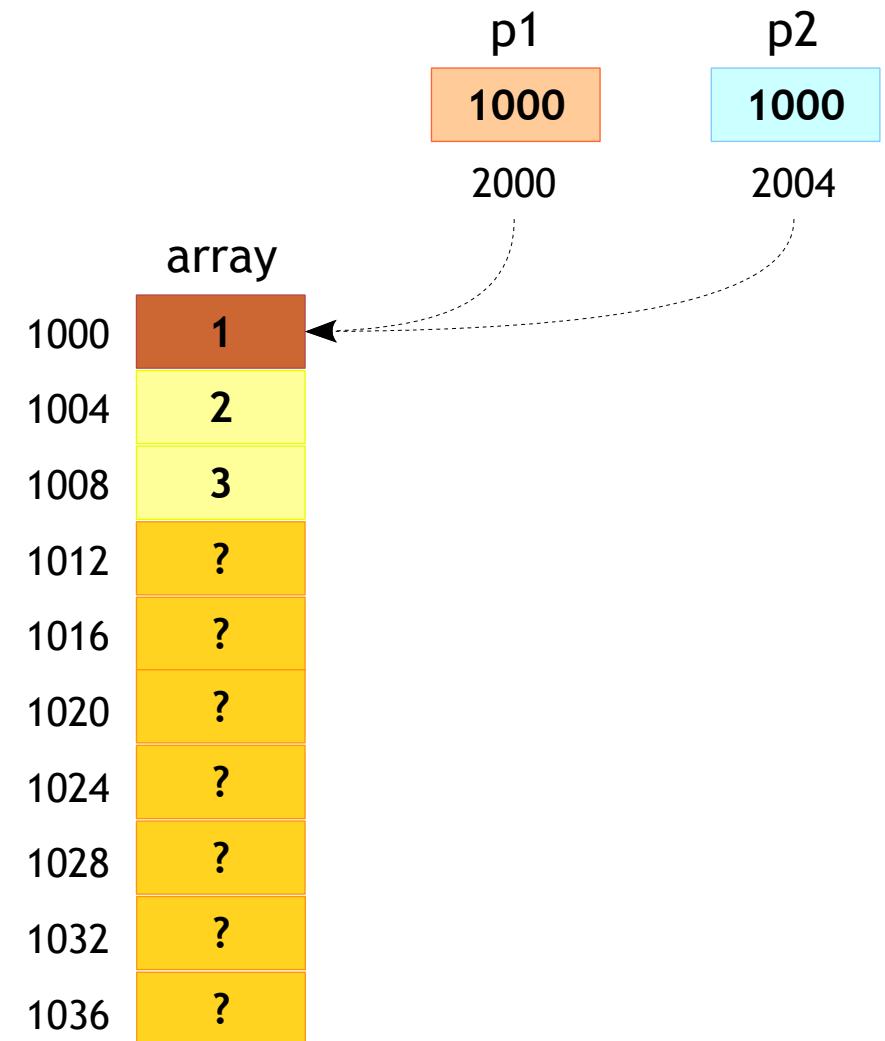
### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    →printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array



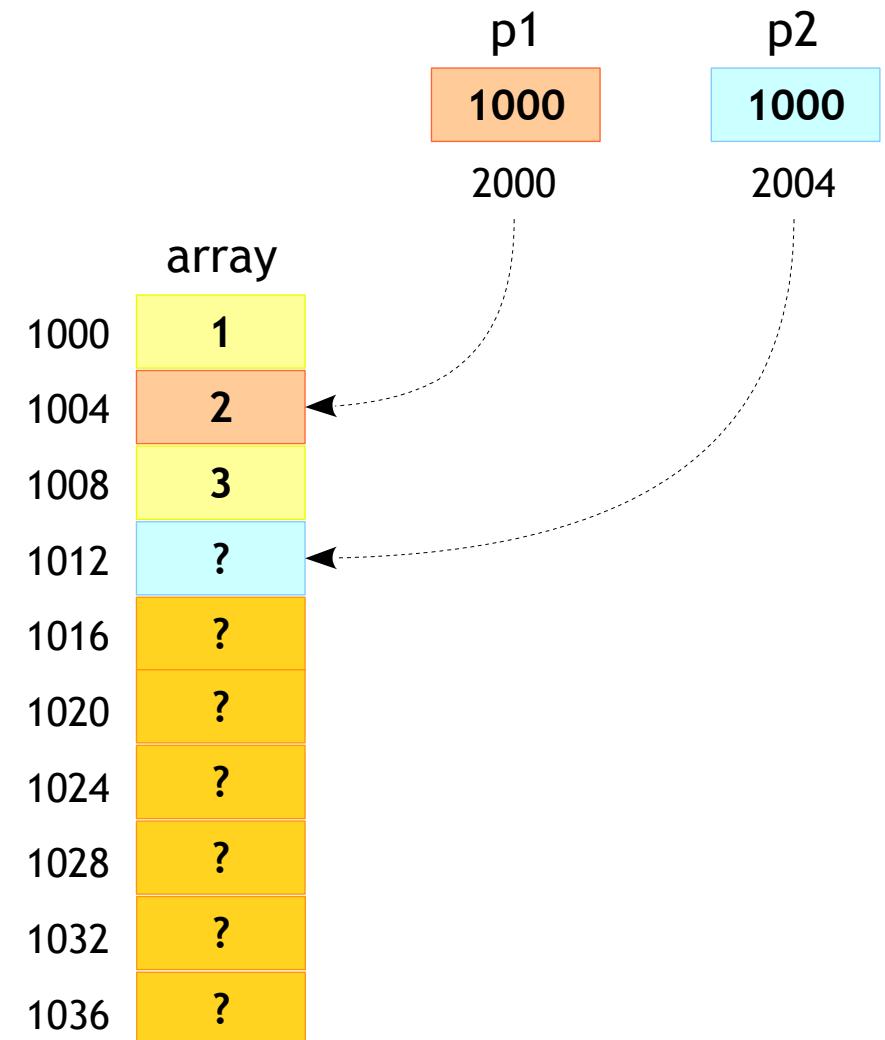
### 012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
    printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

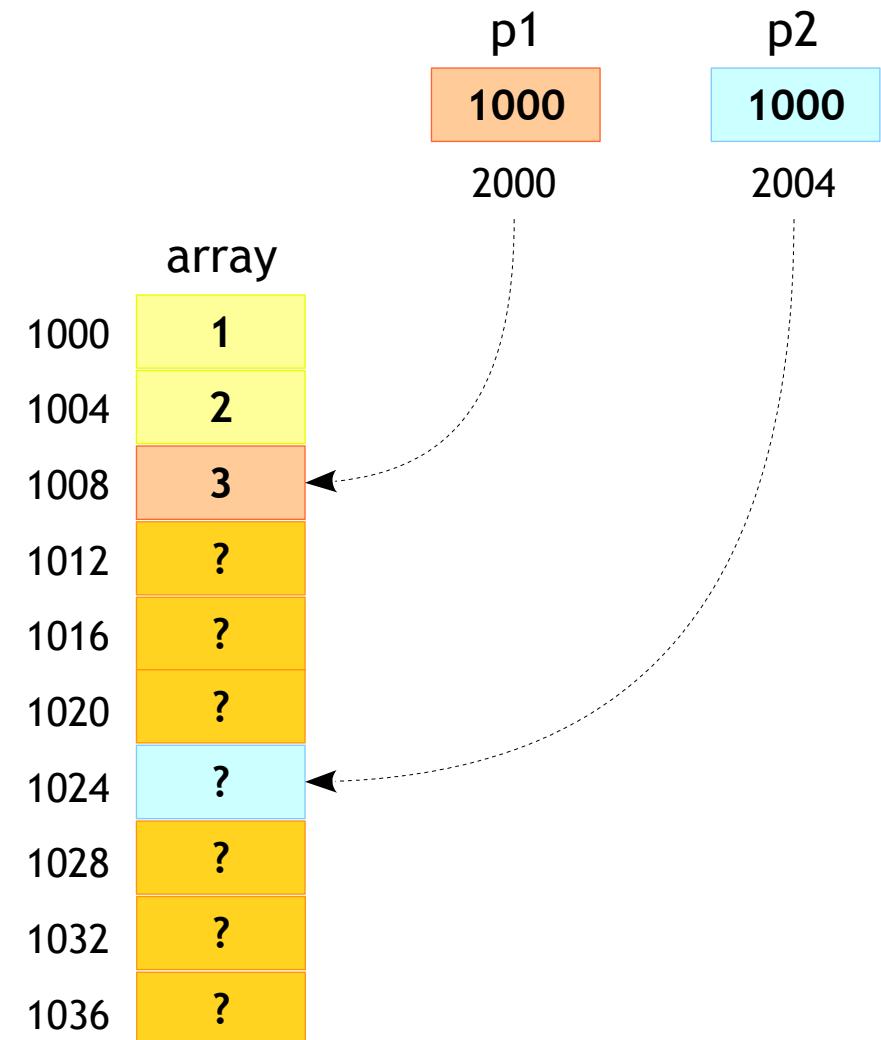
012\_example.c

```
int main()
{
    int array[3] = {1, 2, 3};
    int *p1;
    int (*p2)[3];

    p1 = array;
    p2 = &array;

    printf("%p %p\n", p1 + 0, p2 + 0);
    printf("%p %p\n", p1 + 1, p2 + 1);
→printf("%p %p\n", p1 + 2, p2 + 2);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array



- So as a conclusion we can say the
  - Pointer arithmetic on 1D array is based on the **size of datatype**
  - Pointer arithmetic on 2D array is based on the **size of datatype** and **size of 1D array**
- Still one question remains is what is real use of this syntax if can do **p[i][j]**?
  - In case of dynamic memory allocation as shown in next slide

# Advanced C

## Pointers - Pointer to an Array

### 013\_example.c

```
int main()
{
    →int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*p + 0)[0] = 1;
    (*p + 1)[1] = 2;
    (*p + 2)[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```

p

?

2000

# Advanced C

## Pointers - Pointer to an Array

### 013\_example.c

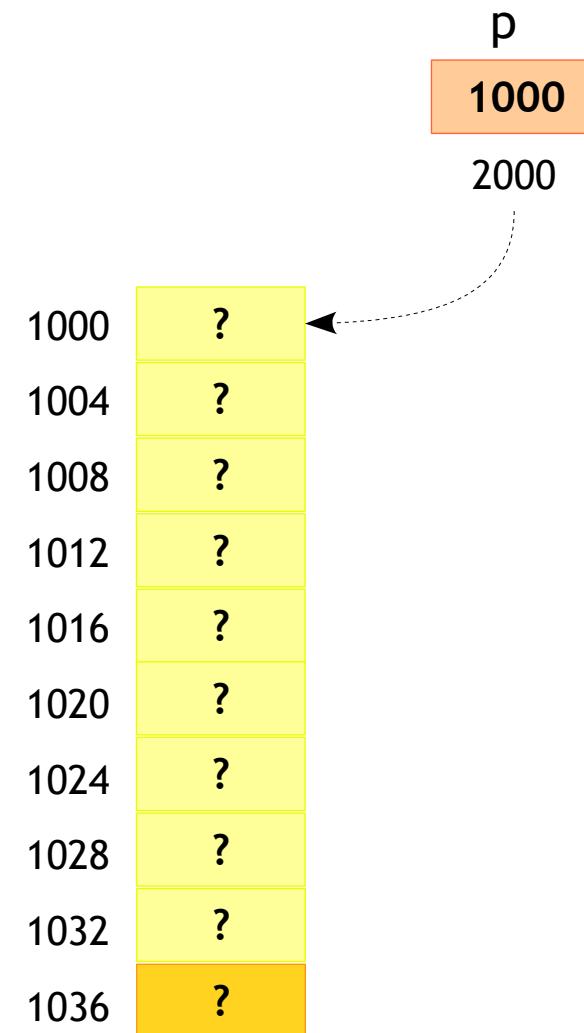
```
int main()
{
    int (*p)[3];

→ p = malloc(sizeof(*p) * 3);

    (*p + 0)[0] = 1;
    (*p + 1)[1] = 2;
    (*p + 2)[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array



### 013\_example.c

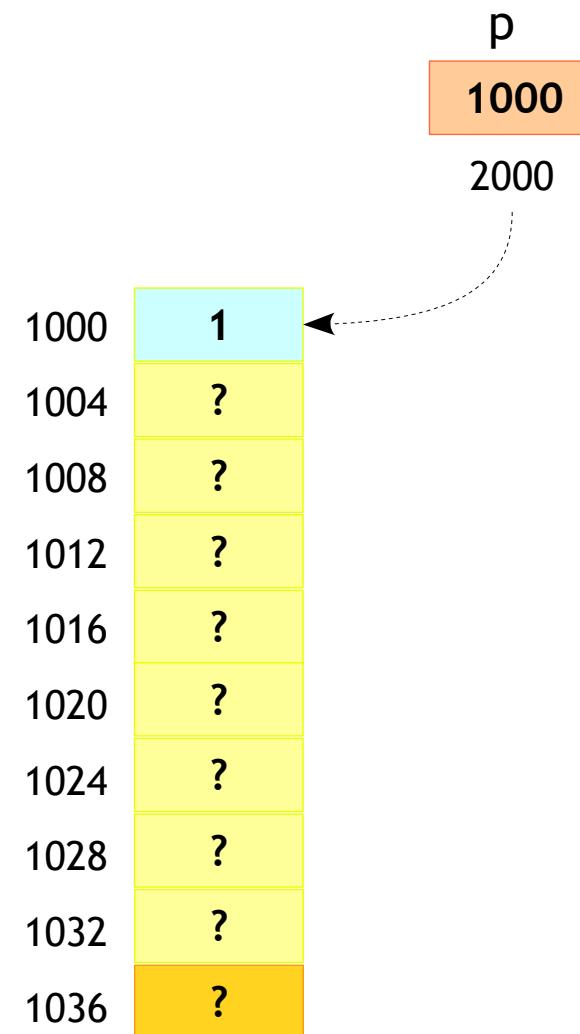
```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    → (*p + 0)[0] = 1;
    (*p + 1)[1] = 2;
    (*p + 2)[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

### 013\_example.c

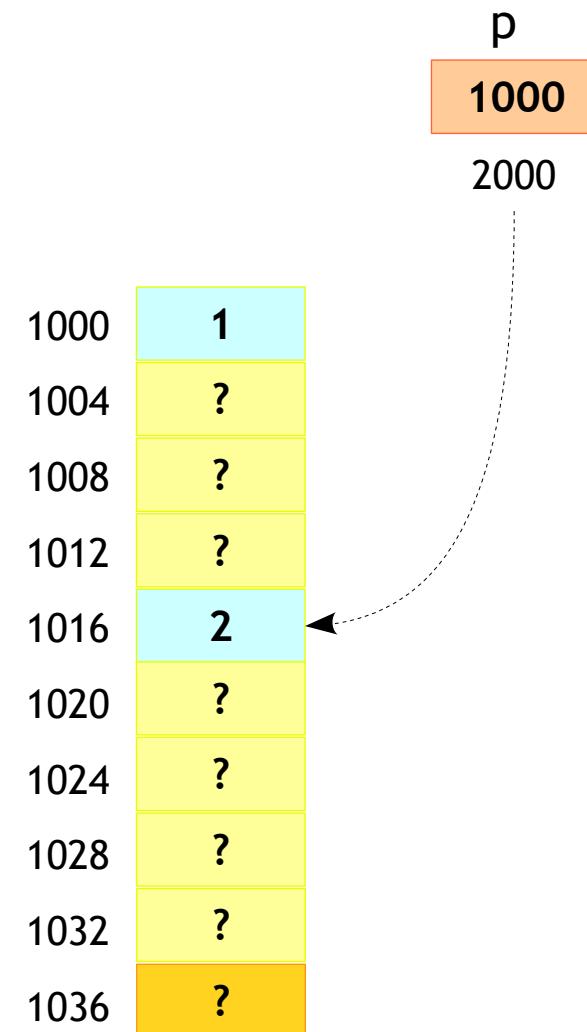
```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*p + 0)[0] = 1;
    (*p + 1)[1] = 2;
    (*p + 2)[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```



# Advanced C

## Pointers - Pointer to an Array

### 013\_example.c

```
int main()
{
    int (*p)[3];

    p = malloc(sizeof(*p) * 3);

    (*p + 0)[0] = 1;
    (*p + 1)[1] = 2;
    → (*p + 2)[2] = 3;

    printf("%d\n", p[0][0]);
    printf("%d\n", p[1][1]);
    printf("%d\n", p[2][2]);

    return 0;
}
```

p  
1000  
2000

|      |   |
|------|---|
| 1000 | 1 |
| 1004 | ? |
| 1008 | ? |
| 1012 | ? |
| 1016 | 2 |
| 1020 | ? |
| 1024 | ? |
| 1028 | ? |
| 1032 | 3 |
| 1036 | ? |

# Advanced C

## Pointers - Pointer to an 2D Array

014\_example.c

```
int main()
{
    →int (*p)[3];
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```

p  
2000    1000

# Advanced C

## Pointers - Pointer to an 2D Array



014\_example.c

```
int main()
{
    int (*p)[3];
→ int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```

p  
2000 1000

a  
1000 1  
1004 2  
1008 3  
1012 4  
1016 5  
1020 6  
1024 ?  
1028 ?  
1032 ?  
1036 ?

# Advanced C

## Pointers - Pointer to an 2D Array

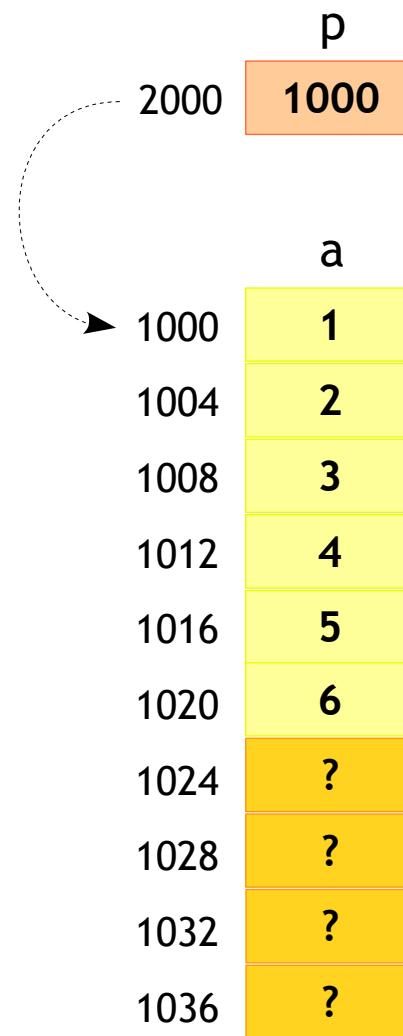


014\_example.c

```
int main()
{
    int (*p)[3];
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    → p = a;

    return 0;
}
```



# Advanced C

## Pointers - Passing 2D array to function

### 015\_example.c

```
#include <stdio.h>

void print_array(int p[2][3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```

| a      |
|--------|
| 1000 1 |
| 1004 2 |
| 1008 3 |
| 1012 4 |
| 1016 5 |
| 1020 6 |
| 1024 ? |
| 1028 ? |
| 1032 ? |
| 1036 ? |

# Advanced C

## Pointers - Passing 2D array to function

### 015\_example.c

```
#include <stdio.h>

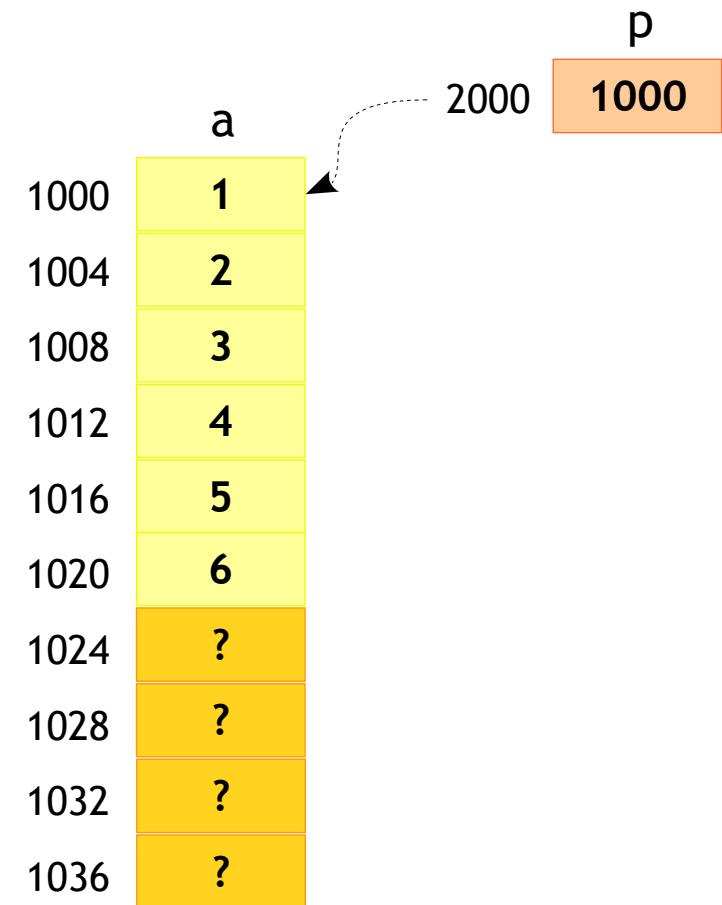
void print_array(int p[2][3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```



# Advanced C

## Pointers - Passing 2D array to function

### 016\_example.c

```
#include <stdio.h>

void print_array(int (*p)[3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```

| a      |
|--------|
| 1000 1 |
| 1004 2 |
| 1008 3 |
| 1012 4 |
| 1016 5 |
| 1020 6 |
| 1024 ? |
| 1028 ? |
| 1032 ? |
| 1036 ? |

# Advanced C

## Pointers - Passing 2D array to function

### 016\_example.c

```
#include <stdio.h>

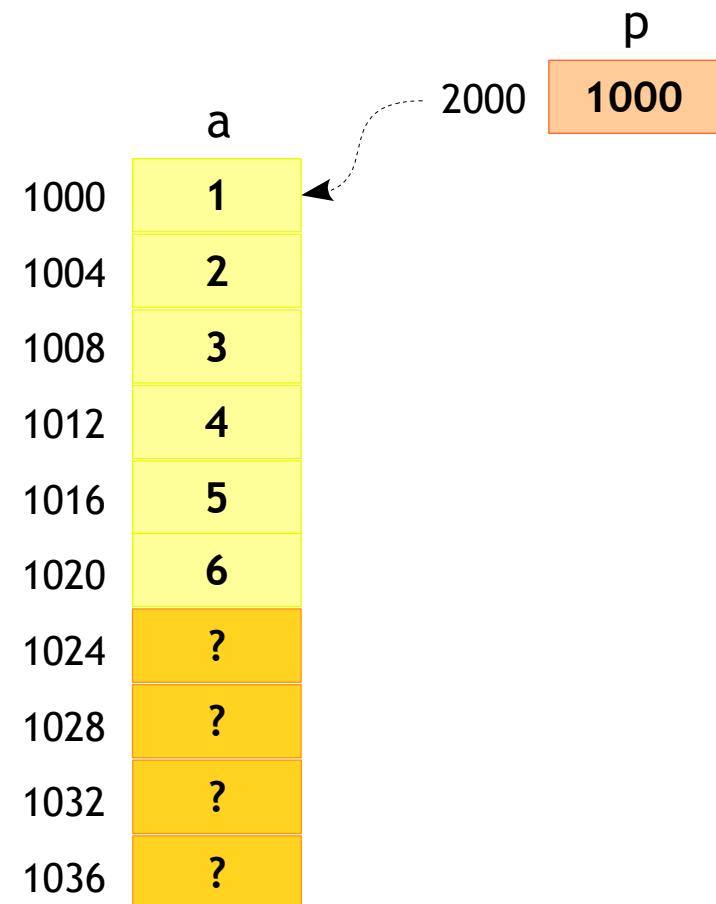
void print_array(int (*p) [3])
{
    int i, j;

    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 3; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(a);

    return 0;
}
```



# Advanced C

## Pointers - Passing 2D array to function

### 017\_example.c

```
#include <stdio.h>

void print_array(int row, int col, int (*p)[col])
{
    int i, j;

    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            printf("%d\n", p[i][j]);
        }
    }
}

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

    print_array(2, 3, a);

    return 0;
}
```

# Advanced C

## Pointers - Passing 2D array to function

### 018\_example.c

```
#include <stdio.h>

void print_array(int row, int col, int *p)
{
    int i, j;

    for (i = 0; i < row; i++)
    {
        for (j = 0; j < col; j++)
        {
            printf("%d\n", *((p + i * col) + j));
        }
    }
}

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

    print_array(2, 3, (int *) a);

    return 0;
}
```

# Advanced C

## Pointers - 2D Array Creations



- Each Dimension could be Static or Dynamic
- Possible combination of creation could be
  - BS: Both Static (Rectangular)
  - FSSD: First Static, Second Dynamic
  - FDSS: First Dynamic, Second Static
  - BD: Both Dynamic



# Advanced C

## Pointers - 2D Array Creations - BS

018\_example.c

```
#include <stdio.h>

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

    return 0;
}
```

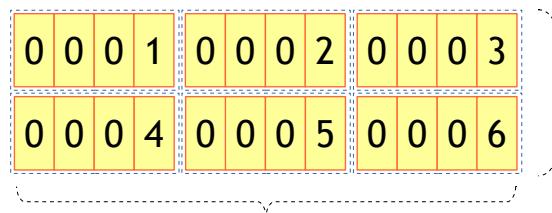
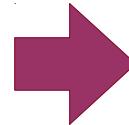
- Both Static (BS)
- Called as an rectangular array
- Total size is  
$$2 * 3 * \text{sizeof(datatype)}$$
$$2 * 3 * 4 = 24 \text{ Bytes}$$
- The memory representation can be as shown in next slide

# Advanced C

## Pointers - 2D Array Creations - BS



| a    |
|------|
| 1000 |
| 1004 |
| 1008 |
| 1012 |
| 1016 |
| 1020 |



**Static  
2 Rows  
On Stack**

**Static  
3 Columns  
On Stack**

# Advanced C

## Pointers - 2D Array Creations - FSSD

### 019\_example.c

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

int main()
{
    int *a[2];

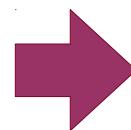
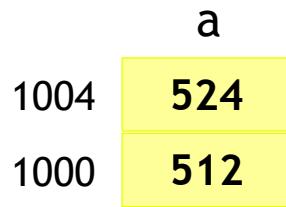
    for ( i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

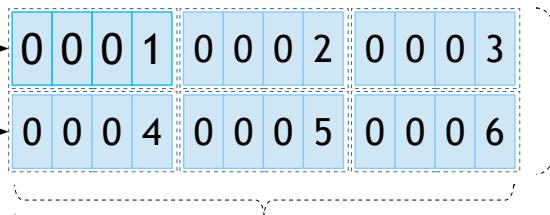
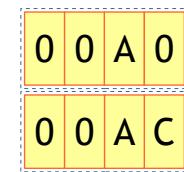
- First Static and Second Dynamic (FSSD)
- Mix of Rectangular & Ragged
- Total size is
$$2 * \text{sizeof(datatype *)} + \\ 2 * 3 * \text{sizeof(datatype)}$$
$$2 * 4 + 2 * 3 * 4 = 32 \text{ Bytes}$$
- The memory representation can be as shown in next slide

# Advanced C

## Pointers - 2D Array Creations - FSSD



Pointers to  
2 Rows  
On Stack



Dynamic  
3 Columns  
On Heap

Static  
2 Rows  
On Heap

# Advanced C

## Pointers - 2D Array Creations - FDSS

### 020\_example.c

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

int main()
{
    int (*a) [3];

    a = malloc(2 * sizeof(int [3]));

    return 0;
}
```

- First Dynamic and Second Static (FDSS)
- Total size is  
 $\text{sizeof(datatype *)} + 2 * 3 * \text{sizeof(datatype)}$   
 $4 + 2 * 3 * 4 = 28 \text{ Bytes}$
- The memory representation can be as shown in next slide

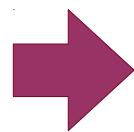
# Advanced C

## Pointers - 2D Array Creations - FDSS

a

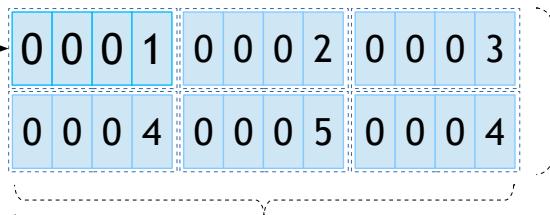
1000 512

|     |   |
|-----|---|
| 532 | 6 |
| 528 | 5 |
| 524 | 4 |
| 520 | 3 |
| 516 | 2 |
| 512 | 1 |



Pointer to  
1 Row  
On Stack

0 0 A 0



Static  
3 Columns  
On Heap

Dynamic  
2 Rows  
On Heap

# Advanced C

## Pointers - 2D Array Creations - BD

### 021\_example.c

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

int main()
{
    int **a;
    int i;

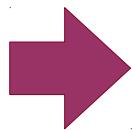
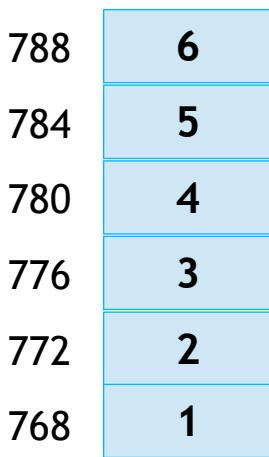
    a = malloc(2 * sizeof(int *));
    for (i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

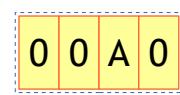
- Both Dynamic (BD)
- Total size is  
 $\text{sizeof(datatype } \star\star) +$   
 $2 * \text{sizeof(datatype } \star) +$   
 $2 * 3 * \text{sizeof(datatype)}$   
 $4 + 2 * 4 + 2 * 3 * 4 = 28$   
Bytes
- The memory representation can be as shown in next slide

# Advanced C

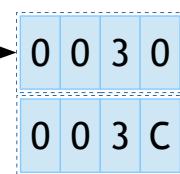
## Pointers - 2D Array Creations - BD



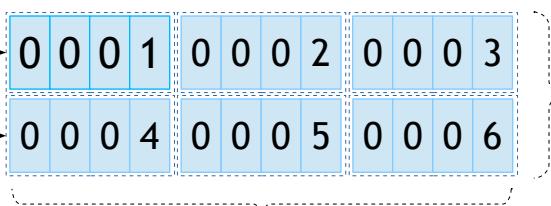
Pointer to  
1 Row  
On Stack



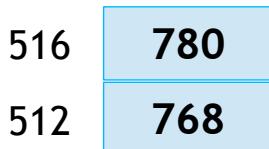
Pointers to  
2 Rows  
On Heap



Dynamic  
3 Columns  
On Heap



Dynamic  
2 Rows  
On Heap



# Advanced Functions



# Command Line Arguments



# Advanced C

## Functions - Command Line Arguments

### Example

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    return 0;
}
```

Environmental Variables

Passed Arguments on CL

Arguments Count

### Usage

```
user@user:~] ./a.out 5 + 3
```

4<sup>th</sup> argument

3<sup>rd</sup> argument

2<sup>nd</sup> argument

1<sup>st</sup> argument

Total counts of the args stored  
in **argc**

All stored in **argv**

# Advanced C

## Functions - Command Line Arguments

### 001\_example.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("No of argument(s) : %d\n", argc);

    printf("List of argument(s) :\n");
    for (i = 0; i < argc; i++)
    {
        printf("\t%d - \"%s\"\n", i + 1, argv[i]);
    }

    return 0;
}
```

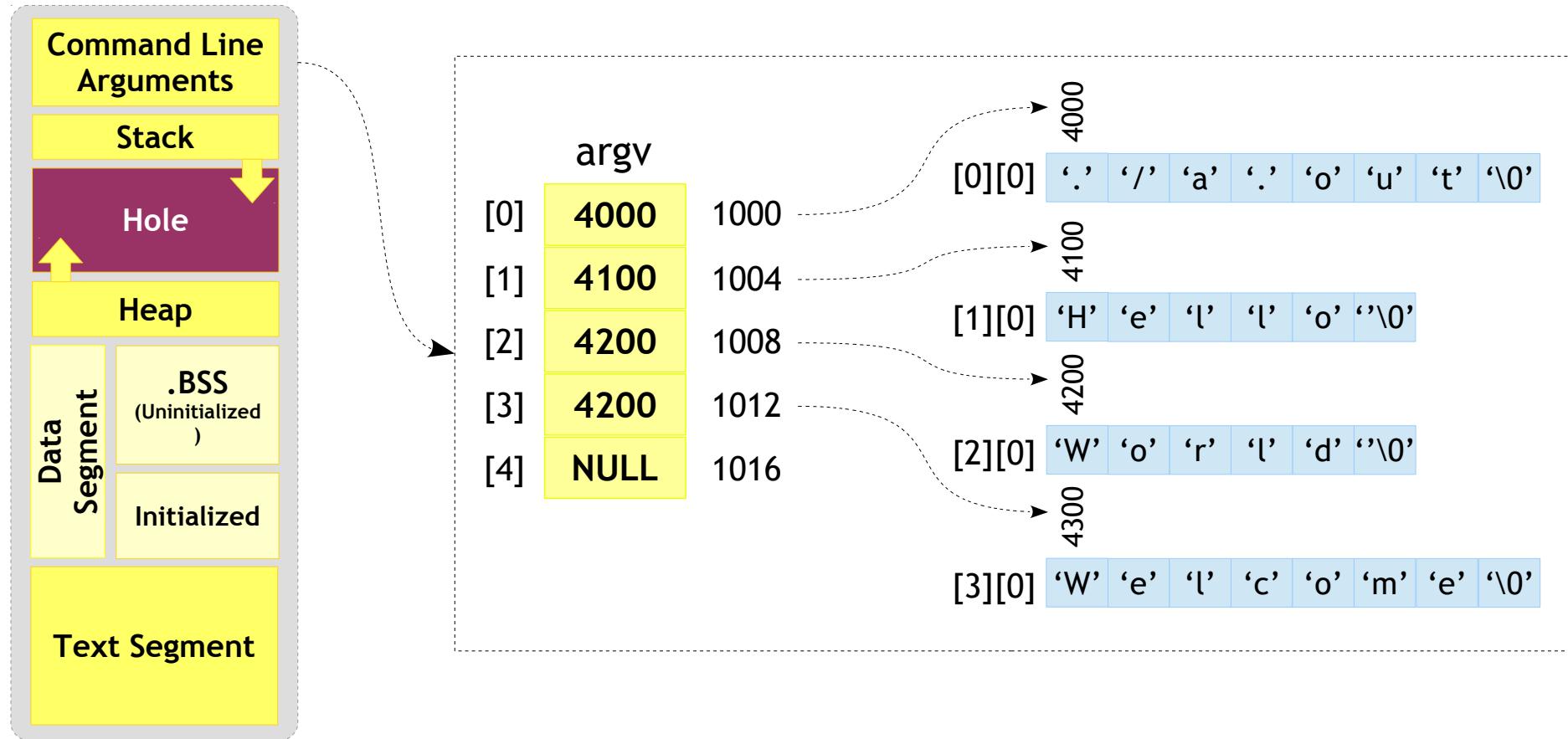
# Advanced C

## Functions - Command Line Arguments

### Example

```
user@user:~] ./a.out Hello World Welcome
```

### Memory Segments



# Advanced C

## Functions - Command Line Arguments - DIY

- Print all the Environmental Variables
- WAP to calculate average of numbers passed via command line

# Function Pointer



# Advanced C

## Functions - Function Pointers



- A variable that stores the address of a function.  
Therefore, points to the function.

### Syntax

```
return_datatype (*foo)(list of argument(s) datatype);
```

# Advanced C

## Functions - Function Pointers



### 002\_example.c

```
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    printf("%p\n", add);
    printf("%p\n", &add);

    return 0;
}
```

- Every function code would be stored in the text segment with an address associated with it
- This example would print the address of the **add** function

# Advanced C

## Functions - Function Pointers

### 003\_example.c

```
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int *fptr;

    fptr = add;

    printf("%p\n", add);
    printf("%p\n", fptr);
    printf("%p\n", &fptr);

    return 0;
}
```

- Hold on!!.. Can't I store the address on the normal pointer??
- Well, Yes you can! But how would you expect the compiler to interpret this?
- The compiler interprets this as a pointer to normal variable and not the code
- Then how to do it?

# Advanced C

## Functions - Function Pointers

### 004\_example.c

```
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int (*fptr) (int, int);

    fptr = add;

    printf("%p\n", add);
    printf("%p\n", fptr);
    printf("%p\n", &fptr);

    return 0;
}
```

- The address of the function should be stored in a function pointer
- Not to forget that the function pointer is a variable and would have address for itself

# Advanced C

## Functions - Function Pointers

### 005\_example.c

```
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int (*fptr) (int, int);

    fptr = add;

    printf("%d\n", fptr(2, 4));
    printf("%d\n", (*fptr)(2, 4));

    return 0;
}
```

- The function pointer could be invoked as shown in the example

# Advanced C

## Functions - Func Ptr - Passing to functions

### 006\_example.c

```
#include <stdio.h>

int main()
{
    int (*fptr)(int, int);

    fptr = add;
    printf("%d\n", oper(fptr, 2, 4));

    fptr = sub;
    printf("%d\n", oper(fptr, 2, 4));

    return 0;
}
```

```
int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int oper(int (*f)(int, int), int a, int b)
{
    return f(a, b);
}
```

# Advanced C

## Functions - Array of Function Pointers

### 007\_example.c

```
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int main()
{
    int (*f[]) (int, int) = {add, sub};

    printf("%d\n", f[0] (2, 4));
    printf("%d\n", f[1] (2, 4));

    return 0;
}
```

# Advanced C

## Functions - Array of Function Pointers

### 008\_example.c

```
#include <stdio.h>

int main()
{
    int (*f[]) (int, int) = {add, sub};

    printf("%d\n", oper(f[0], 2, 4));
    printf("%d\n", oper(f[1], 2, 4));

    return 0;
}
```

```
int add(int num1, int num2)
{
    return num1 + num2;
}

int sub(int num1, int num2)
{
    return num1 - num2;
}

int oper(int (*f)(int, int), int a, int b)
{
    return f(a, b);
}
```

# Advanced C

## Functions - Func Ptr - Std Functions - atexit()

### 009\_example.c

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

static int *ptr;

int main()
{
    /*
     * Registering a callback
     * Function
     */
    atexit(my_exit);

    /* Allocation in main */
    ptr = malloc(100);

    test();

    printf("Hello\n");

    return 0;
}
```

```
void my_exit(void)
{
    printf("Exiting program\n");

    if (ptr)
    {
        /* Deallocation in my_exit */
        free(ptr);
    }
}

void test(void)
{
    puts("In test");

    exit(0);
}
```

# Advanced C

## Functions - Func Ptr - Std Functions - qsort()

### 010\_example.c

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

int main()
{
    int a[5] = {9, 2, 6, 1, 7};

    qsort(a, 5, sizeof(int), sa);
    printf("Ascending: ");
    print(a, 5);

    qsort(a, 5, sizeof(int), sd);
    printf("Descending: ");
    print(a, 5);

    return 0;
}
```

```
int sa(const void *a, const void *b)
{
    return *(int *) a > *(int *) b;
}

int sd(const void *a, const void *b)
{
    return *(int *) a < *(int *) b;
}

void print(int *a, unsigned int size)
{
    int i = 0;

    for (i = 0; i < size; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");
}
```

# Variadic Functions



# Advanced C

## Functions - Variadic



- Variadic functions can be called with any number of trailing arguments
- For example,  
printf(), scanf() are common variadic functions
- Variadic functions can be called in the usual way with individual arguments

### Syntax

```
return_data_type function_name(parameter list, ...);
```

# Advanced C

## Functions - Variadic - Definition & Usage



- Defining and using a variadic function involves three steps:  
**Step 1:** Variadic functions are defined using an ellipsis ('...') in the argument list, and using special macros to access the variable arguments.

**Example**

```
int foo(int a, ...)  
{  
    /* Function Body */  
}
```

**Step 2:** Declare the function as variadic, using a prototype with an ellipsis ('...'), in all the files which call it.

**Step 3:** Call the function by writing the fixed arguments followed by the additional variable arguments.

# Advanced C

## Functions - Variadic - Argument access macros

- Descriptions of the macros used to retrieve variable arguments
- These macros are defined in the header file `stdarg.h`

| Type/Macros           | Description                                                                                                                                                                                                                                     |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>va_list</code>  | The type <code>va_list</code> is used for argument pointer variables                                                                                                                                                                            |
| <code>va_start</code> | This macro initializes the argument pointer variable <code>ap</code> to point to the first of the optional arguments of the current function; <code>last-required</code> must be the last required argument to the function                     |
| <code>va_arg</code>   | The <code>va_arg</code> macro returns the value of the next optional argument, and modifies the value of <code>ap</code> to point to the subsequent argument. Thus, successive uses of <code>va_arg</code> return successive optional arguments |
| <code>va_end</code>   | This ends the use of <code>ap</code>                                                                                                                                                                                                            |

# Advanced C

## Functions - Variadic - Example

### 011\_example.c

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

int main()
{
    int ret;

    ret = add(3, 2, 4, 4);
    printf("Sum is %d\n", ret);

    ret = add(5, 3, 3, 4, 5, 10);
    printf("Sum is %d\n", ret);

    return 0;
}
```

```
int add(int count, ...)
{
    va_list ap;
    int iter, sum;

    /* Initialize the arg list */
    va_start(ap, count);

    sum = 0;
    for (iter = 0; iter < count; iter++)
    {
        /* Extract args */
        sum += va_arg(ap, int);
    }

    /* Cleanup */
    va_end(ap);

    return sum;
}
```

# 2D Arrays

- 2 dimensional array is a collection of rows and columns, which is declared as follows

## Syntax:

```
datatype array_name[rows][columns]
```

## How to calculate the total memory?

Number of elements = rows \* columns

Total bytes = number of elements \* sizeof(datatype)

Or

Total bytes = rows \* columns \* sizeof(datatype)

E.g.,

```
int arr[2][3] = {1,2,3,4,5,6};
```

Or

```
int arr[2][3] = { {1,2,3}, {4,5,6} };
```

$$\text{Total bytes} = 2 * 3 * \text{sizeof(int)}$$

$$= 2 * 3 * 4$$

$$= 24 \text{ bytes}$$

## Memory layout:

| [0][0]<br>R0-C0 | [0][1]<br>R0-C1 | [0][2]<br>R0-C2 | [1][0]<br>R1-C0 | [1][1]<br>R1-C1 | [1][2]<br>R1-C2 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1<br>1000       | 2<br>1004       | 3<br>1008       | 4<br>1012       | 5<br>1016       | 6<br>1020       |

## How to read and print the values of a 2D array?

→ 2D array elements will be accessed by using array\_name[row][column]

E.g., arr[0][0] → 1      arr[1][0] → 4  
arr[0][1] → 2      arr[1][1] → 5  
arr[0][2] → 3      arr[1][2] → 6

## Printing array elements using loop:

- Because of 2 dimension 2d array elements will be fetched by using nested for loop
- Outer loop is for rows and inner loop is to track the columns of each row

```
int arr[2][3] = {1,2,3,4,5,6};
```

```
int i,j;
for(i = 0; i < 2 ; i++)
{
    for(j = 0; j < 3; j++)
    {
        printf("%d\n",arr[i][j]);
    }
}
```

### Reading array elements through user:

```
int arr[2][3];
int i,j;

for(i = 0; i < 2 ; i++)
{
    for(j = 0; j < 3; j++)
    {
        scanf("%d\n",&arr[i][j]);
    }
}
```

### Interpretation of 2d array:

arr[i][j]

- Replace a[i] with x  
 $arr[i][j] \Rightarrow x[j]$
- From 1d array we know that  $x[j]$  can be interpreted as follow:  
$$x[j] = *(x + j)$$
$$= *(x + j * \text{sizeof(datatype)})$$
- So, 2d array can be interpreted as,  
$$*(arr[i] + j)$$
- Further it can interpreted as,  
$$*(arr[i] + j) \Rightarrow *(*(arr + i) + j)$$
- At last 2d array interpretation looks like below  
$$*(*(arr + i * \text{sizeof(1D)}) + j * \text{sizeof(datatype\_array)})$$

|       | 1000 | 1004 | 1008 |          |
|-------|------|------|------|----------|
| Row 0 | 1    | 2    | 3    | 1d array |
| Row 1 | 4    | 5    | 5    | 1d array |
|       | 1012 | 1016 | 1020 |          |

- With the above image, as a summary we can define 2D array as a combination of several 1D array
- It can be said that the base address of 1st row is 1000 and base address of 2nd row is 1012

E.g.,

Consider  $i = 1, j = 1$  and base address of array is 1000

$$\begin{aligned}
 & \text{Arr}[1][1] \\
 & = *(\text{arr}[1] + 1) \\
 & = *(\text{arr}[1] + 1 * \text{sizeof}(\text{int})) \\
 & = *(\text{arr}[1] + 1 * 4) = *(\text{arr}[1] + 4) \\
 & = *(*(\text{arr} + 1 * \text{sizeof}(\text{int})) + 4) \\
 & = *(*(\text{arr} + 1 * 12) + 4) \\
 & = *(*(\text{arr} + 12) + 4) \\
 & = *(1012 + 4) \\
 & = *(1012+4) \\
 & = *1016 \Rightarrow 5
 \end{aligned}$$

With the above interpretation it is clear that 2d array can be interpreted in the following ways

- $\text{arr}[i][j]$
- $*(\text{arr}[i] + j)$
- $*(*(\text{arr} + i) + j)$
- $(*(\text{arr} + i))ij$

## Array of Pointers

→ array of pointers is a collection of address

→ syntax:

**datatype \*pointer\_name[size];**

→ e.g.,  $\text{int } * \text{ptr}[3];$

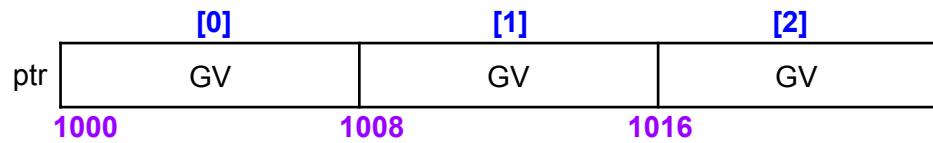
- Meaning of this declaration is  $\text{ptr}$  is a pointer which is capable of holding reference of 3 variable / memory location
- Total memory will be dependant on the bitness of system
  - If 32-bit system  

$$\begin{aligned}
 \text{Total memory} &= \text{size} * \text{sizeof(pointer)} \\
 &= 3 * 4 \\
 &= 12 \text{ bytes}
 \end{aligned}$$
  - If 64-bit system,  
 $\text{Total Memory} = 3 * 8 = 24 \text{ bytes}$
- Memory layout: 32-bit

|     | [0] | [1] | [2] |
|-----|-----|-----|-----|
| ptr | GV  | GV  | GV  |

1000                    1004                    1008

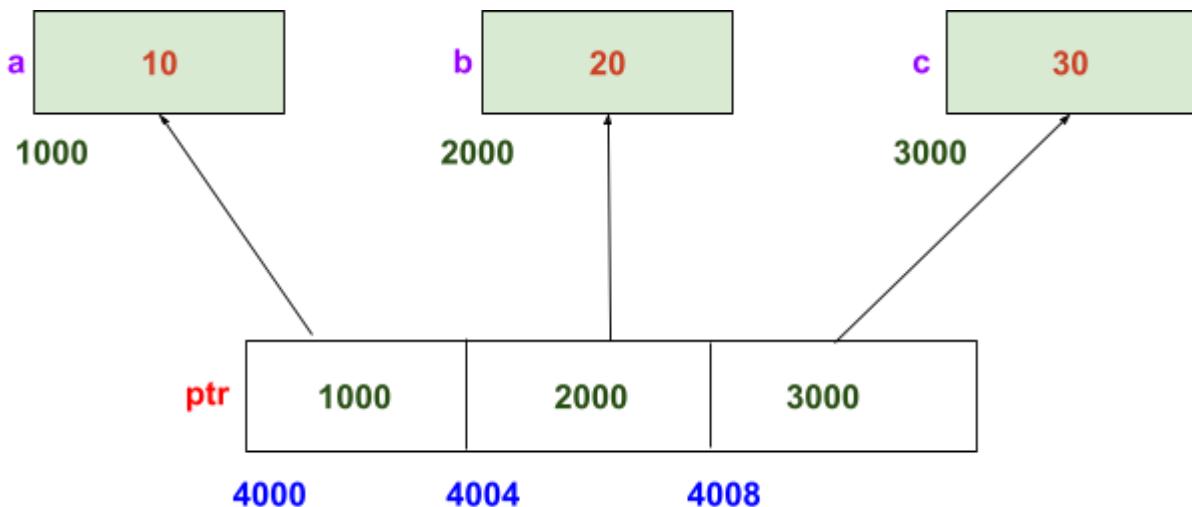
In 64-bit



Initialising and accessing array of pointers elements

```
int a = 10, b = 30, c = 40;  
int *ptr[3];
```

```
ptr[0] = &a;  
ptr[1] = &b;  
ptr[2] = &c;  
OR  
int *ptr = {&a, &b, &c};
```



### Accessing array of pointers elements:

```
*ptr[0] = *(ptr+0)  
=*(4000 + 0 * sizeof(pointer))  
=*(4000 + 0)  
=*1000  
= 10
```

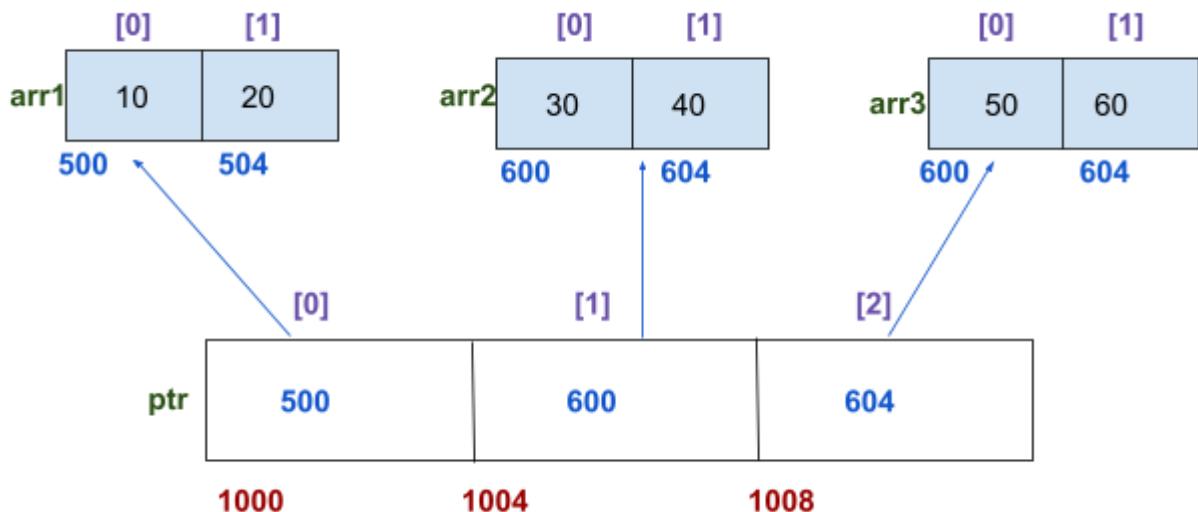
```
*ptr[1] = *(ptr+1)  
= *(4000 + 1 * 4)  
=*(4004)  
=*2000  
=20
```

```
*ptr[2] = *(ptr+2)  
= *(4000 + 2 * 4)
```

```
=*(4008)
=3000 = 30
```

→ array of pointers can be used to hold the address of 2 or more arrays  
E.g.,

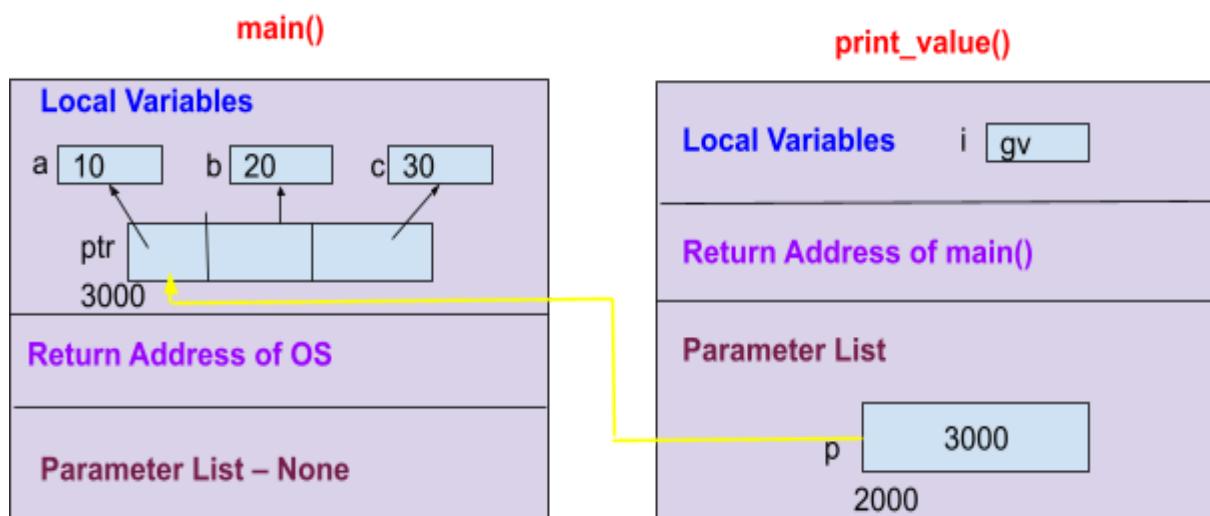
```
int arr1[2] = {10,20};
int arr2[2] = {30,40};
int arr3[2] = {50,60};
Int *ptr[3] = {arr1,arr2,arr3};
```



### Passing array of pointer to function:

```
int main()
{
    int a=10,b=20,c=30;
    int *ptr[3] = {&a, &b, &c};
    print_value(ptr);
}
```

```
void print_value(int **p)
{
    int i;
    for(i = 0; i < 3 ; i++)
        printf("%d\n",*ptr[i]);
}
```



## Array of strings

→ Array of string is a collection of strings, which is a 2 dimensional array

→ the first dimension says how many strings there are and the second dimension says about the maximum length of each string.

E.g., `char s[3][8] = {"Array", "Of", "Strings"};`

In the above example 3 is the number of strings and 8 is the length of each string.

## Interpretation of array of strings:

Consider base address of s is 1000,

$$\begin{aligned}s[0] &=> *(s+0) \\ &= *(1000 + 0 * \text{sizeof}(1D)) \\ &= *(1000 + 0 * 8) \\ &= *(1000) \\ &= 1000\end{aligned}$$

$$\begin{aligned}s[1] &=> *(s+1) \\ &= *(1000 + 1 * 8) \\ &= *(1008) \\ &= 1008\end{aligned}$$

$$\begin{aligned}s[2] &= *(s + 2) \\ &= *(1000 + 2 * 8) \\ &= *(1016) \\ &= 1016\end{aligned}$$

## Pointer to an array (explicitly used in 2d arrays)

- Pointer to an array is a pointer which holds the whole address of an array  
E.g., `int (*ptr)[3];`
- Above example, ptr is a pointer which is pointing to array of 3 integer elements
- Pointer arithmetic on pointer to an array will be,  
 $\text{ptr} + 1 = \text{ptr} + 1 * \text{sizeof}(1D \text{ array})$
- $\text{sizeof}(*\text{ptr}) = 3 * \text{sizeof}(\text{datatype})$

## Passing 2D array to function

1. The way array is declared

→ We can pass 2D array as a way its declared like,

`void print_array(int arr[2][3]);`

2. Pointer to an array

→ Next way of passing 2d array to function is by using pointer to an array

`void print_array(int (*ptr)[3]);`

3. Array of pointer

→ 2D arrays are also passed by using array of pointers

`void print_array(int *ptr[]);`

4. By passing size along with array address
  - One of the recommended way of passing 2d array along with number of rows and Columns
  - void print\_array(int row, int col, int arr[row][col]);**
  - The order of the arguments should be in the above order else it will be an error if the arguments are like below:
  - void print\_array(int arr[row][col],int row,int col); //compile time error**

5. By normal integer pointer

```
void print_array(int row, int col, int *ptr);
```

- one of complex way of passing 2d array is using normal pointer
- To access the 2d array in this method need to use pointer arithmetic
- accessing array element
- $*((p+i+number\_of\_columns) + j)$

E.g., i = 1, j = 1, columns = 3, base address = 1000

```
*((1000+1*3*sizeof(int)) + 1)
*((1012)+1)
*(1012 + 1 * sizeof(int))
*(1012+4)
*1016
```

## 2D array Creations:

### 1. Both Static

- In this method both rows and columns are fixed.
- This kind of array is also known as Rectangular array
- E.g., **int arr[2][3] = {10, 20, 30, 40, 50, 60};**

### 2. First Static Second Dynamic(FSSD)

- Here, the number of rows will be fixed but columns will be variable.
  - To create this type of array, will make use of array of pointers
  - For example consider the number of rows are 2,
- ```
int *ptr[2]; //rows
```

To create the columns for each row we need to use dynamic memory allocation method,

Read col value from user

```
for(i = 0; i < 2;i++)
{
    ptr[i] = malloc(col * sizeof(int));
}
```

### 3. First Dynamic Second Static(FDSS)

- Number of rows are variable but columns are fixed.
- To create such kind of array will use pointer to an array concept

- For example, consider u need 3 columns for each row, so  
`int (*ptr)[3]; //number of columns`

Create rows using dynamic memory allocation  
//read value for row from user  
`ptr = malloc(sizeof(*ptr) * row)`

#### 4. Both Dynamic

- Last way of creating 2d array is both rows and columns are dynamic
- It will achieved by using 2d level pointer that is `**ptr`.

`int **ptr;`

- First create number of rows like,  
`ptr = malloc(row * sizeof(int));`

- Then create columns for each row,

```
for(i = 0; i < row ; i++)
{
    ptr[i] = malloc(col * sizeof(int));
}
```

### Constant (keyword const)

- const is a keyword applied on a variable
- This is to tell the compiler that the value will not be changed throughout the program.

Eg1: int main()

```
{  
    const int num = 10;  
    num = 20; //error  
    return 0;  
}
```

Both const int num and int const num are the same. Read it as num is an integer constant.

Eg2: int main()

```
{  
    int num = 10;  
    const int *ptr = &num;  
    *ptr = 10 //error  
    (*ptr)++; //error  
    return 0;  
}
```

Both const int \*ptr and int const \*ptr are the same. Ptr is the pointer to a constant integer. Changing \*ptr is not allowed.

Eg3: int main()

```
{  
    int num1 = 10, num2 = 20;  
    int * const ptr = &num1;  
    ptr = &num2; //error  
    ptr++; //error  
    return 0;  
}
```

ptr is a constant pointer to an integer. Changing ptr is not allowed.

Eg4: int main()

```
{  
    int num1 = 10, num2 = 20;  
    const int *const ptr = &num1;
```

```

ptr++; //error
(*ptr)++; //error
ptr = &num2; //error
*ptr = 20 ; //error
return 0;
}

```

ptr is a constant pointer to a constant integer. Changing \*ptr and ptr are not allowed.

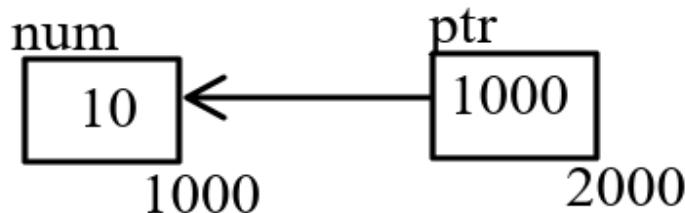
### Dos and Don'ts of pointers

- Eg: int main()

```

{
    int num = 10;
    int *ptr = &num;
    return 0;
}

```



- Adding, subtracting the pointer with a constant is allowed.  
`ptr = ptr + 1; // ptr + 1 * sizeof(int)`  
`ptr = ptr - 1; // ptr - 1 * sizeof(int)`
- Multiplying and dividing the pointer with a constant is an error  
`ptr = ptr * 1; //error`  
`ptr = ptr / 1; //error`
- Subtracting a pointer with a pointer is allowed.  
`ptr = ptr - ptr;`
- Adding, multiplying and dividing a pointer with another pointer is an error.  
`ptr = ptr + ptr; //error`  
`ptr = ptr * ptr; //error`  
`ptr = ptr / ptr; //error`
- Bitwise operators are not allowed

- Logical operators are allowed

## Pitfalls of Pointers

### 1. Segmentation fault:

This is caused whenever we are accessing the address that is not allowed to be accessed.

Eg: int main()

```

{
    int *ptr = NULL;
    printf("%d\n", *ptr); // seg fault as accessing address 0
is not allowed
    return 0;
}

```

### 2. Dangling pointer:

Pointer pointing to a freed location is called a dangling pointer.

Eg: int main()

```

{
    int *ptr = calloc(5, 1);
    free(ptr);
    printf("%d\n", *ptr); //ptr is a dangling pointer
    return 0;
}

```

ptr is still having the address which it was pointing to. But the address is no longer with the user as it has already been freed.

Dereferencing the dangling pointer leads to undefined behaviour.

### 3. Wild pointer:

Uninitialized pointers which are pointing to some random address are called wild pointers. Dereferencing wild pointers also leads to undefined behaviour. Good practice is to initialise it with NULL when the pointer is being declared.

Eg: int main()

```
{
}
```

```
int *ptr; //wild pointer
static int *ptr1; //not a wild pointer because static
pointers are initialised with NULL by default.
return 0;
}
```

#### 4. Memory leak:

Failing to free the dynamically allocated memory leads to memory leak.

Eg: int main()  
{

```
int *ptr;
while(1)
{
    ptr = malloc(100);
    //forgot to free the memory
}
return 0;
}
```

#### 5. Bus error:

This error is caused when the CPU cannot handle the address in the address bus.

Eg: int main()  
{

```
char array[4];
int *ptr = &array[1];
scanf("%d\n", ptr); //bus error
return 0;
}
```

Usually the CPU can handle the address which is a multiple of 4. Trying to read an integer to the address which is not a multiple of 4 leads to bus error.

# Dynamic Memory Allocation

## Static Memory

- So far whatever memory was used is static memory which will be either in stack, data or code segment.
- In static memory the size is fixed, cannot modify, exchange or delete the memory.
- Static memories are called as named location

## Dynamic Memory

- Memory will be allocated in heap.
- Dynamic memory can be modified, extended or deleted whenever it is required.
- Dynamic memory is managed with the help of pointers and functions like malloc, calloc, realloc and free.
- Because dynamic memory is an unnamed location, to control them precisely we need pointers.
- All the functions are part of stdlib.h

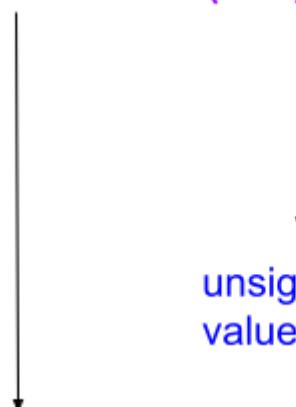
Example:

```
int main()
{
    //dynamic memory allocation 100 byte
    // some operation
    //free the memory
}
```

## 1. Malloc()

- Prototype of the function - void \* malloc(size\_t size)

**void \*malloc(size\_t size);**

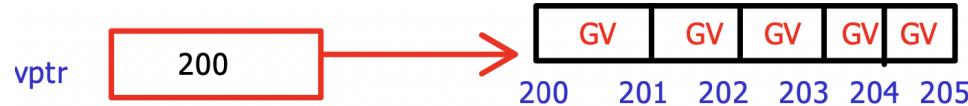


unsigned int or unsigned long int  
value in bytes

```

void *vptr;
vptr = malloc(5);

```



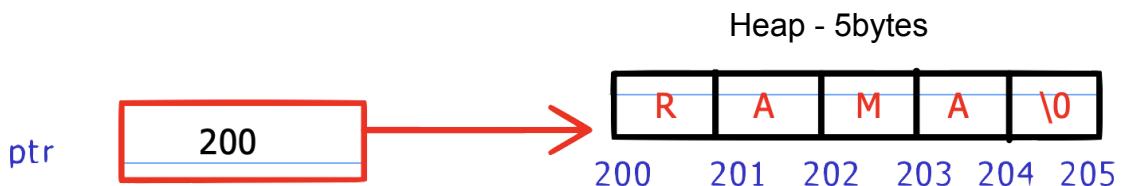
- Malloc will always search for 5 bytes of contiguous memory in heap segment
- If continuous memory bytes are not available then malloc will return NULL
- By default it will be having garbage value

Example usage:

```

int *ptr;
ptr = malloc(5); //implicit type casting from void * to int *
strcpy(ptr, "RAMA");

```

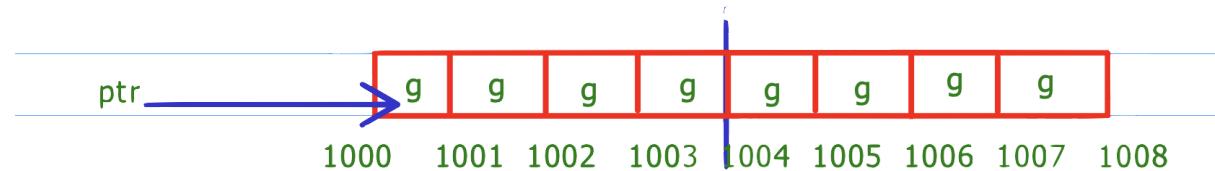



---

```

ptr = malloc(2 * sizeof(int));

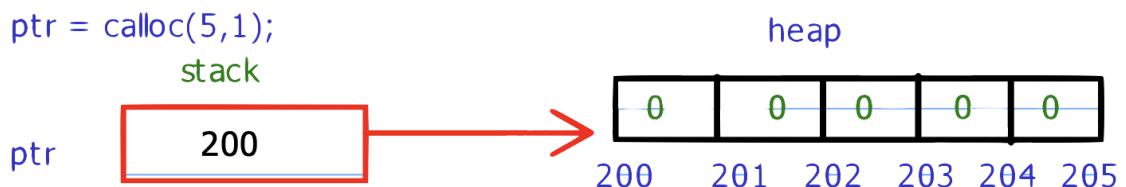
```



## 2. Calloc()

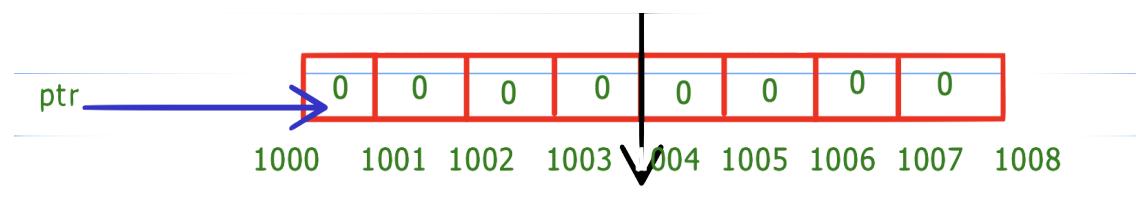
- Prototype of calloc - `void *calloc(size_t nmemb, size_t size);`  
Where, nmemb - number of member or number of elements or number of blocks  
Size - size of each member

Example,  
int \*ptr;



- Calloc will always search for 5 bytes of contiguous memory in heap segment
- If continuous memory bytes are not available then malloc will return NULL
- By default it will be initialised with 0

Ptr = calloc(2,sizeof(int));



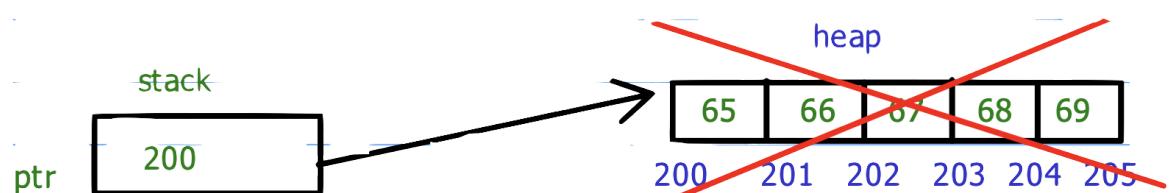
## Malloc vs calloc

- With respect to space the same works in a similar way, where both the function allocates contiguous memory in heap.
- When it comes to time calloc takes more time than malloc because of initialization of each block to 0
- Malloc is better with structure and basic memory allocation
- Calloc is better with array because of initialisation to 0

## 3. Free()

- Free function is used whenever user wants to free/ deallocate unused memory
- free is used to deallocate the dynamically allocated memory by malloc or calloc function
- Prototype -

void free(void \*ptr);

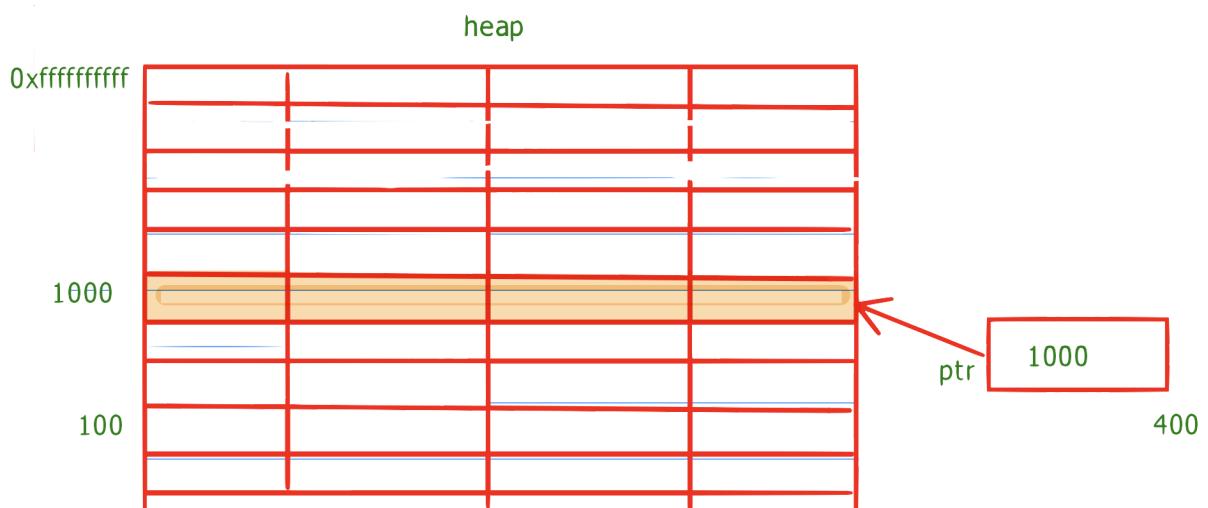


- although heap is deleted from free function, the address which is freed will be available in the pointer
- Pointer which still holds onto the already freed memory is known as a dangling pointer. It is recommended to make dangling pointer to point to Null to avoid undefined behaviour at later

#### 4. Realloc function

- Realloc is used to extend or shrink the previously allocated memory whenever required
- Prototype :
  - `void *realloc(void *ptr, size_t size);`

Consider `void *ptr = malloc(4);`



1. `realloc(ptr,3)` //shrinking the memory
  - a. Here, 1 byte will be freed from the pointer.
  - b. Legally 3 bytes can be accessed.
2. `realloc(ptr,10);`
  - a. When the size is more than the previous allocation then the realloc will extend the remaining bytes
  - b. Here, 7 bytes will be extended.
  - c. If 7 bytes are available continuously in the same location then realloc extends the memory and returns the same address.
  - d. Else if 7 bytes is not available in the same location then realloc search for whole 10 bytes in some other location and returns the newly allocated memory.
  - e. If realloc() fails to expand memory as requested then **it returns NULL , the data in the old memory remains unaffected.**

# Preprocessing



# Advanced C

## Preprocessor



- One of the step performed before compilation
- Is a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- Instructions given to preprocessor are called preprocessor directives and they begin with “#” symbol
- Few advantages of using preprocessor directives would be,
  - Easy Development
  - Readability
  - Portability



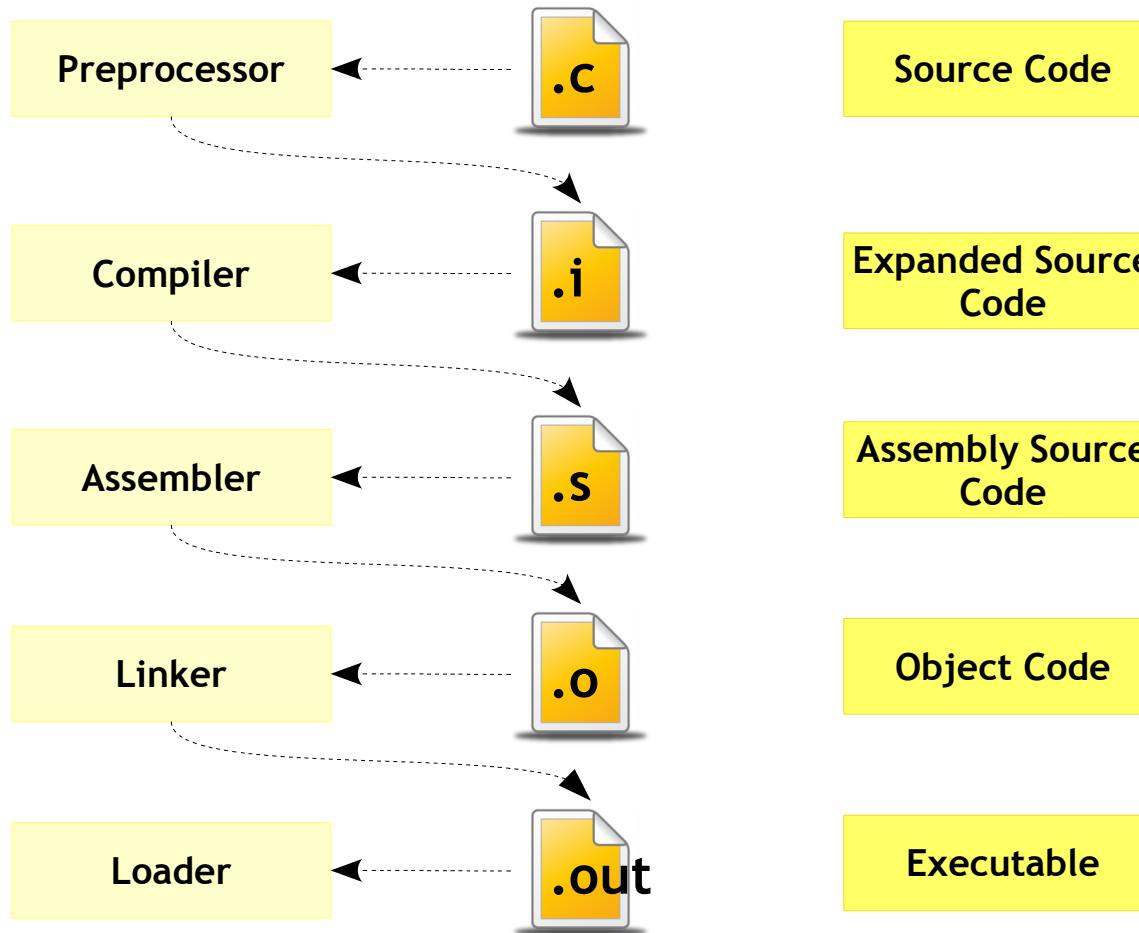
# Advanced C

## Preprocessor - Compilation Stages

- Before we proceed with preprocessor directive let's try to understand the stages involved in compilation
- Some major steps involved in compilation are
  - Preprocessing (Textual replacement)
  - Compilation (Syntax and Semantic rules checking)
  - Assembly (Generate object file(s))
  - Linking (Resolve linkages)
- The next slide provide the flow of these stages

# Advanced C

## Preprocessor - Compilation Stages



```
user@user:~] gcc -E file.c
```

```
user@user:~] gcc -S file.c
```

```
user@user:~] gcc -c file.c
```

```
user@user:~] gcc file.c -o file.out
```

```
user@user:~]gcc -save-temp file.c #would generate all intermediate files
```

# Advanced C

## Preprocessor - Compilation Steps



```
user@user:~] cpp file.c -o file.i
```



```
user@user:~] cc -S file.i -o file.s
```



```
user@user:~] as file.s -o file.o
```



Bit complex step

```
user@user:~] ld file.o -o file.out <LIBRARY PATH>
```



```
user@user:~]./file.out
```

# Advanced C

## Preprocessor - Compilation Steps



```
user@user:~] gcc -E file.c -o file.i
```



```
user@user:~] gcc -S file.i -o file.s
```



```
user@user:~] gcc -c file.s -o file.o
```



```
user@user:~] gcc file.o -o file.out
```



```
user@user:~]./file.out
```

# Advanced C

## Preprocessor - Directives

```
#include      #error
#define       #warning
#undef        #line
#endif       #pragma
#ifndef      #
#if          ##
#elif
#else
#endif
```

# Advanced C

## Preprocessor - Header Files

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- Has to be included using C preprocessing directive '**#include**'
- Header files serve two purposes.
  - Declare the interfaces to parts of the operating system by supplying the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program.

# Advanced C

## Preprocessor - Header Files vs Source Files



VS



- Declarations
- Sharable/reusable
  - #defines
  - Datatypes
- Used by more than 1 file

- Function and variable definitions
- Non sharable/reusable
  - #defines
  - Datatypes



# Advanced C

## Preprocessor - Header Files - Syntax



### Syntax

```
#include <file.h>
```

- System header files
- It searches for a file named *file* in a standard list of system directories

### Syntax

```
#include "file.h"
```

- Local (your) header files
- It searches for a file named *file* first in the directory containing the current file, then in the quote directories and then the same directories used for <file>



# Advanced C

## Preprocessor - Header Files - Operation

002\_file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

003\_file2.h

```
char *test(void);
```

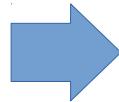
001\_file1.c

```
int num;

#include "003_file2.h"

int main()
{
    puts(test());

    return 0;
}
```



```
int num;
```

```
char *test(void);

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E file1.c file2.c # You may add -P option too!!
```

# Advanced C

## Preprocessor - Header Files - Search Path

002\_file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

003\_file2.h

```
char *test(void);
```

001\_file1.c

```
int num;

#include "file2.h"

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E file1.c file2.c
```

# Advanced C

## Preprocessor - Header Files - Search Path

002\_file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

003\_file2.h

```
char *test(void);
```

001\_file1.c

```
int num;

#include <file2.h>

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E file1.c file2.c -l .
```

# Advanced C

## Preprocessor - Header Files - Search Path

- On a normal Unix system GCC by default will look for headers requested with #include <file> in:
  - /usr/local/include
  - libdir/gcc/target/version/include
  - /usr/target/include
  - /usr/include
- You can add to this list with the -I <dir> command-line option

Get it as

```
user@user:~] cpp -v /dev/null -o /dev/null #would show search the path info
```

# Advanced C

## Preprocessor - Macro - Object-Like



- An object-like macro is a simple identifier which will be replaced by a code fragment
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants

### Syntax

```
#define SYMBOLIC_NAME      CONSTANTS
```

### Example

```
#define BUFFER_SIZE      1024
```



# Advanced C

## Preprocessor - Macro - Object-Like

### 004\_example.c

```
#define SIZE      1024
#define MSG       "Enter a string"

int main()
{
    char array[SIZE];

    printf("%s\n", MSG);
    fgets(array, SIZE, stdin);

    printf("%s\n", array);

    return 0;
}
```

Compile as

```
user@user:~] gcc -E 004_example.c -o 004_example.i
```

### 004\_example.i

```
# 1 "main.c"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "main.c"

int main()
{
    char array[1024];

    printf("%s\n", "Enter a string");
    fgets(array, 1024, stdin);

    printf("%s\n", array);

    return 0;
}
```

# Advanced C

## Preprocessor - Macro - Standard Predefined

- Several object-like macros are predefined; you use them without supplying their definitions.
- Standard are specified by the relevant language standards, so they are available with all compilers that implement those standards

### 005\_example.c

```
#include <stdio.h>

int main()
{
    printf("Program: \"%s\" ", __FILE__);
    printf("was compiled on %s at %s. ", __DATE__, __TIME__);
    printf("This print is from Function: \"%s\"", __func__);
    printf("at line %d\n", __LINE__);

    return 0;
}
```

# Advanced C

## Preprocessor - Macro - Arguments



- Function-like macros can take arguments, just like true functions
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like

### Syntax

```
#define MACRO (ARGUMENT (S) )
```

(EXPRESSION WITH ARGUMENT (S) )

# Advanced C

## Preprocessor - Macro - Arguments



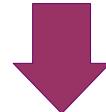
### 006\_example.c

```
#include <stdio.h>

#define SET_BIT(num, pos)      num | (1 << pos)

int main()
{
    printf("%d\n", 2 * SET_BIT(0, 2));

    return 0;
}
```



### 006\_example.i

```
int main()
{
    printf("%d\n", 2 * 0 | (1 << 2));

    return 0;
}
```

# Advanced C

## Preprocessor - Macro - Arguments



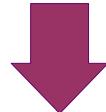
### 007\_example.c

```
#include <stdio.h>

#define SET_BIT(num, pos)      (num | (1 << pos))

int main()
{
    printf("%d\n", 2 * SET_BIT(0, 2));

    return 0;
}
```



### 007\_example.i

```
int main()
{
    printf("%d\n", 2 * (0 | (1 << 2)));
}

return 0;
}
```

# Advanced C

Preprocessor - Macro - Arguments - DIY

- WAM to find the sum of two nos
- Write macros to get, set and clear N<sup>th</sup> bit in an integer
- WAM to swap a nibble in a byte

# Advanced C

## Preprocessor - Macro - Multiple Lines

- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
- This could be done to achieve readability
- When the macro is expanded, however, it will all come out on one line

# Advanced C

## Preprocessor - Macro - Multiple Lines



### 008\_example.c

```
#include <stdio.h>

#define SWAP(a, b)
    int temp = a;
    a = b;
    b = temp;

int main()
{
    int n1 = 10, n2= 20;

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    return 0;
}
```

/\



### 008\_example.i

```
int main()
{
    int n1 = 10, n2= 20;

    int temp = n1;n1 = n2;n2 = temp;
    printf("%d %d\n", n1, n2);

    int temp = n1;n1 = n2;n2 = temp;
    printf("%d %d\n", n1, n2);

    return 0;
}
```



# Advanced C

## Preprocessor - Macro - Multiple Lines

009\_example.c

```
#include <stdio.h>

#define SWAP(a, b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int n1 = 10, n2= 20;

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    SWAP(n1, n2);
    printf("%d %d\n", n1, n2);

    return 0;
}
```

009\_example.i

```
int main()
{
    int n1 = 10, n2= 20;

    {int temp = n1;n1 = n2;n2 = temp;}
    printf("%d %d\n", n1, n2);

    {int temp = n1;n1 = n2;n2 = temp;}
    printf("%d %d\n", n1, n2);

    return 0;
}
```

# Advanced C

Preprocessor - Macro - Multiple Lines - DIY

- WAM to swap any two numbers of basic type using temporary variable

# Advanced C

## Preprocessor - Macro vs Function

### Function

```
#include <stdio.h>

int set_bt(int n, int p)
{
    return (n | (1 << p));
}

int main()
{
    printf("%d\n", 2 * set_bt(0, 2));
    printf("%d\n", 4 * set_bt(0, 2));

    return 0;
}
```

- Context switching overhead
- Stack frame creation overhead
- Space optimized on repeated call
- Compiled at compile stage, invoked at run time
- Type sensitive
- Recommended for larger operation

### Macro

```
#include <stdio.h>

#define set_bt(n, p)      (n | (1 << p))

int main()
{
    printf("%d\n", 2 * set_bt(0, 2));
    printf("%d\n", 4 * set_bt(0, 2));

    return 0;
}
```

- No context switching overhead
- No stack frame creation overhead
- Time optimized on repeated call
- Preprocessed and expanded at preprocessing stage
- Type insensitive
- Recommended for smaller operation

# Advanced C

## Preprocessor - Macro - Stringification

### 010\_example.c

```
#include <stdio.h>

#define WARN_IF(EXP)
do
{
    x--;
    if (EXP)
    {
        fprintf(stderr, "Warning: " #EXP "\n");
    }
} while (x);

int main()
{
    int x = 5;

    WARN_IF(x == 0);

    return 0;
}
```

- You can convert a macro argument into a string constant by adding #

# Advanced C

## Preprocessor - Conditional Compilation

- A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler
- Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator
- A conditional in the C preprocessor resembles in some ways an if statement in C with the only difference being it happens in compile time
- Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

# Advanced C

## Preprocessor - Conditional Compilation

- There are three general reasons to use a conditional.
  - A program may need to use different code depending on the machine or operating system it is to run on
  - You may want to be able to compile the same source file into two different programs, like one for debug and other as final
  - A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference

# Advanced C

## Preprocessor - Header Files - Once-Only

- If a header file happens to be included twice, the compiler will process its contents twice causing an error
- E.g. when the compiler sees the same structure definition twice
- This can be avoided like

### Syntax

```
#ifndef NAME
#define NAME

/* The entire file is protected */

#endif
```

# Advanced C

## Preprocessor - Header Files - Once-Only



### 011\_example.c

```
#include "012_example.h"
#include "012_example.h"

int main()
{
    struct UserInfo p = {420, "Tingu"};

    return 0;
}
```

- Note that, **011\_exampe.h** is included 2 times which would lead to redefinition of the structure **UserInfo**

### 012\_example.h

```
struct UserInfo
{
    int id;
    char name[30];
};
```

# Advanced C

## Preprocessor - Header Files - Once-Only



### 013\_example.c

```
#include "014_example.h"
#include "014_example.h"

int main()
{
    struct UserInfo p = {420, "Tingu"};

    return 0;
}
```

- The multiple inclusion is protected by the **#ifndef** preprocessor directive

### 014\_example.h

```
#ifndef EXAMPLE_014_H
#define EXAMPLE_014_H

struct UserInfo
{
    int id;
    char name[30];
};

#endif
```

# Advanced C

## Preprocessor - Header Files - Once-Only

### 015\_example.c

```
#include "016_example.h"
#include "016_example.h"

int main()
{
    struct UserInfo p = {420, "Tingu"};

    return 0;
}
```

- The other way to do this would be **#pragma once** directive
- This is not portable

### 016\_example.h

```
#pragma once

struct UserInfo
{
    int id;
    char name[30];
};
```

# Advanced C

## Preprocessor - Conditional Compilation - #ifdef

### Syntax

```
#ifdef MACRO  
  
/* Controlled Text */  
  
#endif
```

### 017\_example.c

```
#include <stdio.h>  
  
#define METHOD1  
  
int main()  
{  
    #ifdef METHOD1  
        puts("Hello World");  
    #else  
        printf("Hello World");  
    #endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Conditional Compilation - #ifndef

### Syntax

```
#ifndef MACRO  
  
/* Controlled Text */  
  
#endif
```

### 018\_example.c

```
#include <stdio.h>  
  
#undef METHOD1  
  
int main()  
{  
    #ifndef METHOD1  
        puts("Hello World");  
    #else  
        printf("Hello World");  
    #endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Conditional Compilation - defined

### Syntax

```
#if defined condition  
/* Controlled Text */  
  
#endif
```

### 019\_example.c

```
#include <stdio.h>  
  
#define METHOD1  
  
int main()  
{  
    #if defined (METHOD1)  
        puts("Hello World");  
    #endif  
    #if defined (METHOD2)  
        printf("Hello World");  
    #endif  
    #if defined (METHOD1) && defined (METHOD2)  
        puts("Hello World");  
        printf("Hello World");  
    #endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Conditional Compilation - if

### Syntax

```
#if expression  
/* Controlled Text */  
  
#endif
```

### 020\_example.c

```
#include <stdio.h>  
  
#define METHOD 1  
  
int main()  
{  
    #if METHOD == 1  
        puts("Hello World");  
    #endif  
    #if METHOD == 2  
        printf("Hello World");  
    #endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Conditional Compilation - else

### Syntax

```
#if expression  
/* Controlled Text if true */  
  
#else  
  
/* Controlled Text if false */  
  
#endif
```

### 021\_example.c

```
#include <stdio.h>  
  
#define METHOD 0  
  
int main()  
{  
    #if METHOD == 1  
        puts("Hello World");  
    #else  
        printf("Hello World");  
    #endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Conditional Compilation - elif

### Syntax

```
#if expression1  
  
/* Controlled Text */  
  
#elif expression2  
  
/* Controlled Text */  
  
#else  
  
/* Controlled Text */  
  
#endif
```

### 022\_example.c

```
#include <stdio.h>  
  
#define METHOD 1  
  
int main()  
{  
    char msg[] = "Hello World";  
  
#if METHOD == 1  
    puts(msg);  
#elif METHOD == 2  
    printf("%s\n", msg);  
#else  
    int i;  
    for (i = 0; i < 12; i++)  
    {  
        putchar(msg[i]);  
    }  
#endif  
  
    return 0;  
}
```

# Advanced C

## Preprocessor - Cond... Com... - CL Option

### 023\_example.c

```
#include <stdio.h>

int main()
{
    int x = 10, y = 20;

#ifdef SPACE_OPTIMIZED
    x = x ^ y;
    y = x ^ y;
    x = x ^ y;
    printf("Selected Space Optimization\n");
#else
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("Selected Time Optimization\n");
#endif

    return 0;
}
```

Compile as

```
user@user:~] gcc main.c -D SPACE_OPTIMIZED
```

# Advanced C

Preprocessor - Cond... Com... - Deleted Code



## Syntax

```
#if 0

/* Deleted code while compiling */
/* Can be used for nested code comments */
/* Avoid for general comments */
/* Don't write lines like these!! with '

#endif
```



# Advanced C

## Preprocessor - Diagnostic

- The directive **#error** causes the preprocessor to report a fatal error. The tokens forming the rest of the line following **#error** are used as the error message
- The directive **#warning** is like **#error**, but causes the preprocessor to issue a warning and continue preprocessing. The tokens following **#warning** are used as the warning message

# Advanced C

## Preprocessor - Diagnostic - #warning

### 024\_example.c

```
#include <stdio.h>

#if defined DEBUG_PRINT
#warning "Debug print enabled"
#endif

int main()
{
    int sum, num1, num2;

    printf("Enter 2 numbers: ");
    scanf("%d %d", &num1, &num2);

#ifdef DEBUG_PRINT
    printf("The entered values are %d %d\n", num1, num2);
#endif

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

    return 0;
}
```

# Advanced C

## Preprocessor - Diagnostic - #error



### 025\_example.c

```
#include <stdio.h>

#if defined (STATIC) || defined (DYNAMIC)
#define SIZE      100
#else
#error "Memory not allocated!! Use -D STATIC or DYNAMIC while compiling"
#endif

int main()
{
#if defined STATIC
    char buffer[SIZE];
#elif defined DYNAMIC
    char *buffer = malloc(SIZE * sizeof(char));
#endif

#if defined (STATIC) || defined (DYNAMIC)
    fgets(buffer, SIZE, stdin);
    printf("%s\n", buffer);
#endif

    return 0;
}
```



# Advanced C

## Preprocessor - Diagnostic - #line

- Also known as preprocessor line control directive
- **#line** directive can be used to alter the line number and filename
- The line number will start from the set value, from the **#line** is encountered with the provided name

### 026\_example.c

```
#include <stdio.h>

int main()
{
    #line 100 "project tuntun"
    printf("This is from file %s at line %d \n", __FILE__, __LINE__);

    return 0;
}
```

# User Defined Datatypes



# Advanced C

## User Defined Datatypes (Composite Data Types)

- Sometimes it becomes tough to build a whole software that works only with integers, floating values, and characters.
- In circumstances such as these, you can create your own data types which are based on the standard ones
- There are some mechanisms for doing this in C:
  - Structures (derived)
  - Unions (derived)
  - Typedef (storage class)
  - Enums (user defined)
- Hoo!!, let's not forget our old friend \_r\_a\_ which is a user defined data type too!!.

# Advanced C

## User Defined Datatypes (Composite Data Types)



### : Composite (or Compound) Data Type :

- Any data type which can be constructed from primitive data types and other composite types
- It is sometimes called a structure or aggregate data type
- Primitives types - int, char, float, double

# Advanced C UDTs



# Advanced C

## UDTs - Structures



### Syntax

```
struct StructureName  
{  
    /* Group of data types */  
};
```

- If we consider the Student as an example, The admin should have at least some important data like name, ID and address.

- So if we create a structure of the above requirement, it would look like,

### Example

```
struct Student  
{  
    int id;  
    char name[20];  
    char address[60];  
};
```

# Advanced C

## UDTs - Structures - Declaration and definition

### 001\_example.c

```
struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    return 0;
}
```

- Name of the datatype. Note it's **struct Student** and not **Student**
- Are called as **fields** or **members** of the structure
- Declaration ends here
- The memory is not yet allocated!!
- **s1** is a **variable** of type **struct Student**
- The memory is allocated now

# Advanced C

## UDTs - Structures - Memory Layout



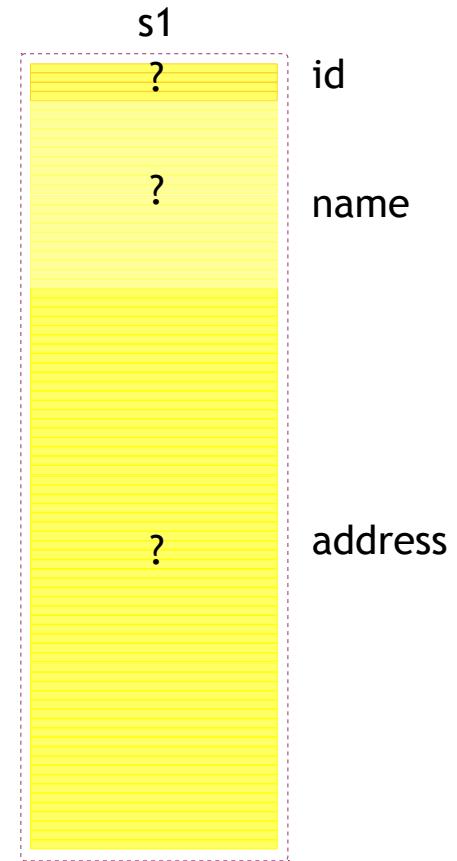
001\_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    return 0;
}
```



- What does `s1` contain?
- How can we draw it's memory layout?

# Advanced C

## UDTs - Structures - Memory Layout

### 002\_example.c

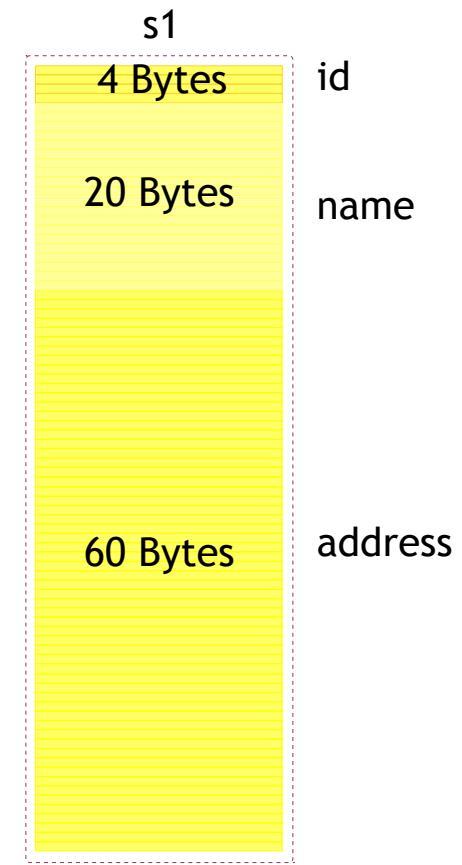
```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));
    printf("%zu\n", sizeof(s1));

    return 0;
}
```



Structure size depends in the member arrangement!! Will discuss that shortly

# Advanced C

## UDTs - Structures - Access

### 003\_example.c

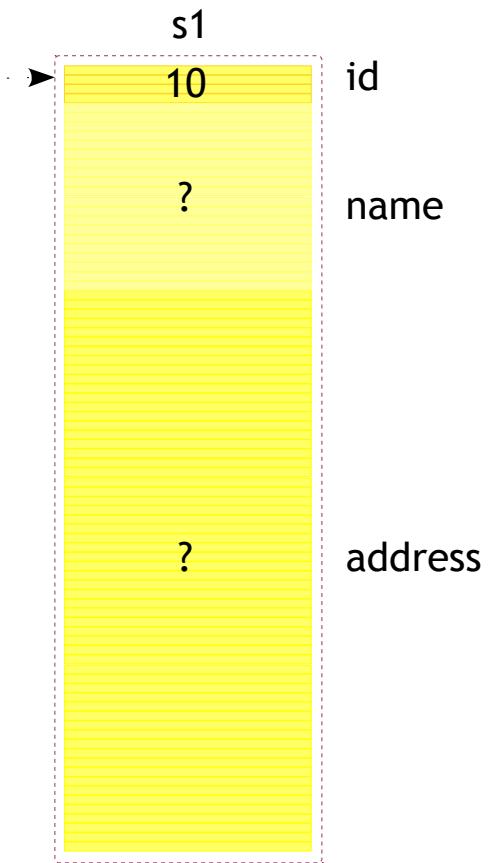
```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1;

    s1.id = 10;

    return 0;
}
```



- How to write into `id` now?
- It's by using “.” (Dot) operator (member access operator)
- Now please assign the `name` member of `s1`

# Advanced C

## UDTs - Structures - Initialization

### 004\_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    return 0;
}
```



# Advanced C

## UDTs - Structures - Copy

### 005\_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};
    struct Student s2;

    s2 = s1;

    return 0;
}
```



Structure name does not represent its address. (No correlation with arrays)

# Advanced C

## UDTs - Structures - Address

### 006\_example.c

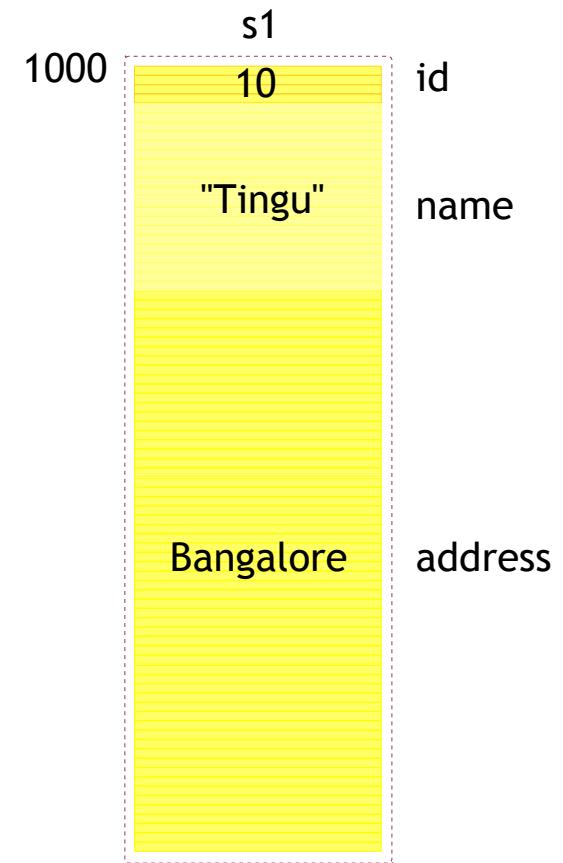
```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    printf("Struture starts at %p\n", &s1);
    printf("Member id is at %p\n", &s1.id);
    printf("Member name is at %p\n", s1.name);
    printf("Member address is at %p\n", s1.address);

    return 0;
}
```



# Advanced C

## UDTs - Structures - Pointers



- Pointers!!!. Not again ;). Fine don't worry, not a big deal
- But do you any idea how to create it?
- Will it be different from defining them like in other data types?

# Advanced C

## UDTs - Structures - Pointer

### 007\_example.c

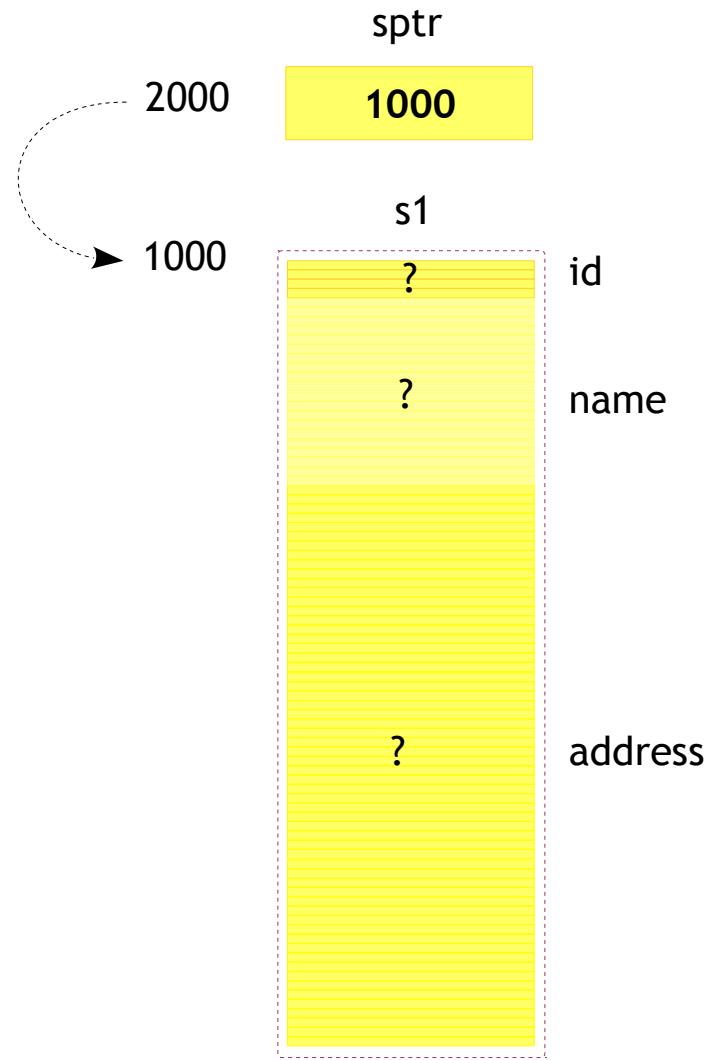
```
#include <stdio.h>

struct Student
{
    int id;
    char name[20];
    char address[60];
};

static struct Student s1;

int main()
{
    struct Student *sptr = &s1;

    return 0;
}
```



# Advanced C

## UDTs - Structures - Pointer - Access

### 008\_example.c

```
#include <stdio.h>

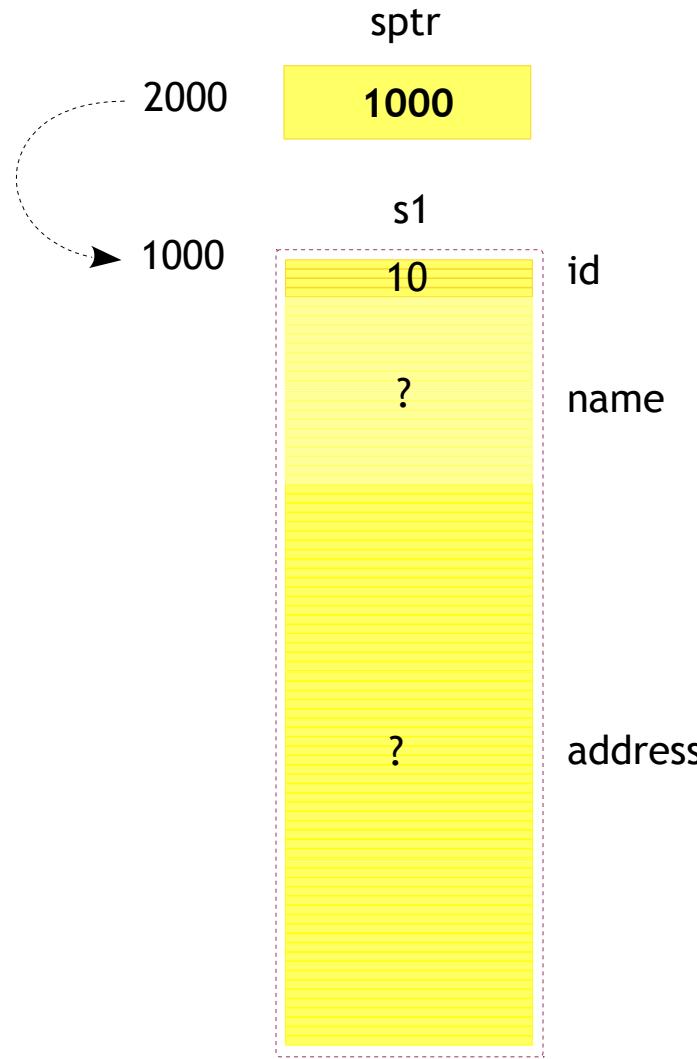
struct Student
{
    int id;
    char name[20];
    char address[60];
};

static struct Student s1;

int main()
{
    struct Student *sptr = &s1;

    (*sptr).id = 10;

    return 0;
}
```



# Advanced C

## UDTs - Structures - Pointer - Access - Arrow

### 009\_example.c

```
#include <stdio.h>

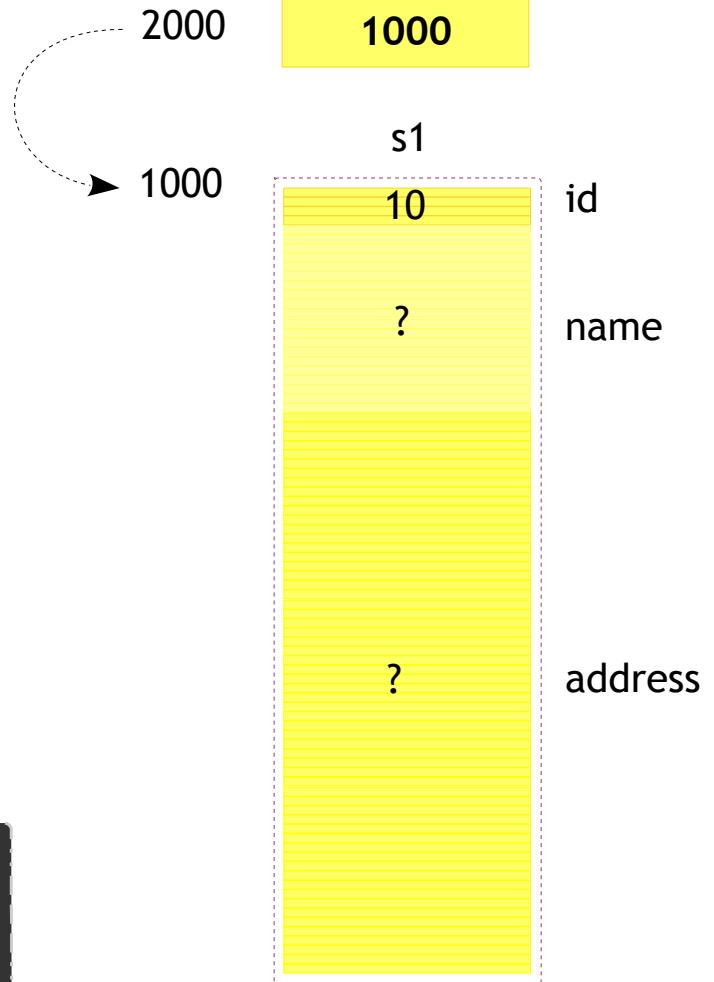
struct Student
{
    int id;
    char name[20];
    char address[60];
};

static struct Student s1;

int main()
{
    struct Student *sptr = &s1;

    sptr->id = 10;

    return 0;
}
```



Note: we can access the structure pointer as seen in the previous slide. The Arrow operator is just convenience and frequently used

# Advanced C

## UDTs - Structures - Functions



- The structures can be passed as parameter and can be returned from a function
- This happens just like normal datatypes.
- The parameter passing can have two methods again as normal
  - Pass by value
  - Pass by reference



# Advanced C

## UDTs - Structures - Functions - Pass by Value



### 010\_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student s)
{
    s.id = 10;
}

int main()
{
    struct Student s1;

    data(s1);

    return 0;
}
```

Not recommended on  
larger structures



# Advanced C

## UDTs - Structures - Functions - Pass by Reference

### 011\_example.c

```
#include <stdio.h>

struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student *s)
{
    s->id = 10;
}

int main()
{
    struct Student s1;

    data(&s1);

    return 0;
}
```

Recommended on  
larger structures

# Advanced C

## UDTs - Structures - Functions - Return

### 012\_example.c

```
struct Student
{
    int id;
    char *name;
    char *address;
};

struct Student data(void)
{
    struct Student s;

    s.name = (char *) malloc(30 * sizeof(char));
    s.address = (char *) malloc(150 * sizeof(char));

    return s;
}

int main()
{
    struct Student s1;

    s1 = data();

    return 0;
}
```

# Advanced C

## UDTs - Structures - Padding

- Adding of few extra useless bytes (in fact skip address) in between the address of the members are called structure padding.
- What!!?, wasting extra bytes!!, Why?
- This is done for Data Alignment.
- Now!, what is data alignment and why did this issue suddenly arise?
- No its is not sudden, it is something the compiler would be doing internally while allocating memory.
- So let's understand data alignment in next few slides

# Advanced C

## Data Alignment

- A way the data is arranged and accessed in computer memory.
- When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (4 bytes in 32 bit system) or larger.
- The main idea is to increase the efficiency of the CPU, while handling the data, by arranging at a memory address equal to some multiple of the word size
- So, Data alignment is an important issue for all programmers who directly use memory.

# Advanced C

## Data Alignment



- If you don't understand data and its address alignment issues in your software, the following scenarios, in increasing order of severity, are all possible:
  - Your software will run slower.
  - Your application will lock up.
  - Your operating system will crash.
  - Your software will silently fail, yielding incorrect results.

# Advanced C

## Data Alignment



### Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

0	ch
1	78
2	56
3	34
4	12
5	?
6	?
7	?

- Lets consider the code as given
- The memory allocation we expect would be like shown in figure
- So lets see how the CPU tries to access these data in next slides

# Advanced C

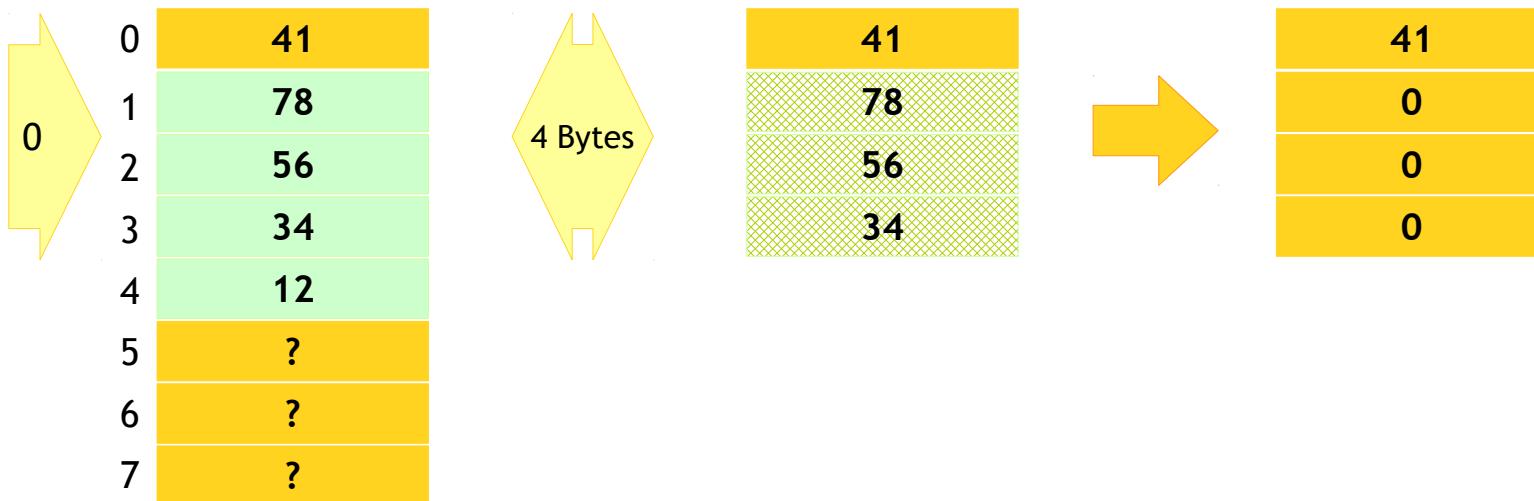
## Data Alignment



### Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching a character by the CPU will be like shown below



# Advanced C

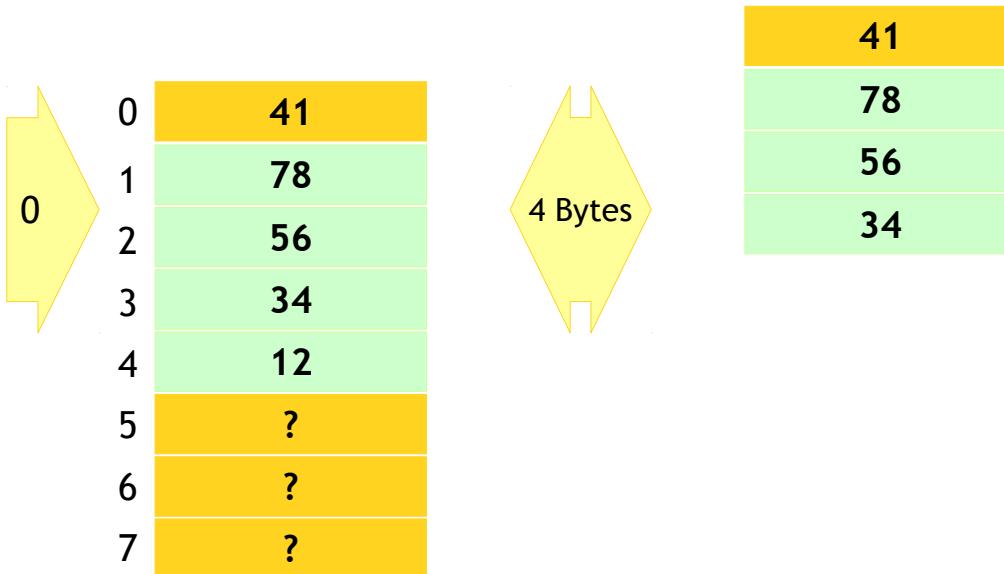
## Data Alignment



### Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching integer by the CPU will be like shown below



# Advanced C

## Data Alignment



### Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below

0	41
1	78
2	56
3	34
4	12
5	?
6	?
7	?



41
78
56
34

12
?
?
?

# Advanced C

## Data Alignment



### Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

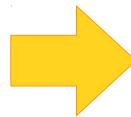
- Fetching the integer by the CPU will be like shown below

0	41
1	78
2	56
3	34
4	12
5	?
6	?
7	?

41
78
56
34

12
?
?
?



Shift 1 Byte Up

78
56
34
0



78
56
34
12

Combine

0
0
0
12

Shift 3 Byte Down

# Advanced C

## UDTs - Structures - Data Alignment - Padding

- Because of the data alignment issue, structures uses padding between its members if they don't fall under even address.
- So if we consider the following structure the memory allocation will be like shown in below figure

### Example

```
struct Test
{
    char ch1;
    int num;
    char ch2;
}
```

0	ch1
1	pad
2	pad
3	pad
4	num
5	num
6	num
7	num
8	ch2
9	pad
A	pad
B	pad

# Advanced C

## UDTs - Structures - Data Alignment - Padding

- You can instruct the compiler to modify the default padding behavior using **#pragma pack** directive

### Example

```
#pragma pack(1)

struct Test
{
    char ch1;
    int num;
    char ch2;
};
```

0	ch1
1	num
2	num
3	num
4	num
5	ch2

# Advanced C

## UDTs - Structures - Padding



### 013\_example.c

```
#include <stdio.h>

struct Student
{
    char ch1;
    int num;
    char ch2;
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));

    return 0;
}
```



# Advanced C

## UDTs - Structures - Padding



### 014\_example.c

```
#include <stdio.h>

#pragma pack(1)

struct Student
{
    char ch1;
    int num;
    char ch2;
};

int main()
{
    struct Student s1;

    printf("%zu\n", sizeof(struct Student));

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields

- The compiler generally gives the memory allocation in multiples of bytes, like 1, 2, 4 etc.,
- What if we want to have freedom of having getting allocations in bits?!
- This can be achieved with bit fields.
- But note that
  - The minimum memory allocation for a bit field member would be a byte that can be broken in max of 8 bits
  - The maximum number of bits assigned to a member would depend on the length modifier
  - The default size is equal to word size

# Advanced C

## UDTs - Structures - Bit Fields



### Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};
```

- The above structure divides a char into two nibbles
- We can access these nibbles independently

# Advanced C

## UDTs - Structures - Bit Fields



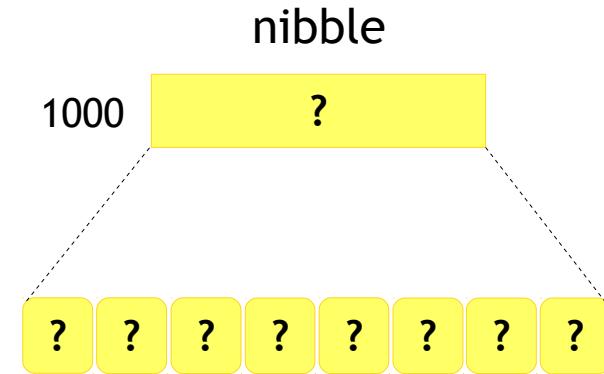
### 015\_example.c

```
struct Nibble
{
    unsigned char lower      : 4;
    unsigned char upper      : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



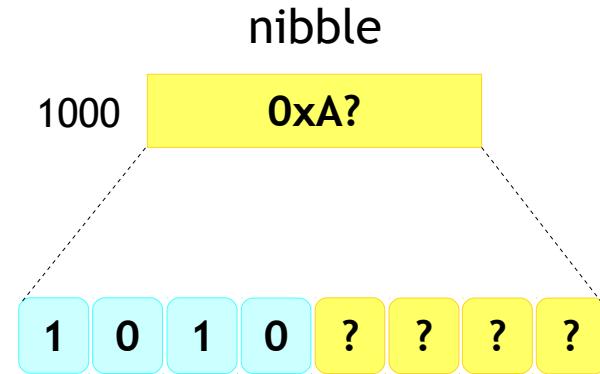
### 015\_example.c

```
struct Nibble
{
    unsigned char lower      : 4;
    unsigned char upper      : 4;
};

int main()
{
    struct Nibble nibble;

    → nibble.upper = 0x0A;
    nibble.lower = 0x02;

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



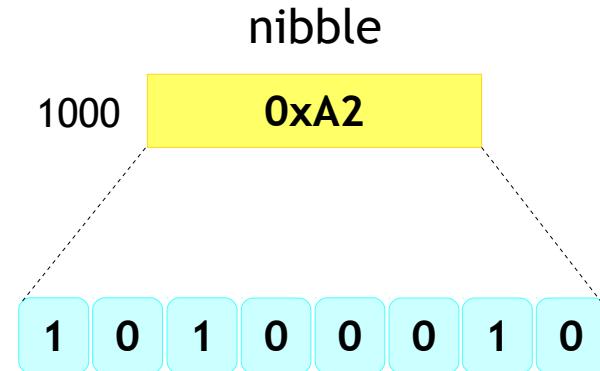
### 015\_example.c

```
struct Nibble
{
    unsigned char lower      : 4;
    unsigned char upper      : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02; → [dashed box]

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



### 016\_example.c

```
struct Nibble
{
    unsigned char lower      : 4;
    unsigned char upper      : 4;
};

int main()
{
    struct Nibble nibble;

    printf("%zu\n", sizeof(nibble));

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



### 017\_example.c

```
struct Nibble
{
    unsigned lower : 4;
    unsigned upper : 4;
};

int main()
{
    struct Nibble nibble;

    printf("%zu\n", sizeof(nibble));

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



### 018\_example.c

```
struct Nibble
{
    char lower : 4;
    char upper : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0x0A;
    nibble.lower = 0x02;

    printf("%d\n", nibble.upper);
    printf("%d\n", nibble.lower);

    return 0;
}
```



# Advanced C

## UDTs - Structures - Bit Fields



### 019\_example.c

```
struct Nibble
{
    char lower : 4;
    char upper : 4;
};

int main()
{
    struct Nibble nibble = {0x02, 0x0A};

    printf("%#o\n", nibble.upper);
    printf("%#x\n", nibble.lower);

    return 0;
}
```



# Advanced C

## UDTs - Unions



# Advanced C

## UDTs - Unions



- Like structures, unions may have different members with different data types.
- The major difference is, the structure members get different memory allocation, and in case of unions there will be single memory allocation for the biggest data type



# Advanced C

## UDTs - Unions



### Example

```
union Test
{
    char option;
    int id;
    double height;
};
```

- The above union will get the size allocated for the type double
- The size of the union will be 8 bytes.
- All members will be using the same space when accessed
- The value the union contain would be the latest update
- So as summary a single variable can store different type of data as required

# Advanced C

## UDTs - Unions



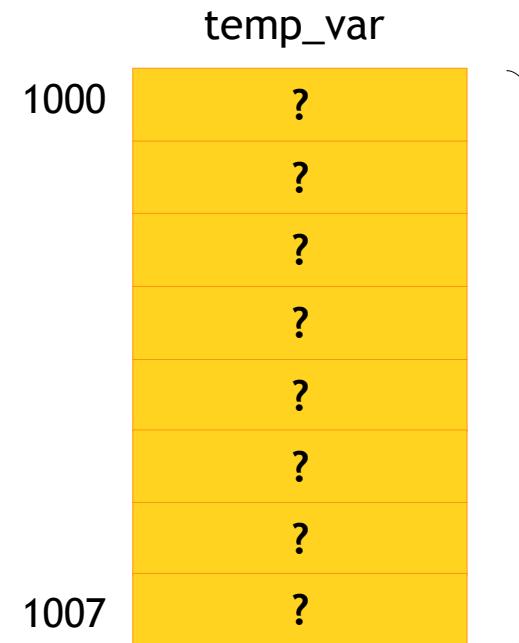
### 020\_example.c

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
→ union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```



Total 8 Bytes allocated since longest member is **double**

# Advanced C

## UDTs - Unions



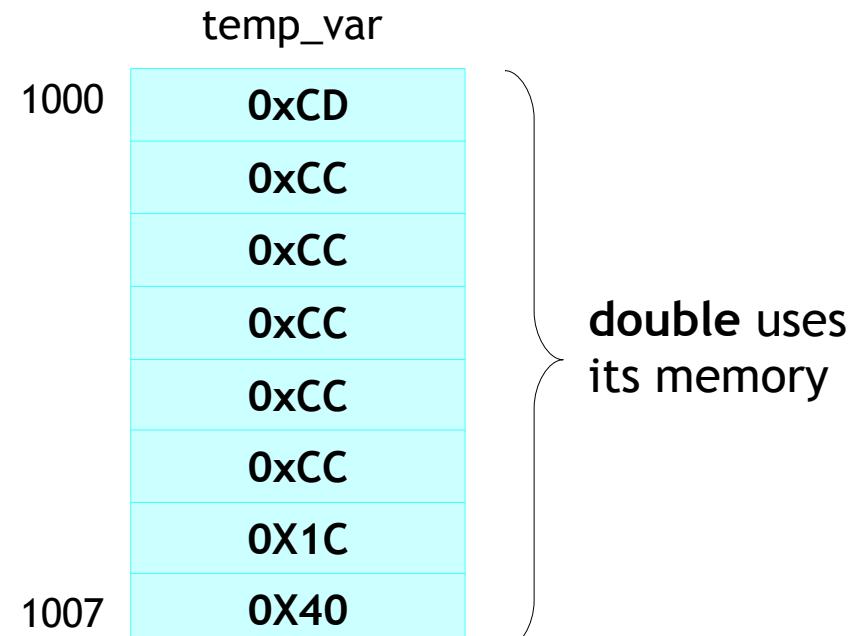
### 020\_example.c

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

→| temp_var.height = 7.2;
  temp_var.id = 0x1234;
  temp_var.option = '1';

    return 0;
}
```



# Advanced C

## UDTs - Unions



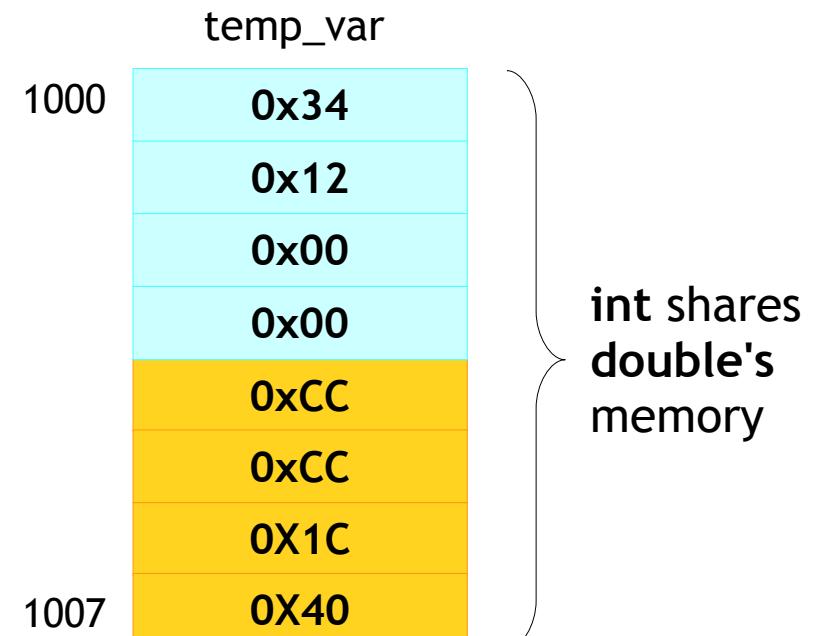
### 020\_example.c

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
→ temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```



# Advanced C

## UDTs - Unions



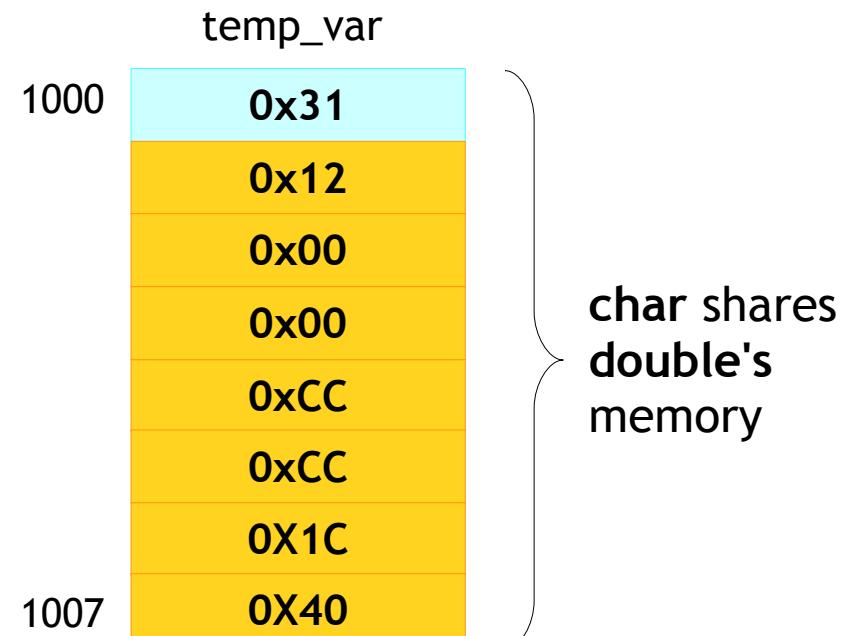
### 020\_example.c

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
→ temp_var.option = '1';

    return 0;
}
```



# Advanced C

## UDTs - Unions



### 021\_example.c

```
union FloatBits
{
    float degree;
    struct
    {
        unsigned m : 23;
        unsigned e : 8;
        unsigned s : 1;
    } elements;
};

int main()
{
    union FloatBits fb = {3.2};

    printf("Sign: %x\n", fb.elements.s);
    printf("Exponent: %x\n", fb.elements.e);
    printf("Mantissa: %x\n", fb.elements.m);

    return 0;
}
```

	fb
1000	0xCD
	0xCC
	0x4C
1004	0x40



# Advanced C

## UDTs - Unions



### 022\_example.c

```
union Endian
{
    unsigned int vlaue;
    unsigned char byte[4];
};

int main()
{
    union Endian e = {0x12345678};

    e.byte[0] == 0x78 ? printf("Little\n") : printf("Big\n");

    return 0;
}
```



# Advanced C

## UDTs - Typedefs



- Typedef is used to create a new name to the existing types.
- K&R states that there are two reasons for using a typedef.
  - First, it provides a means to make a program more portable. Instead of having to change a type everywhere it appears throughout the program's source files, only a single typedef statement needs to be changed.
  - Second, a typedef can make a complex definition or declaration easier to understand.



# Advanced C

## UDTs - Typedefs



### 023\_example.c

```
typedef unsigned int uint;

int main()
{
    uint number;

    return 0;
}
```

### 025\_example.c

```
typedef int array_of_100[100];

int main()
{
    array_of_100 array;

    printf("%zu\n", sizeof(array));

    return 0;
}
```

### 024\_example.c

```
typedef int * int_ptr;
typedef float * float_ptr;

int main()
{
    int_ptr ptr1, ptr2, ptr3;
    float_ptr fptr;

    return 0;
}
```

# Advanced C

## UDTs - Typedefs



### 026\_example.c

```
typedef struct _Student
{
    int id;
    char name[30];
    char address[150]
} Student;

void data(Student s)
{
    s.id = 10;
}

int main()
{
    Student s1;

    data(s1);

    return 0;
}
```

### 027\_example.c

```
#include <stdio.h>

typedef int (*fptr)(int, int);

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    fptr function;

    function = add;
    printf("%d\n", function(2, 4));

    return 0;
}
```

# Advanced C

## UDTs - Typedefs



### 028\_example.c

```
#include <stdio.h>

typedef signed int          sint, si;
typedef unsigned int         uint, ui;
typedef signed char          s8;
typedef signed short         s16;
typedef signed int           s32;
typedef unsigned char        u8;
typedef unsigned short       u16;
typedef unsigned int          u32;

int main()
{
    u8 count = 200;
    s16 axis = -70;

    printf("%u\n", count);
    printf("%d\n", axis);

    return 0;
}
```



# Advanced C

## UDTs - Typedefs - Standard



### Example

```
size_t - stdio.h  
ssize_t - stdio.h  
va_list - stdarg.h
```



# Advanced C

## UDTs - Typedefs - Usage



### 029\_example.c

```
typedef struct Sensor {
    int id;
    char name[12];
    int version;
    /*
     * The members of an anonymous union
     * are considered to be members of the
     * containing structure.
     */
    union { // Anonymous union
        float temperature;
        float humidity;
        char motion[4];
    };
} Sensor;
```



# Advanced C

## UDTs - Enums



- Set of named integral values
- Generally referred as named integral constants

### Syntax

```
enum name
{
    /* Members separated with , */
};
```

# Advanced C

## UDTs - Enums

### 030\_example.c

```
enum bool
{
    e_false,
    e_true
};

int main()
{
    printf("%d\n", e_false);
    printf("%d\n", e_true);

    return 0;
}
```

- The above example has two members with its values starting from 0.  
i.e, e\_false = 0 and e\_true = 1.

# Advanced C

## UDTs - Enums

### 031\_example.c

```
typedef enum
{
    e_red = 1,
    e_blue = 4,
    e_green
} Color;

int main()
{
    Color e_white = 0, e_black;

    printf("%d\n", e_white);
    printf("%d\n", e_black);
    printf("%d\n", e_green);

    return 0;
}
```

- The member values can be explicitly initialized
- There is no constraint in values, it can be in any order and same values can be repeated
- The derived data type can be used to define new members which will be uninitialized

# Advanced C

## UDTs - Enums

### 032\_example.c

```
int main()
{
    typedef enum
    {
        red,
        blue
    } Color;

    int blue;

    printf("%d\n", blue);
    printf("%d\n", blue);

    return 0;
}
```

- Enums does not have name space of its own, so we cannot have same name used again in the same scope.

# Advanced C

## UDTs - Enums



033\_example.c

```
typedef enum
{
    red,
    blue,
    green
} Color;

int main()
{
    Color c;

    printf("%zu\n", sizeof(Color));
    printf("%zu\n", sizeof(c));

    return 0;
}
```

- Size of Enum does not depend on number of members



# Advanced C

## UDTs - DIY



- WAP to accept students record. Expect the below output

### Screen Shot

```
user@user:~]./students_record.out
Enter the number of students : 2
Enter name of the student : Tingu
Enter P, C and M marks : 23 22 12
Enter name of the student : Pingu
Enter P, C and M marks : 98 87 87
```

Name	Maths	Physics	Chemistry
Tingu	12	23	22
Pingu	87	98	87
Average	49.50	60.50	54.50

```
user@user:~]
```

# Advanced C

## UDTs - DIY



- WAP to program to swap a nibble using bit fields



# User Defined Datatypes (UDT)

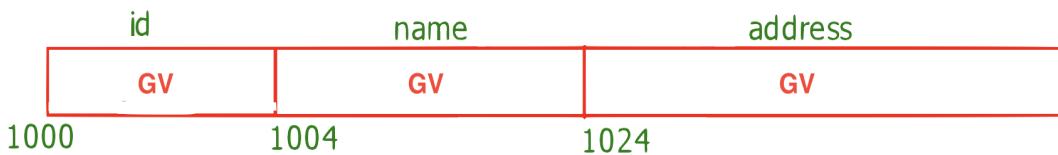
- User defined data types are collection of standard data types
- UDTs are used to create our own data type with the help of primitive data types.
- In UDT mainly we have,
  - Structures
  - Unions
  - Enum
  - Typedef

## 1. Structures

- Structure is a collection of the same or different data type under one common name.
  - Syntax:
    - ```
struct structure_name
{
    //structure members
};
```
    - To create structure variable
      - ```
struct structure_name variable_name;
```
  - Example:

```
struct Student
{
    int id;
    char name[20];
    char address[30];
};
```

    - Memory will not be allocated when u define/write structure
    - Memory is allocated when you create structure variable
- ```
struct Student st1;
```



- Memory allocated will be continuous and also depending on the structure padding
- Here, 84 bytes are allocated
- To access structure members/fields will use the .(dot) operator.  
E.g., st1.id = 100;

### Structure copy

- In the case of arrays we learned that copying one array to another is not straight forward, we need to use a loop.
- But in the case of structure, it can be copied by using assignment operator
- Example:
  - struct Student s1 = {10,"Rama","Bangalore"};

```
struct Student s2;
```

```
s2 = s1;
```

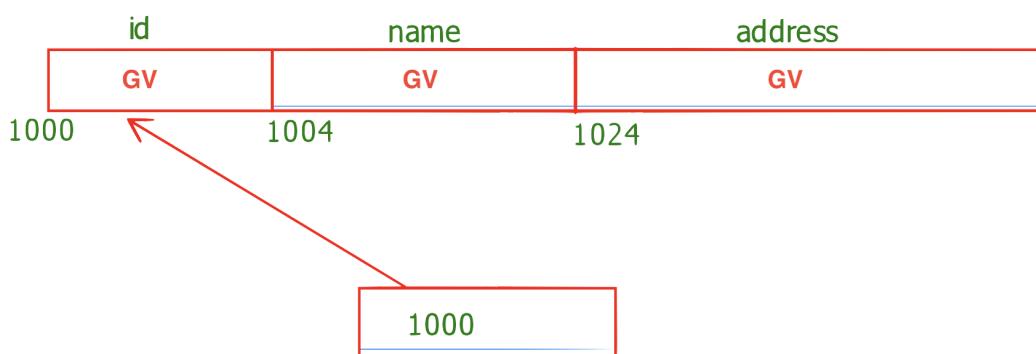
### Structure Pointers

- We can have structure pointer to store the address of structure variable
- For example:

```
struct Student *sptr
```

```
struct Student s1;
```

```
struct Student *sptr = &s1;
```



## How to access structure members or fields?

(\*sptr).id = 10; or sptr -> id = 10  
(\*sptr).name; or sptr -> name  
(\*sptr).address or sptr->address

## Passing structure to function

- Pass by value : Not recommended in case of structure because structure is huge user defined data type so it will take more space in stackframe

### main() - stackframe

| id   | name | address |
|------|------|---------|
| gv   | gv   | gv      |
| 1000 | 1004 | 1034    |

### data() - frame

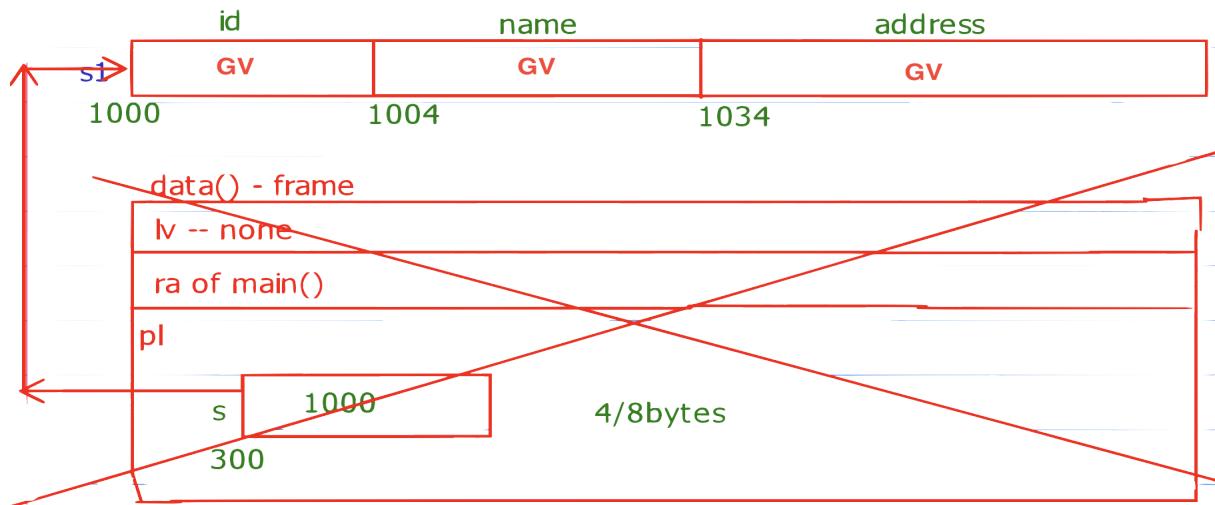
|                           |
|---------------------------|
| lv -- none                |
| ra of main()              |
| pl                        |
| id      name      address |
| s      10      gv         |
| 500      504      534     |

- As you can see in the above example structure takes upto 184 bytes in stack which is huge in size, so normally pass by value method is avoided in case of structure

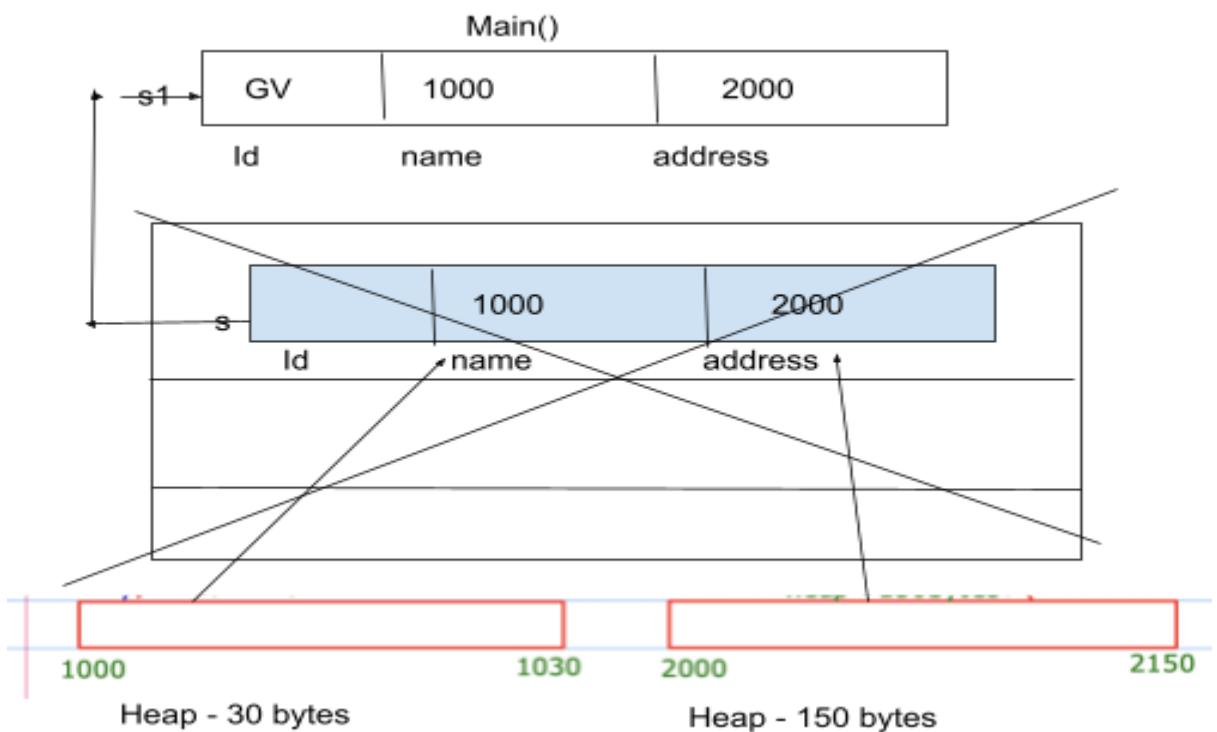
## Pass by Reference: preferred method because takes less memory

- As you can see in pass by reference it only need 4/8 bytes of memory.

### main() - stackframe



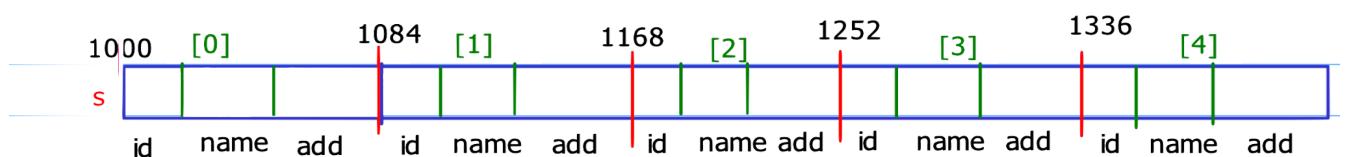
### Returning structure from function



- Like a normal variable a structure can be returned from function if its created in the heap like below:

## Array of structure

- Array of structure is a collection of more than 2 structure variables
- For example,
  - struct Student
  - {
  - int id;
  - char name[20];
  - char address[60];
- };
- struct Student s[5];



## Accessing array of structure

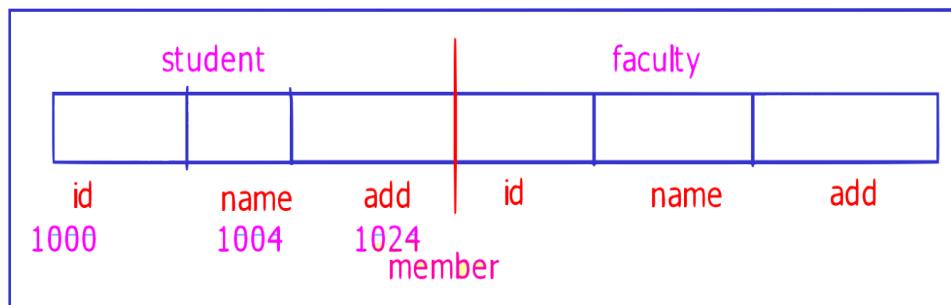
- Like a normal array we can access array of structure through loop:

```
for(i = 0; i < 5; i++)
{
    printf("%d %s %s\n", s[i].id,s[i].name,s[i].address);
}
```

## Nested Structures

```
struct College
{
    struct Student
    {
        int id;
        char name[20];
        char address[60];
    } student;

    struct
    {
        int id;
        char name[20];
        char address[60];
    } faculty;
};
```



member.student.id = 101  
member.student.name  
membebr.student.address

member.faculty.id = 11  
member.faculty.name  
member.faculty.address

## Structure Padding

- Structure padding is a process of adding useless bytes in between members of structure to make processor work easy.
- Structure padding is done based on following points:
  - based on the member arrangement
  - based on the largest member size
  - based on word size
- For example:

```
struct Student
```

```
{  
    char ch1;  
    int id;  
    char ch2;  
}
```



## 2. Unions

- Like structures, union is also a user defined datatype which can have the same or different types of data.
- Major difference between structure and union is, union shares the memory of the largest member among the other members.
- E.g.,
  - union Test
    - {
    - char option;
    - int id;
    - double height;
  - };
- In above example height is biggest member of the union so, the sizeof the union will be 8bytes and that memory is shared among the others

### **3. Typedefs**

- Typedefinition is used to create a new name to the xisting datatype
- E.g,
  - `typedef unsigned long int uli;`
- Typedef makes complex definition simpler like in the above example whenever you need to create a unsigned long int you dont have to write that full name, simply one can create like:
  - `uli n1;`
- Its that simple to use and typedef is used with all of the datatypes

### **4. Enum**

- Enum is a another user defined datatype which is used to group named constant under a single name.
- It is called as set of named integral values.
- For example:
  - `enum boolean`  
    {  
        false,  
        true  
    }
- In the enums first named constant always will have 0 as initialised value and next member will be incremented by 1.
- So, false is having value 0 and true is having value 1.
- We can also initialised own integral vlaues to the enum members like,
  - `enum boolean`  
    {  
        green = 90,  
        Blue = 40  
    }

# File I/O

- File is a stream of characters or sequence of bytes
- Files are used for persistent or permanent storage
- In C, accessing a file is not that straightforward, there are functions available in the stdio.h header file.
- With the help of standard I/O function we can read and write the content from a file.

## To open a file

- To open a file fopen function will be used and prototype of that function is:

**FILE \*fopen(const char \*path, const char \*mode);**

where, → \*path - name of a file/ file along path enclosed within double quotes  
e.g., "text.txt" , "/usr/desktop/text.txt" , "c:/folder/text.txt"

→ \*mode - mode of file, which can be any one from below list

- **"r" mode**
  - read only mode
  - can only read information from a file / only read operation
  - If file exists, then it will open the file Else returns a NULL
  - If success -> returned pointer will points to first character of the file points to beginning of the file
- **"w" mode**
  - Write only mode
  - can only write information to a file / only write operation
  - If file exists, then it will open the file and can write the info Else it will create a new file in the given path
  - If success –
    - File pointer will points to first character of the file
    - If already file contains the information then write mode will replace the previous contents with new contents
- **"a" mode**
  - append mode
  - Append mode is used to add the content to a file.
  - If file exists, then it will open the file and add the info Else it will create a new file
  - If success –
    - File pointer will points to last character of the file
    - If file already contains the info then append mode will concat or add the previous contents with new contents

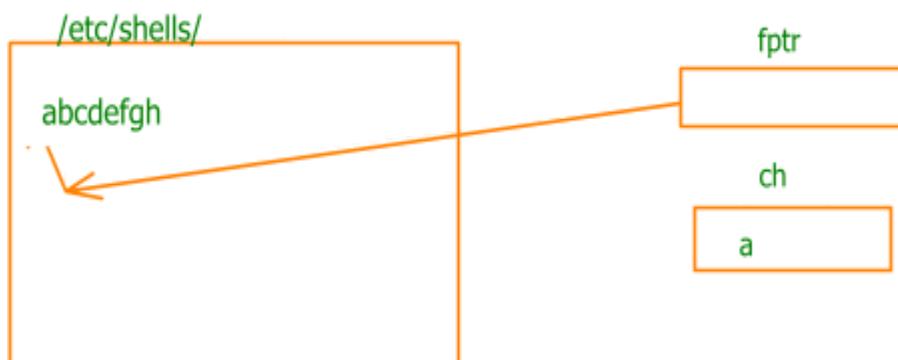
- “**r+**” mode
  - both read and write operation
  - If file exists it open the file else it will return NULL
- “**w+**” mode
  - both write and read operation
  - If file exists open the file else create a new file
- “**a+**” mode
  - both read and write operation
  - If file exists open the file else create a new file
  - new content will be concatenated with previous

## To close a file

- It's recommended to close the file pointer after all the file operation is completed.
- To close a file fclose or fcloseall will be used
- Function prototype:
  - fclose(filepointername)
  - fcloseall()
- fclose will close the particular file only, but fclose all will close all the file pointers opened in the program.

## How to read content from a file?

- To read content from file, fgetc is used
- Fgetc will fetch a character from file
- To check the end of file, feof function will be helpful
  - feof(filepointer)
- fputc(ch,stdout) -> it will print a character on stdout/on screen



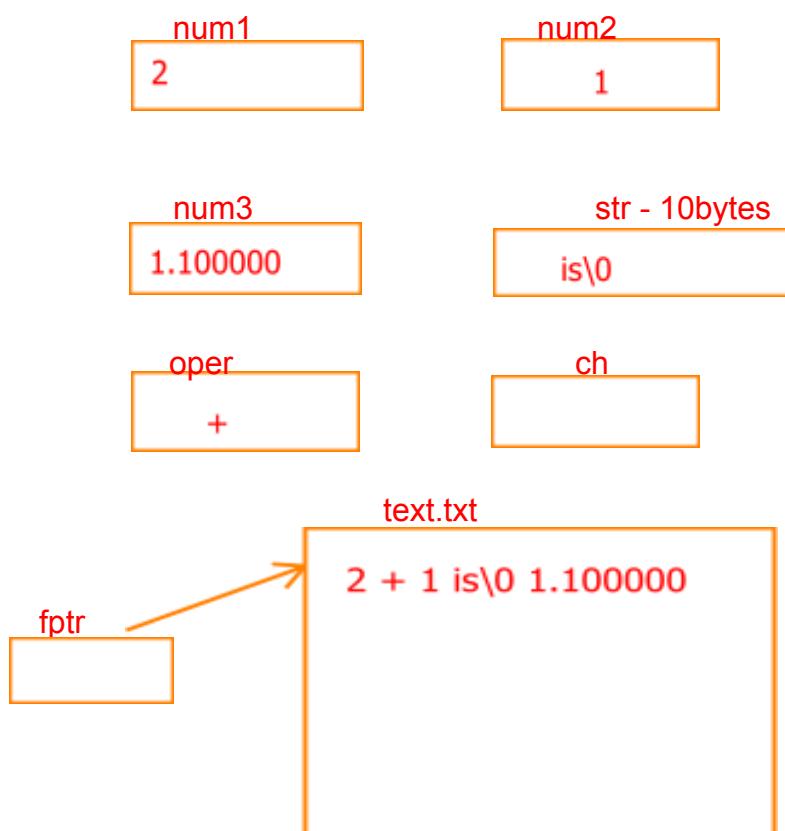
## Error check in files

- Initially whenever a file pointer opens there is an error flag associated with it.
- That error flag will be 0 in the initial stage.
- After a certain FILE I/O operation, consider an example
  - fopen("file.txt", "w");
  - File.txt is opened in write mode, if you try to read content from file it will be an error
  - In such scenario the error flag will be set to 1, to check the errors we use function ferror(FILE \*)

## To know the file pointer position in the file

- ftell(FILE \*)
- By using ftell we can fetch the position of file pointer
- Provide some info/convey the message
- it will give information about where the file pointer is pointing
- fptr position will always be starting 0

006.c -- rewind(FILE \*) --> make the file pointer to point to starting position



## To move file pointer anywhere in the file:

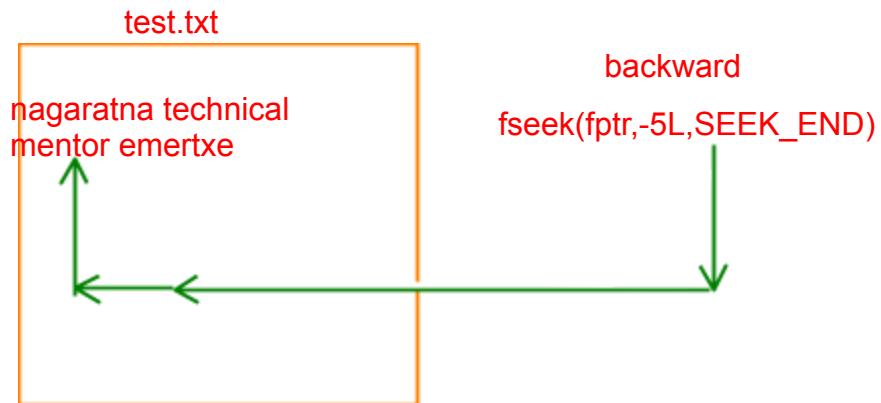
- Fseek function is used to move file pointer
- `fseek(fp, how_much, from_where)`

how much u need to move --> 0L (long value), 10L, -10L

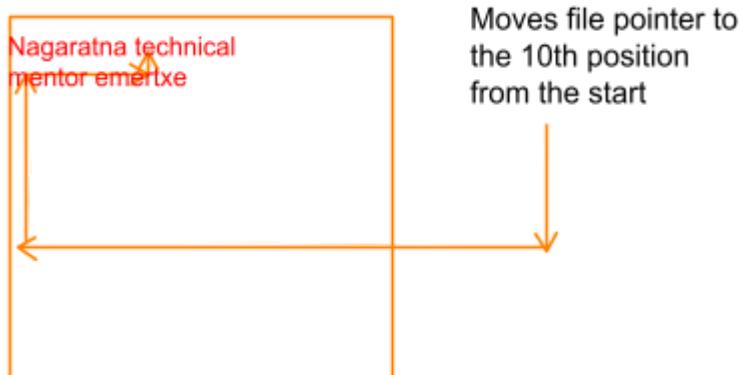
from where --> SEEK\_SET --> file pointer will start from beginning of file

SEEK\_END --> file pointer will be set to end of the file

SEEK\_CUR --> file pointer will move from current position



`fseek(fp,10,SEEK_SET)`



`fseek(fp,10,SEEK_CUR);`

→ moves file pointer to the 10th position forward from current position

`fseek(fp,0L,SEEK_SET); // equivalent to rewind() method`

→ brings back the file pointer to the beginning of the position

## Fwrite and fread

- These 2 functions are another way of reading or writing the content from a file.
- Both the function reads and writes the content as raw data which is non-human readable.
- Syntax:
  - `size_t fwrite(const *ptr, size_t size, size_t nitems, FILE *stream)`
  - `size_t fread(const *ptr, size_t size, size_t nitems, FILE *stream)`

# File I/O



# Advanced C

## File Input / Output



- Sequence of bytes
- Could be regular or binary file
- Why?
  - Persistent storage
  - Theoretically unlimited size
  - Flexibility of storing any type data



# Advanced C File Input / Output



## Regular File

A regular file looks normal as it appears here.

regular files are generally group of ASCII characters hence the Sometimes called as text files which is human readable.

The binary file typically contains the raw data. The contents of a Binary file is not human readable

## Binary File



# Advanced C

## File Input / Output - Via Redirection



- General way for feeding and getting the output is using standard input (keyboard) and output (screen)
- By using redirection we can achieve it with files i.e  
`./a.out < input_file > output_file`
- The above line feed the input from input\_file and output to output\_file
- The above might look useful, but its the part of the OS and the C doesn't work this way
- C has a general mechanism for reading and writing files, which is more flexible than redirection

# Advanced C

## File Input / Output



- C abstracts all file operations into operations on streams of bytes, which may be "**input streams**" or "**output streams**"
- No direct support for random-access data files
- To read from a record in the middle of a file, the programmer must create a stream, seek to the middle of the file, and then read bytes in sequence from the stream
- Let's discuss some commonly used file I/O functions

# Advanced C

## File IO - File Pointer



- **stdio.h** is used for file I/O library functions
- The data type for file operation generally is

Type

```
FILE *fp;
```

- FILE pointer, which will let the program keep track of the file being accessed
- Operations on the files can be
  - Open
  - File operations
  - Close

# Advanced C

## File IO - Functions - fopen() & fclose()



### Prototype

```
FILE *fopen(const char *filename, const char *mode);  
int fclose(FILE *filename);
```

Where mode are:

- r - open for reading
- w - open for writing (file need not exist)
- a - open for appending (file need not exist)
- r+ - open for reading and writing, start at beginning
- w+ - open for reading and writing (overwrite file)
- a+ - open for reading and writing (append if file exists)

### 001\_example.c

```
#include <stdio.h>  
  
int main()  
{  
    FILE *fp;  
  
    fp = fopen("test.txt", "r");  
    fclose(fp);  
  
    return 0;  
}
```

# Advanced C

## File IO - Functions - fopen() & fclose()

### 002\_example.c

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

int main()
{
    FILE *input_fp;

    input_fp = fopen("text.txt", "r");

    if (input_fp == NULL)
    {
        return 1;
    }

    fclose(input_fp);

    return 0;
}
```

# Advanced C

## File IO - DIY



- Create a file named **text.txt** and add some content to it.
  - WAP to print its contents on standard output
  - WAP to copy its contents in **text\_copy.txt**



# Advanced C

## File IO - Functions - feof()

### 003\_example.c

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

int main()
{
    FILE *fptr;
    char ch;

    fptr = fopen("/etc/shells", "r");

    /* Need to do error checking on fopen() */

    while (ch = fgetc(fptr))
    {
        if (feof(fptr))
            break;

        fputc(ch, stdout);
    }

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - Functions - perror() and clearerr()

004\_example.c

```
#include <stdio.h>

int main()
{
    FILE *fptr;
    char ch;

    fptr = fopen("file.txt", "w");

    ch = fgetc(fptr); /* This should fail since reading a file in write mode*/
    if (ferror(fptr))
        fprintf(stderr, "Error in reading from file : file.txt\n");

    clearerr(fptr);

    /* This loop should be false since we cleared the error indicator */
    if (ferror(fptr))
        fprintf(stderr, "Error in reading from file : file.txt\n");

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - Functions - ftell()

### 005\_example.c

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

int main()
{
    FILE *fptr;
    char ch;

    fptr = fopen("/etc/shells", "r");

    /* Need to do error checking on fopen() */

    printf("File offset is at -> %ld\n\n", ftell(fptr));
    printf("--> The content of file is <--\n");

    while ((ch = fgetc(fptr)) != EOF)
        fputc(ch, stdout);

    printf("\nFile offset is at -> %ld\n", ftell(fptr));

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - DIY



- Create a file named **text.txt** and add “abcdabcabcdaabc”.
  - WAP program to find the occurrences of character 'c' using **fscanf()**



# Advanced C

## File IO - Functions - fprintf(), fscanf() & rewind()

### 006\_example.c

```
#include <stdio.h>

int main()
{
    int num1, num2;
    float num3;
    char str[10], oper, ch;
    FILE *fptr;

    if ((fptr = fopen("text.txt", "w+")) == NULL)
    {
        fprintf(stderr, "Can't open input file text.txt!\n");
        return 1;
    }

    fprintf(fptr, "%d %c %d %s %f\n", 2, '+', 1, "is", 1.1);
    rewind(fptr);
    fscanf(fptr, "%d %c %d %s %f", &num1, &oper, &num2, str, &num3);

    printf("%d %c %d %s %f\n", num1, oper, num2, str, num3);

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - Functions - fseek()

### 007\_example.c

```
#include <stdio.h>

int main()
{
    int num1, num2;
    float num3;
    char str[10], oper, ch;
    FILE *fptr;

    if ((fptr = fopen("text.txt", "w+")) == NULL)
    {
        fprintf(stderr, "Can't open input file text.txt!\n");
        return 1;
    }

    fprintf(fptr, "%d %c %d %s %f\n", 2, '+', 1, "is", 1.1);
    fseek(fptr, 0L, SEEK_SET);
    fscanf(fptr, "%d %c %d %s %f", &num1, &oper, &num2, str, &num3);

    printf("%d %c %d %s %f\n", num1, oper, num2, str, num3);

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - Functions - fwrite() & fread()

### 008\_example.c

```
#include <stdio.h>

int main()
{
    int num1, num2;
    FILE *fptr;

    if ((fptr = fopen("text.txt", "w+")) == NULL)
    {
        fprintf(stderr, "Can't open input file text.txt!\\n");
        return 1;
    }

    scanf("%d%d", &num1, &num2);

    fwrite(&num1, sizeof(num1), 1, fptr);
    fwrite(&num2, sizeof(num2), 1, fptr);
    rewind(fptr);
    fread(&num1, sizeof(num1), 1, fptr);
    fread(&num2, sizeof(num2), 1, fptr);

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

    fclose(fptr);

    return 0;
}
```

# Advanced C

## File IO - Functions - fwrite() & fread()

### 009\_example.c

```
#include <stdio.h>

int main()
{
    struct Data d1 = {2, '+', 1, "is", 1.1};
    struct Data d2;
    FILE *fptr;

    if ((fptr = fopen("text.txt", "w+")) == NULL)
    {
        fprintf(stderr, "Can't open input file text.txt!\n");
        return 1;
    }

    fwrite(&d1, sizeof(d1), 1, fptr);
    rewind(fptr);
    fread(&d2, sizeof(d2), 1, fptr);

    printf("%d %c %d %s %f\n", d2.num1, d2.oper, d2.num2, d2.str, d2.num3);

    fclose(fptr);

    return 0;
}
```

```
struct Data
{
    int num1;
    char oper;
    int num2;
    char str[10];
    float num3;
};
```

# Advanced C

## File IO - DIY

- WAP to accept students record from user. Store all the data in as a binary file
- WAP to read out entries by the previous program

### Screen Shot

```
user@user:~]./students_record_enrty.out
Enter the number of students : 2
Enter name of the student : Tingu
Enter P, C and M marks : 23 22 12
Enter name of the student : Pingu
Enter P, C and M marks : 98 87 87
user@user:~]./read_students_record.out
```

| Name    | Maths | Physics | Chemistry |
|---------|-------|---------|-----------|
| Tingu   | 12    | 23      | 22        |
| Pingu   | 87    | 98      | 87        |
| Average | 49.50 | 60.50   | 54.50     |

```
user@user:~]
```