

C

Come let's see how deep it is!!

Team Emertxe



Advanced C

Course: Education goals and objectives

- Expectations out of this module
- Course: Education goals and objectives
- At the end of the course the student will learn
- Collaboration Policy
- Assignment Policy
- Course flow

Advanced C ontents

```
int main(void) {
```

Course: Education goals and objectives

Problem Solving

Introduction to C Programming

Basic Refreshers

Functions Part1

Pointers Part1

Standard Input and Output

Strings

Storage Classes

Pointers Part2

Functions Part2

Preprocessing

User Defined Datatypes

File Input and Output

Miscellaneous

}

Thank You

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 - 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

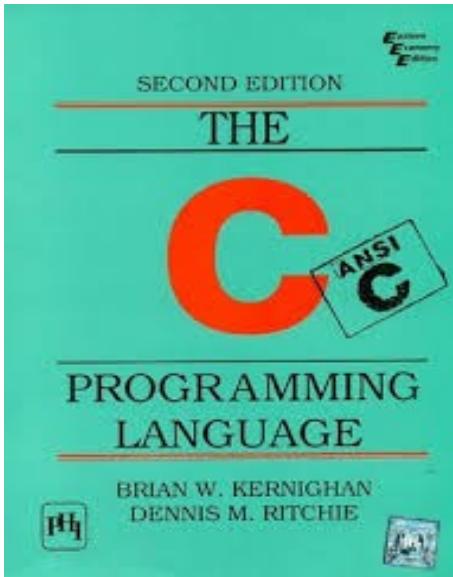
Advanced C

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 - 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 - 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 int float double	Decision	if else switch case default
Modifiers	signed unsigned short long	Storage Class	auto register static extern
Qualifiers	const volatile	Derived	struct union
Loops	for while do	User defined	enum typedef
Jump	goto break continue	Others	void return 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

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

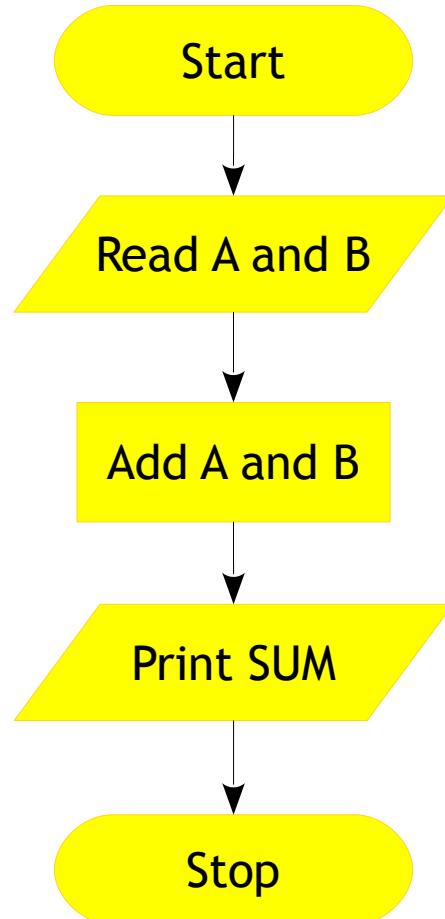
END

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

Basic Refreshers



Number System



Advanced C

Number Systems

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

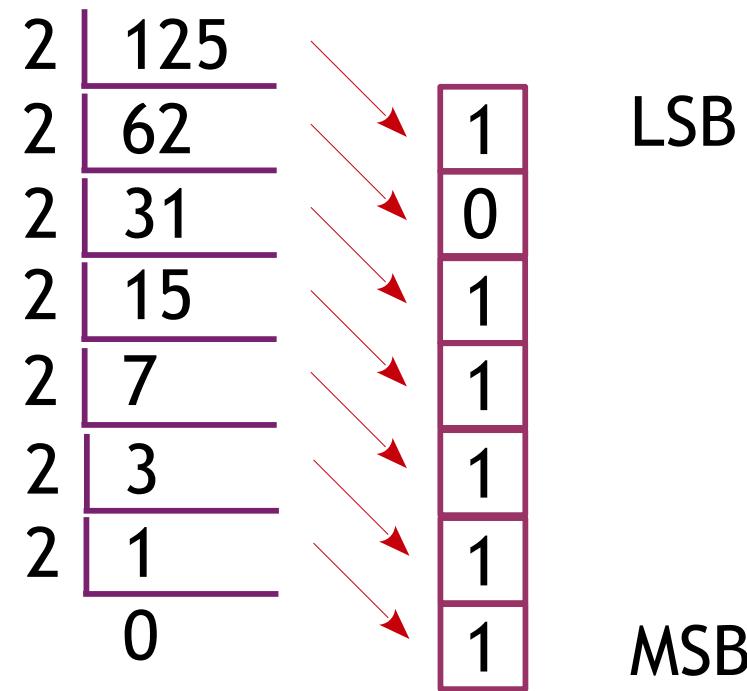
Type	Range (8 Bits)
Decimal	0 - 255
Octal	000 - 0377
Hexadecimal	0x00 - 0xFF
Binary	0b00000000 - 0b11111111

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

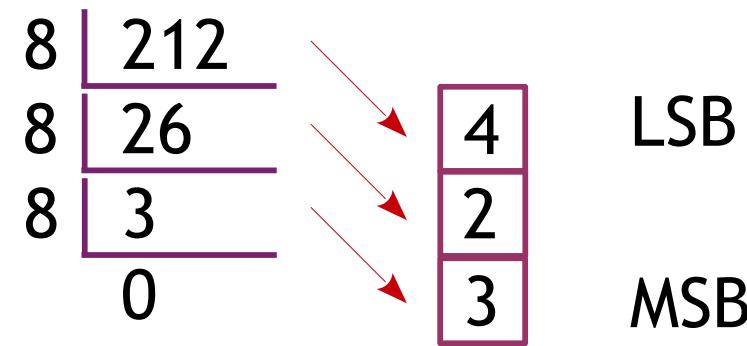


- So 125_{10} is 1111101_2

Advanced C

Number Systems - Decimal to Octal

- 212_{10} to Octal

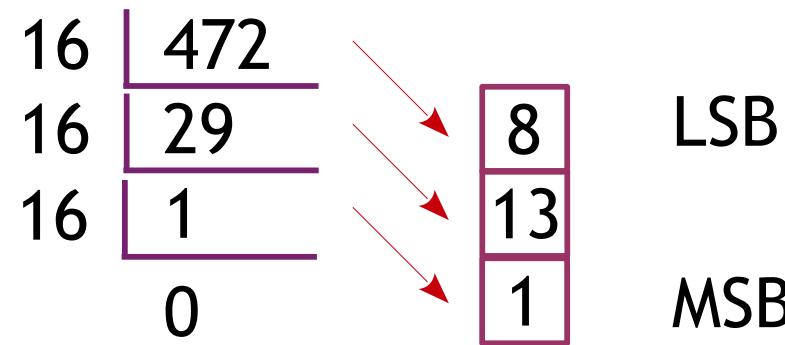


- So 212_{10} is 324_8

Advanced C

Number Systems - Decimal to Hexadecimal

- 472_{10} to Hexadecimal



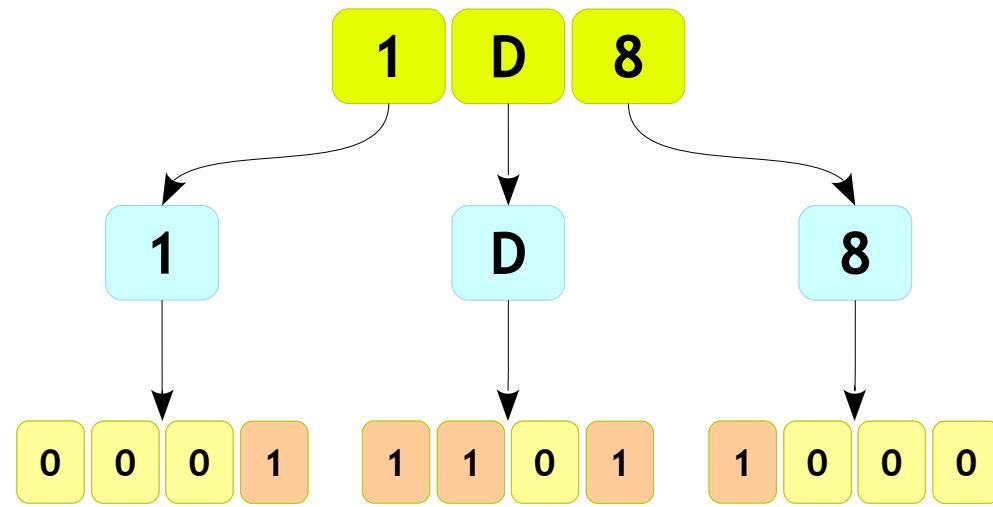
Representation	Substitutes
Dec	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Hex	0 1 2 3 4 5 6 7 8 9 A B C D E F

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

Advanced C

Number Systems - Hexadecimal to Binary

- $1D8_{16}$ to Binary

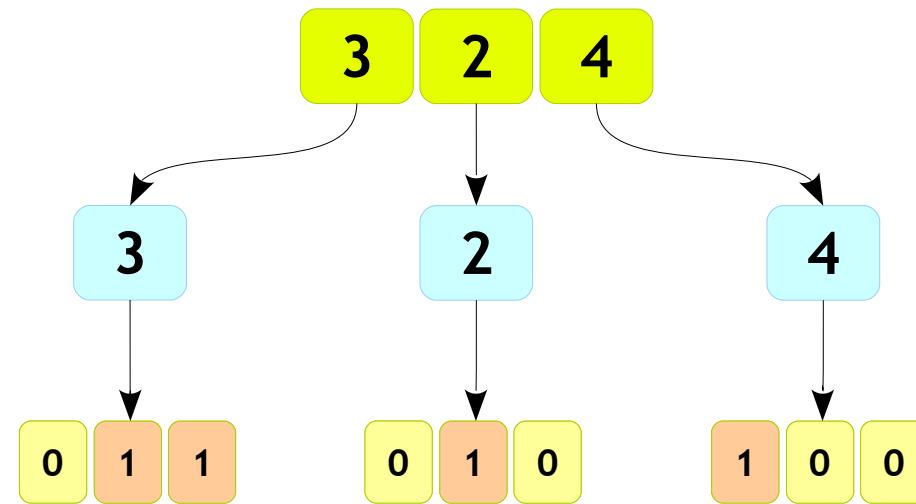


- So $1D8_{16}$ is 000111011000_2 which is nothing but 111011000_2

Advanced C

Number Systems - Octal to Binary

- 324_8 to Binary

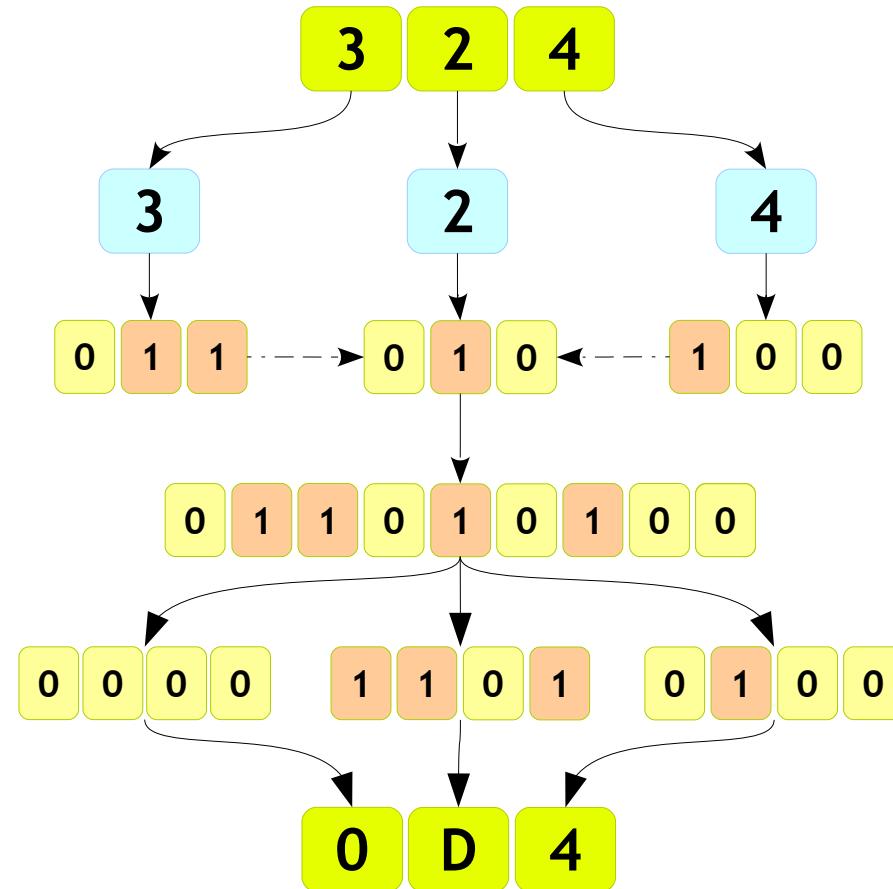


- So 324_8 is 011010100_2 which is nothing but 11010100_2

Advanced C

Number Systems - Octal to Hexadecimal

- 324_8 to Hexadecimal

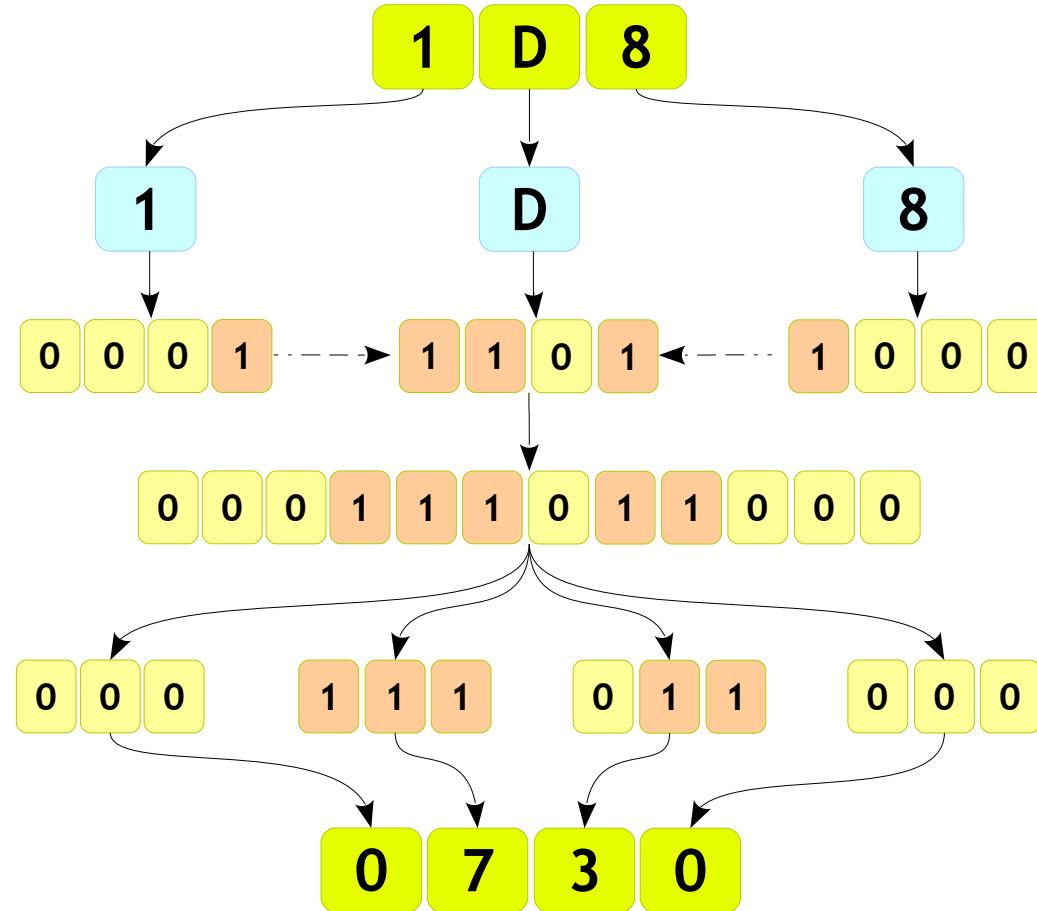


- So 324_8 is $0D4_{16}$ which is nothing but $D4_{16}$

Advanced C

Number Systems - Hexadecimal to Octal

- $1D8_{16}$ to Octal

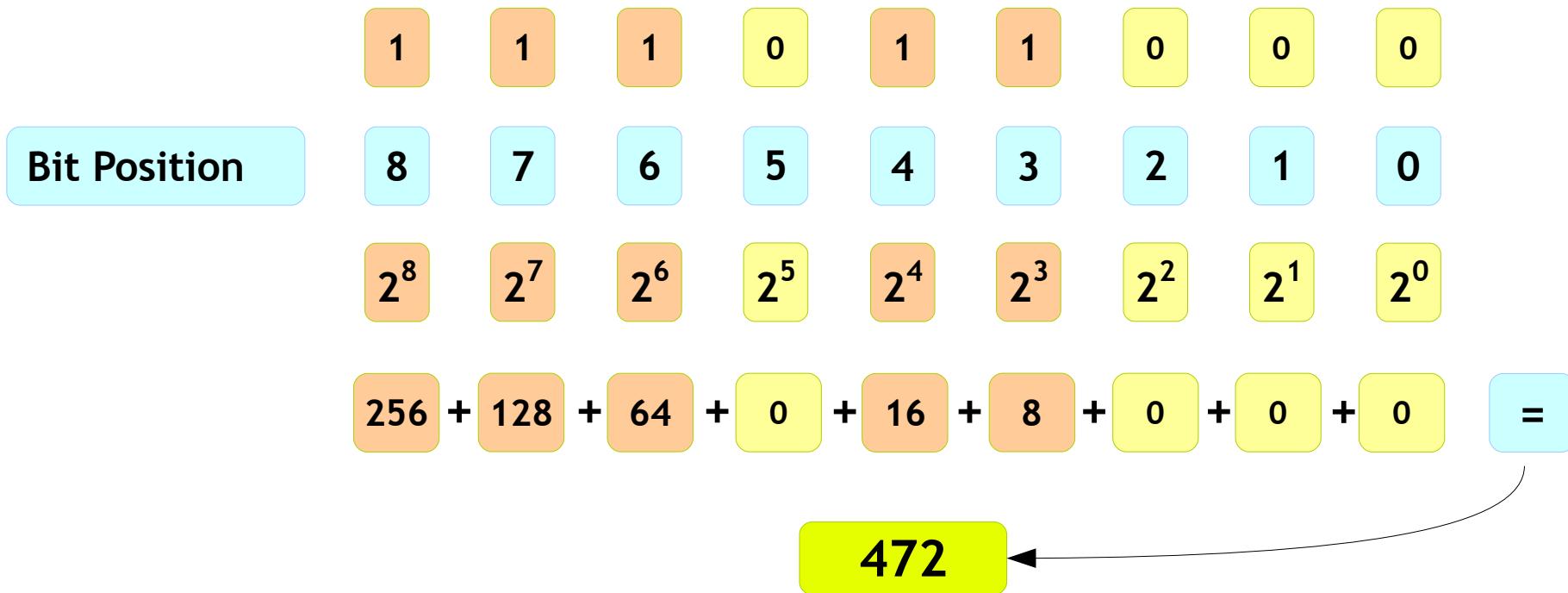


- So $1D8_{16}$ is 0730_8 which is nothing but 730_8

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
1	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 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 1 1 0 1
1's Compli	1 0 0 1 0
Add 1	0 1
2's Compli	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
Float (32 bits) Single Precision	1 bit	8 bits	23 bits
Double (64 bits) Double Precision	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 0.5 to IEEE 32-bit floating point format

Step 1:

$$0.5 \times 2 = 1.0$$

$$0.5_{10} = 0.1_2$$

Step 2:

Normalize:

$$0.1_2 = 1.0_2 \times 2^{-1}$$

Mantissa is 00000000000000000000

Exponent is $-1 + 127 = 126 = 01111110$,

Sign bit is 0

Advanced C

Float Point Number - Conversion - Example 2

Convert 8.25 to IEEE 32-bit floating point format

Step 1:

Integer Part to Binary:

8	/ 2	4	0
4	/ 2	2	0
2	/ 2	1	0
1			1

Fractional Part to Binary:

0.25	$\times 2$	0.5	0
0.5	$\times 2$	1.0	1

Result:

$$8.25_{10} = 1000.01_2$$

Step 2:

Normalize:

$$1000.01, = 1.00001, \times 2^3$$

Mantissa is 000010000000000000000000

Exponent is $3 + 127 = 130 = 1000\ 0010_2$,

Sign bit is 0

Advanced C

Float Point Number - Conversion - Example 3

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 4

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

$39887.5625_{10} =$

100110111001111.1001_2

Step 2:

Normalize:

$$100110111001111.1001_2 = \\ 1.00110111001111001_2 \times 2^{15}$$

Mantissa is 00110111001111001000
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 0 0 1 1 1 1 0 0 1 0 0 0 0 0

Advanced C

Float Point Number - Conversion - Example 6

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

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

Step 2:

Normalize:

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

Mantissa is 10101010000000000000000

Exponent is $3 + 127 = 130 = 1000\ 0010$,

Sign bit is 1

Advanced C

Float Point Number - Conversion - Example 8

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 0

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

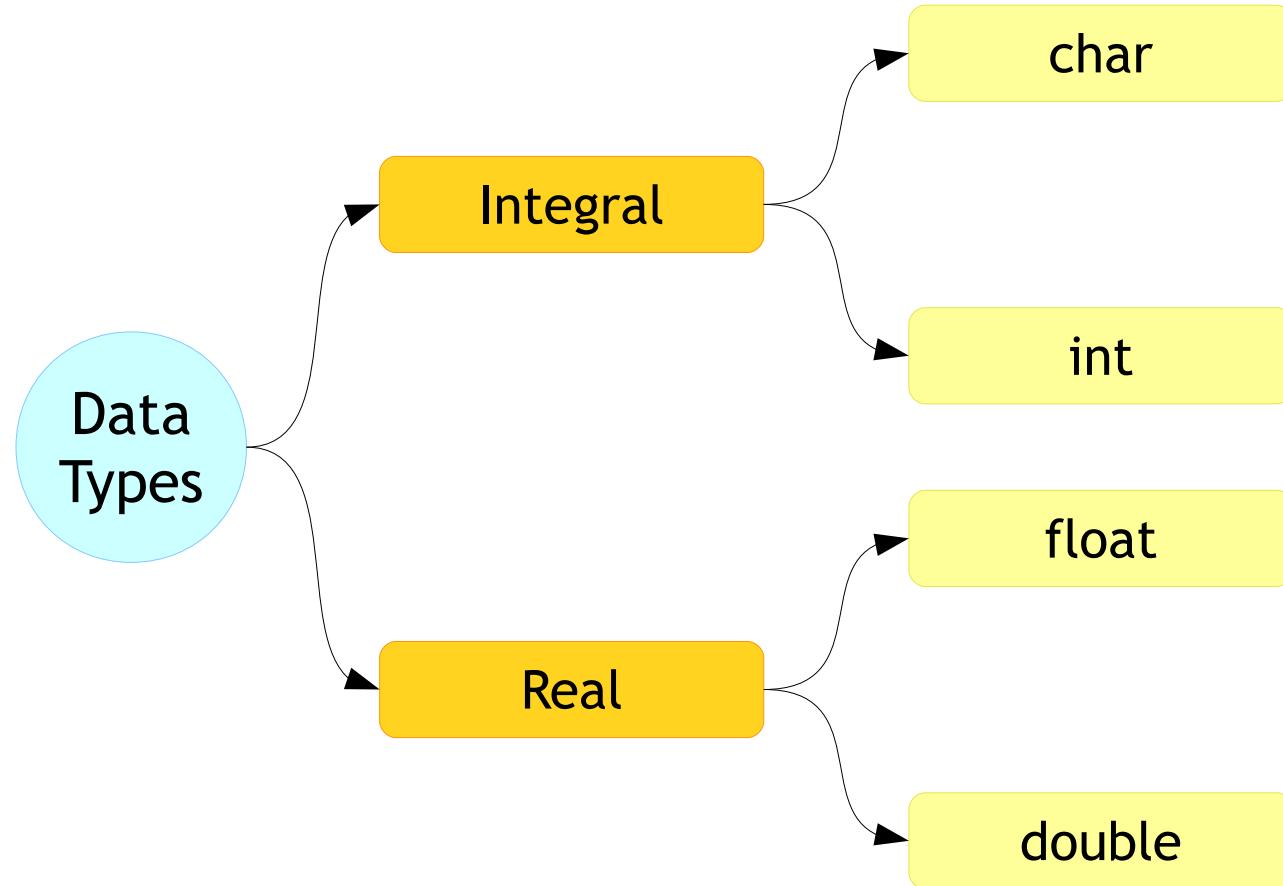
Bit	S	Exponent	Mantissa
Position	31	30 29 28 27 26 25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Value	0	0 1 1 1 1 1 1	1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1

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;
```

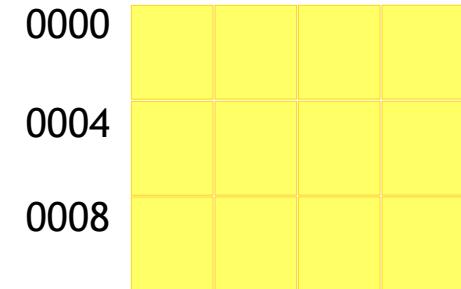


Advanced C

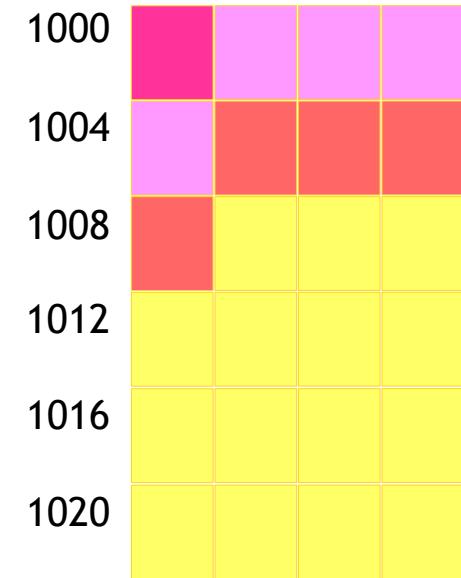
Data Types - Storage

Example

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



•
•
•



Advanced C

Data Types - Printing

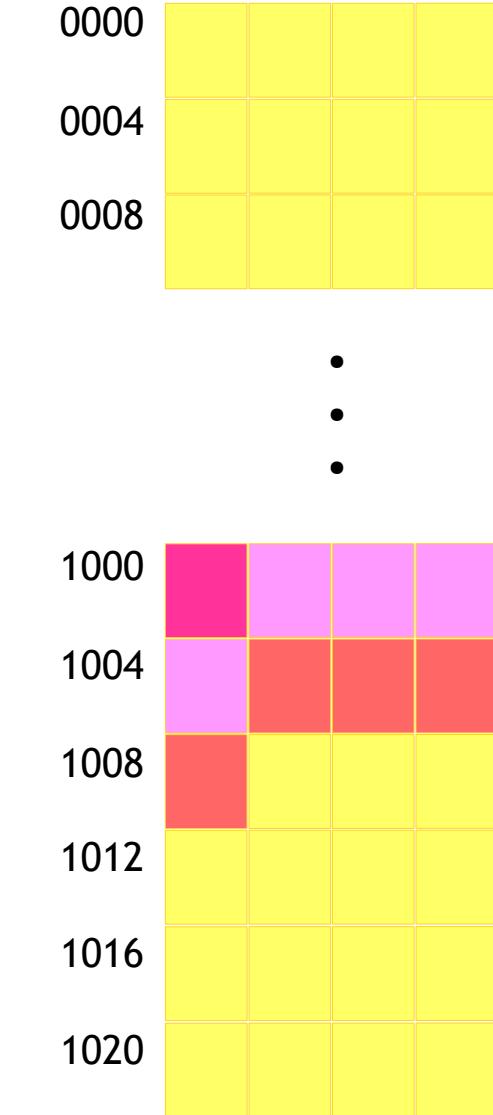
001_example.c

```
#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 - Scanning



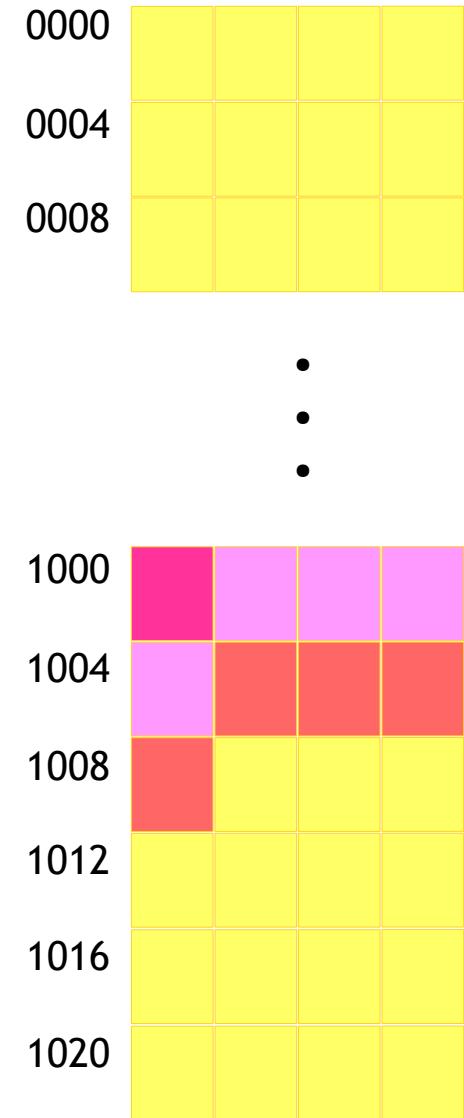
002_example.c

```
#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

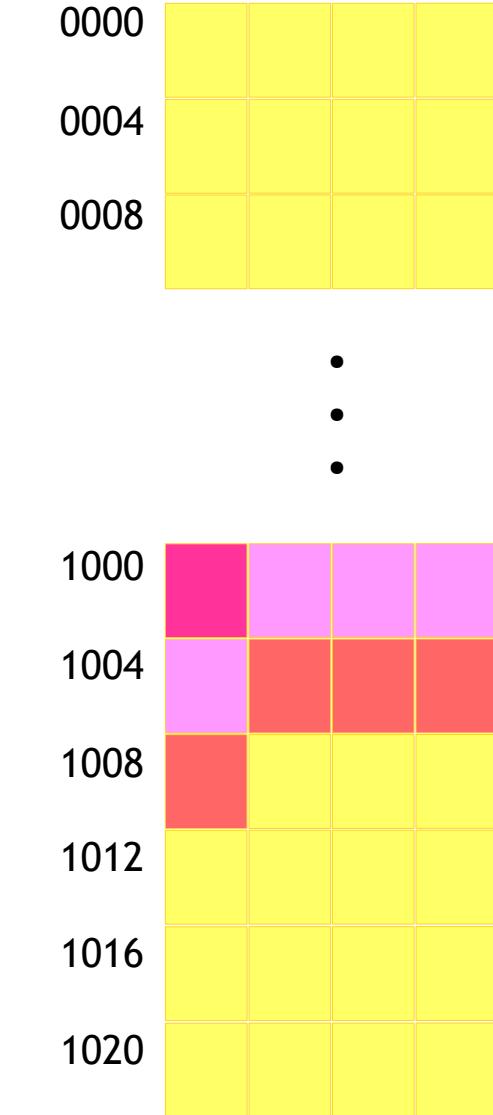
003_example.c

```
#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;
}
```



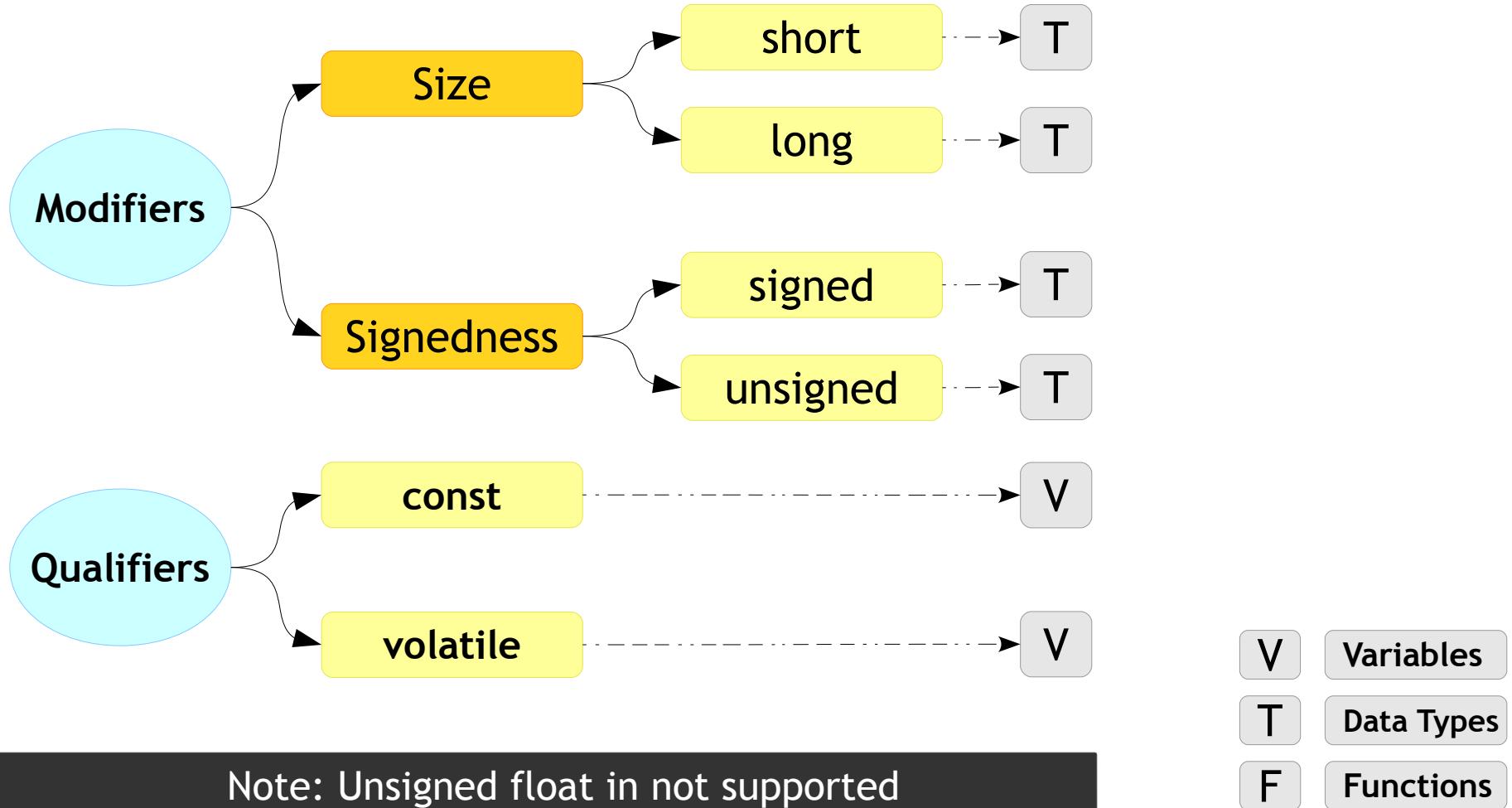
Advanced C

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 specifers and modifiers and some people even call all as qualifiers!

Advanced C

Data Types - Modifiers and Qualifiers



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

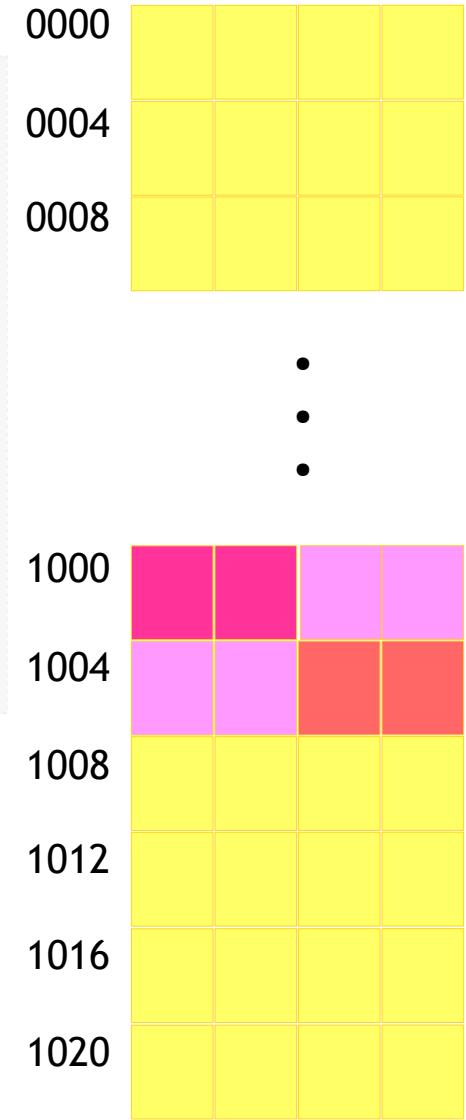
004_example.c

```
#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

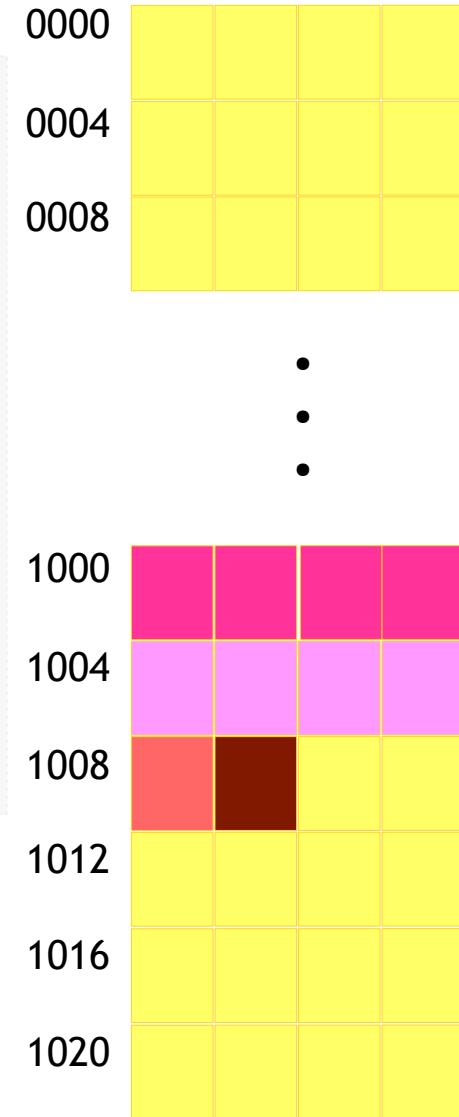
005_example.c

```
#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

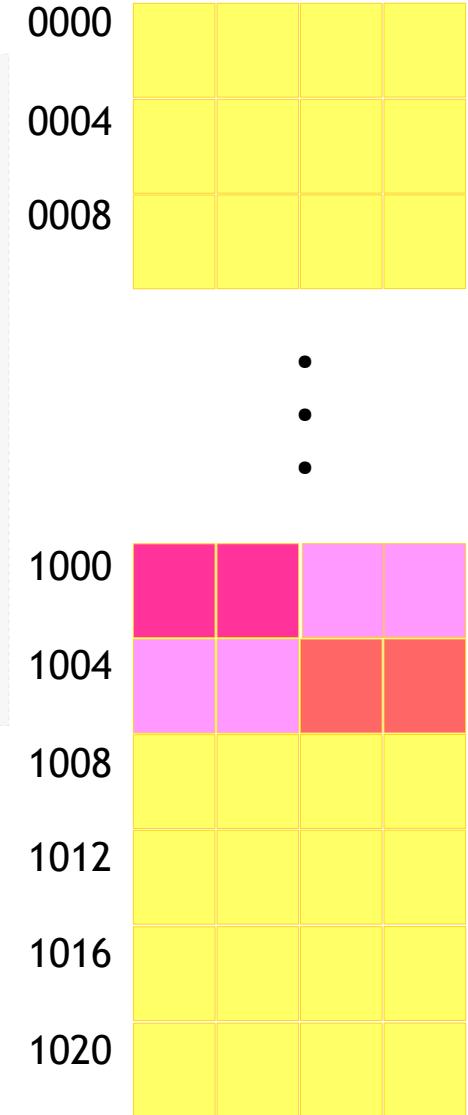
006_example.c

```
#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>

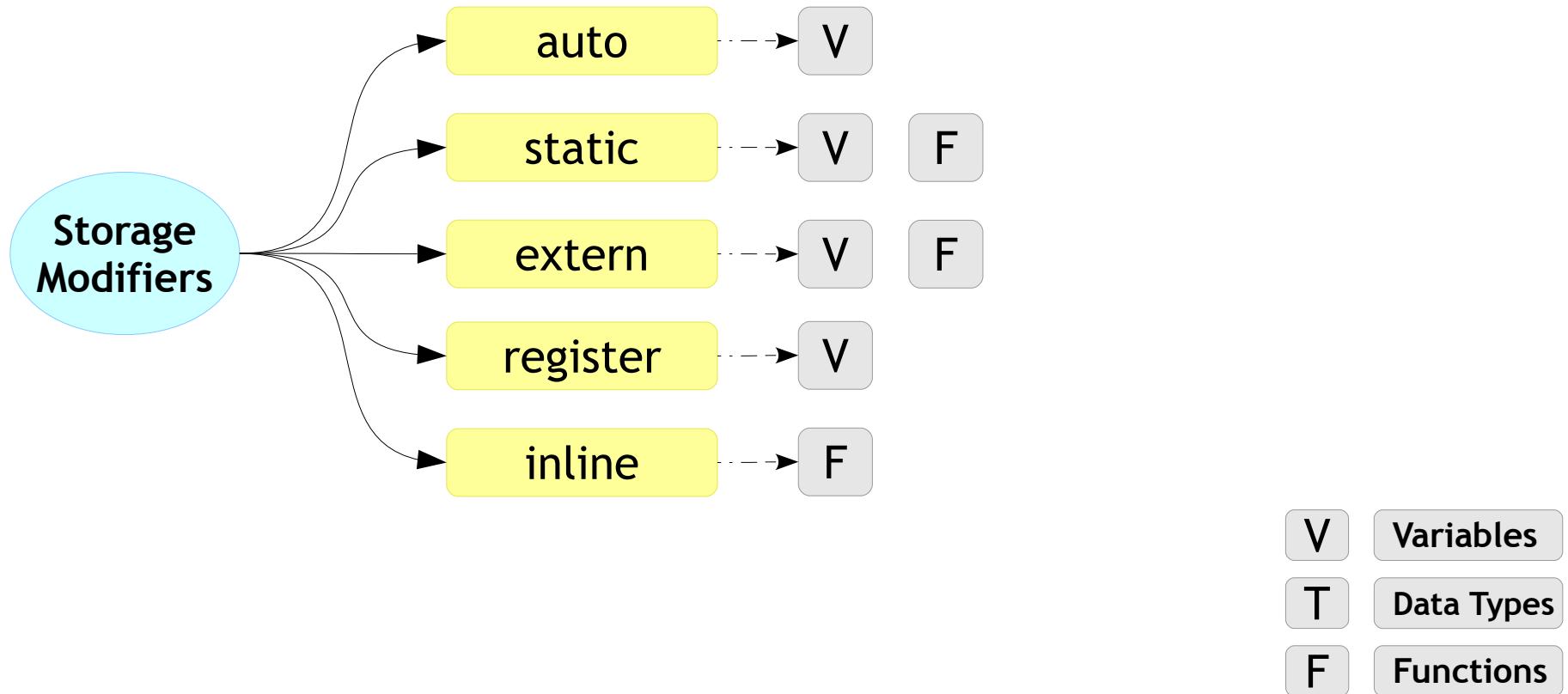
int main()
{
    unsigned int count1 = 10;
    signed int count2 = -1;

    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 (num1 > num3)
        {
            printf("Hello");
        }
        else
        {
            printf("World");
        }
    }
    ...
}
```

If conditional statement

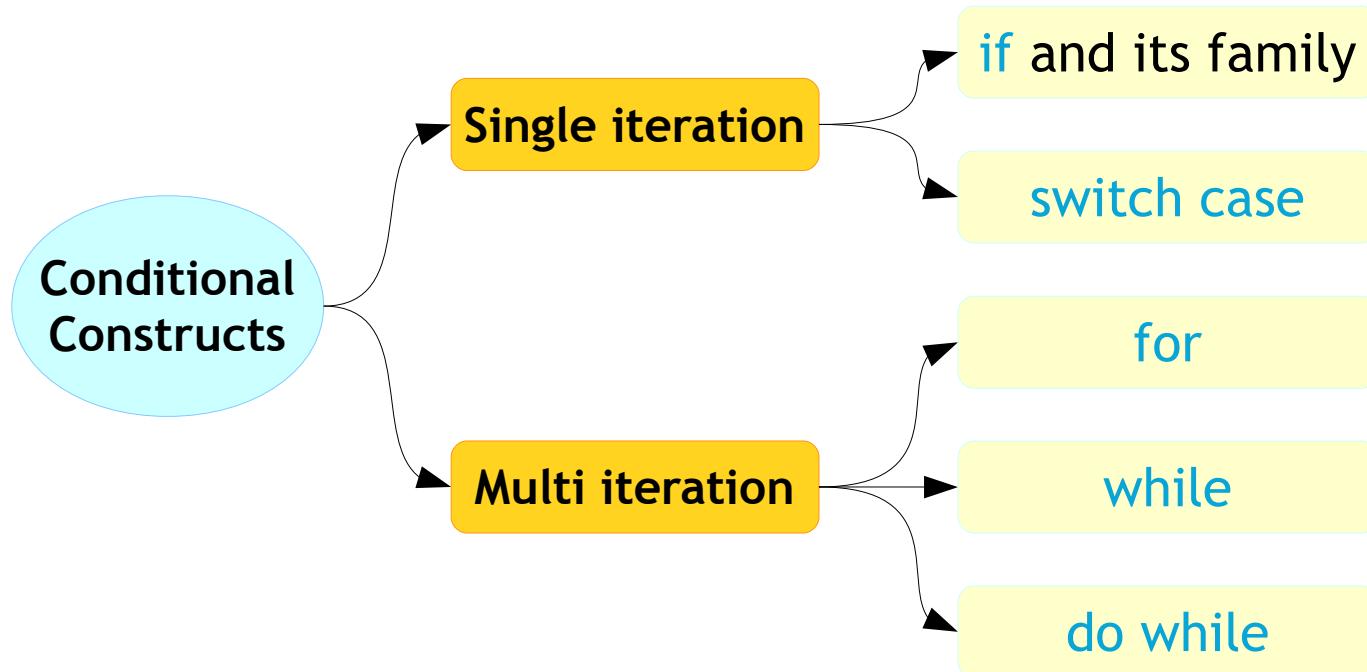
Nested if statement

Conditional Constructs



Advanced C

Conditional Constructs



Advanced C

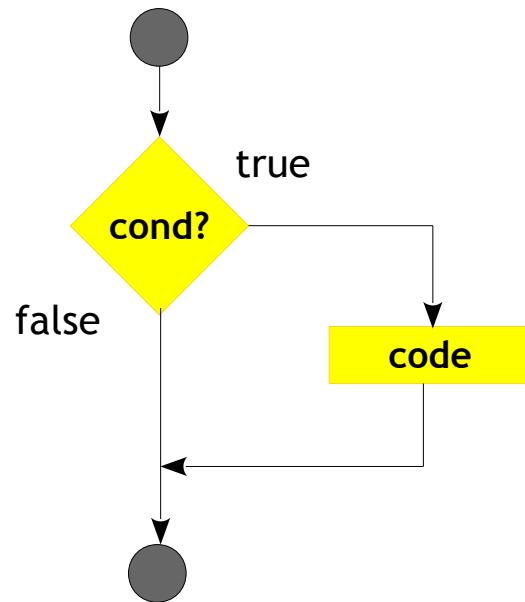
Conditional Constructs - if



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;
}
```

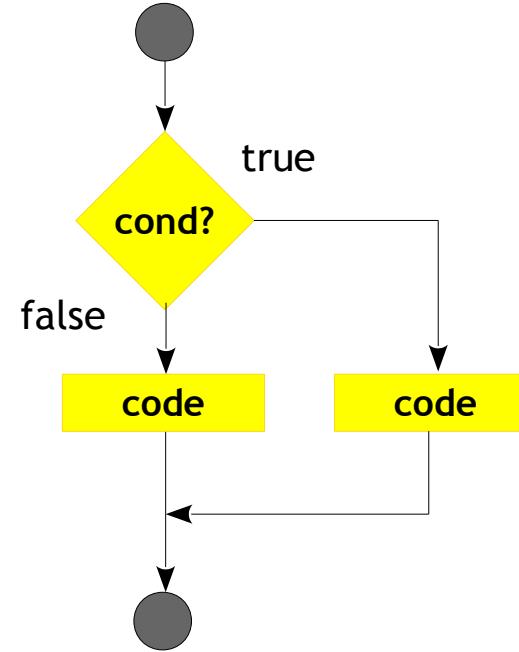
Advanced C

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;
}
```

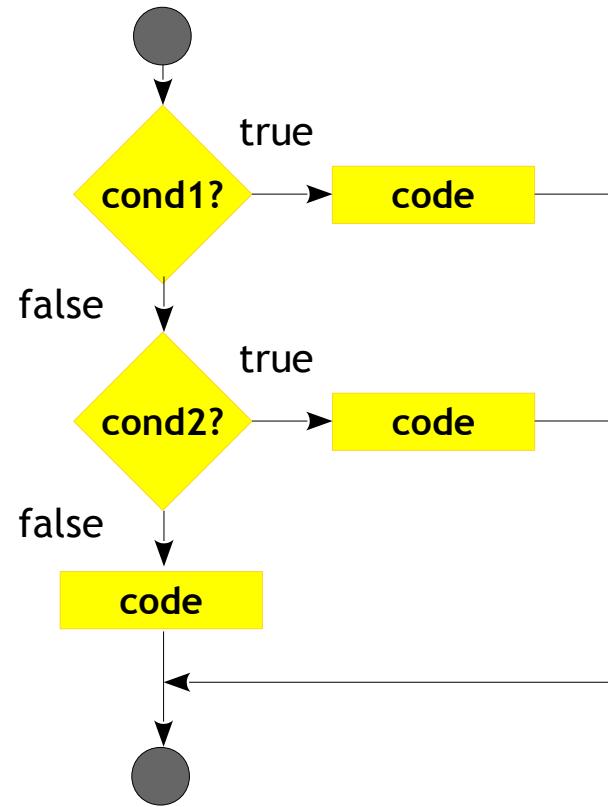
Advanced C

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

010_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
 - None of the above
- WAP to find the middle number (by value) of given 3 numbers



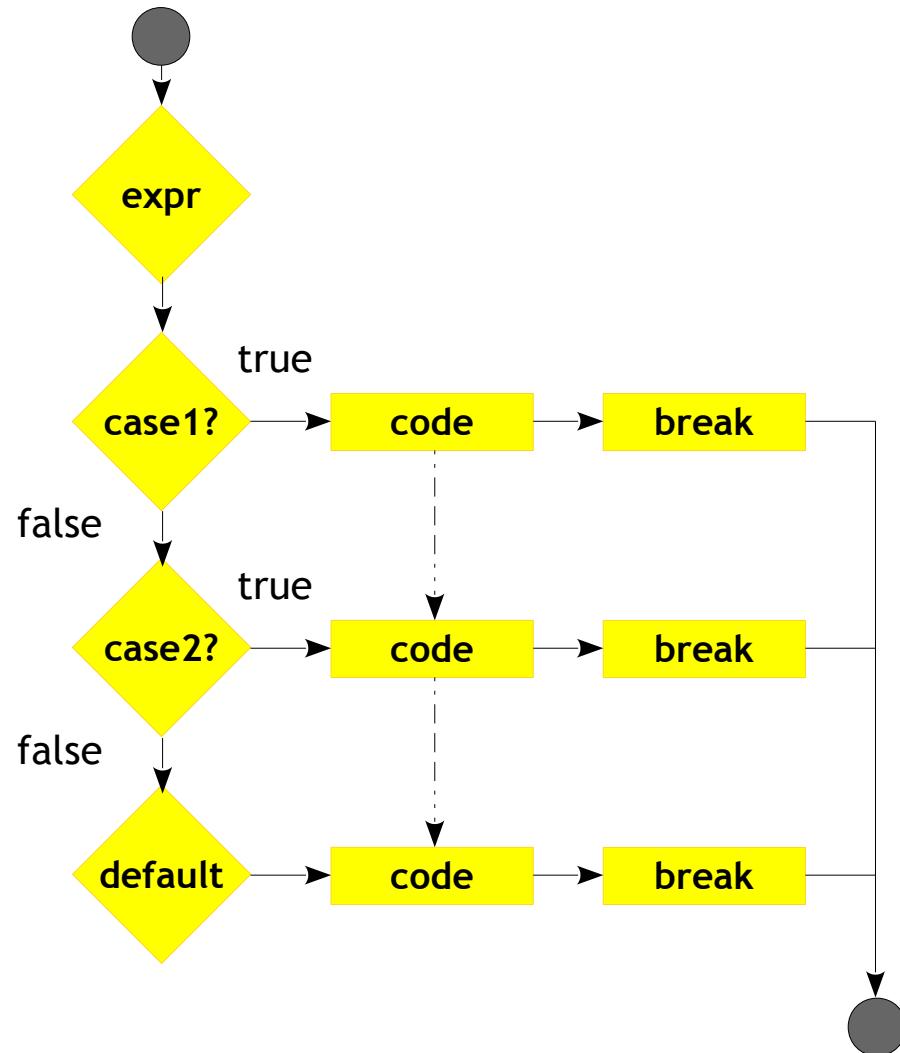
Advanced C

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

011_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

012_example.c

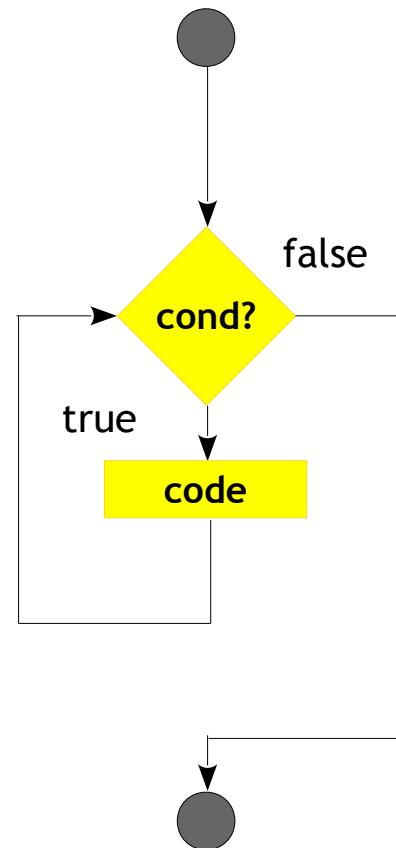
```
#include <stdio.h>

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

013_example.c

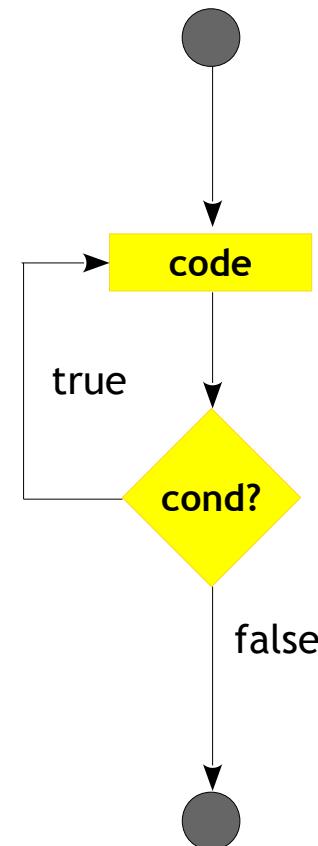
```
#include <stdio.h>

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

Syntax

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

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

014_example.c

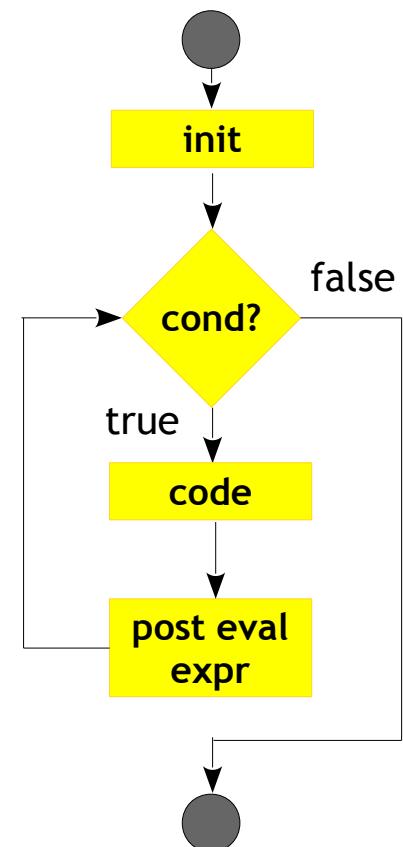
```
#include <stdio.h>

int main()
{
    int iter;

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

    return 0;
}
```

Flow



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

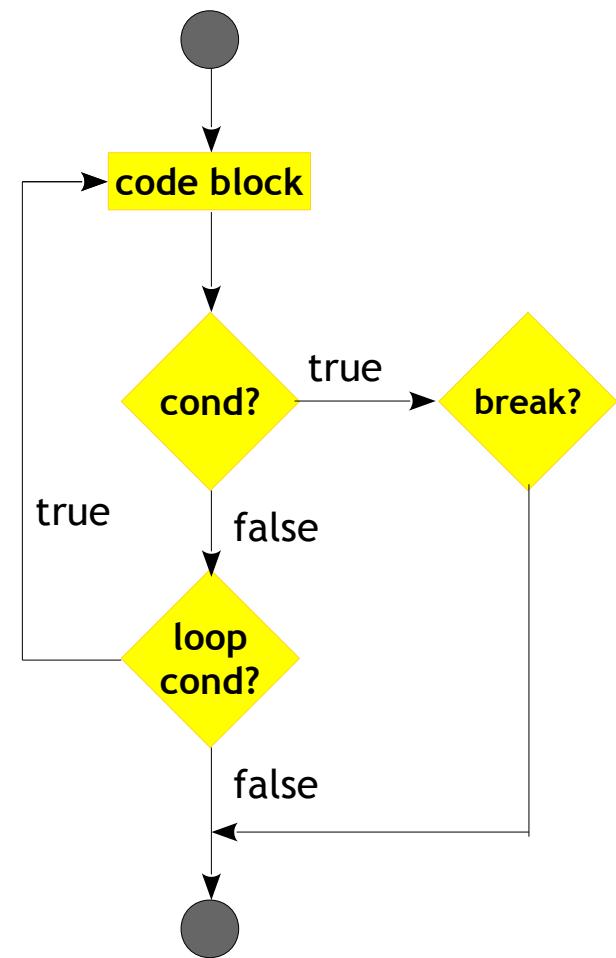
Conditional Constructs - break

- 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

015_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;
}
```

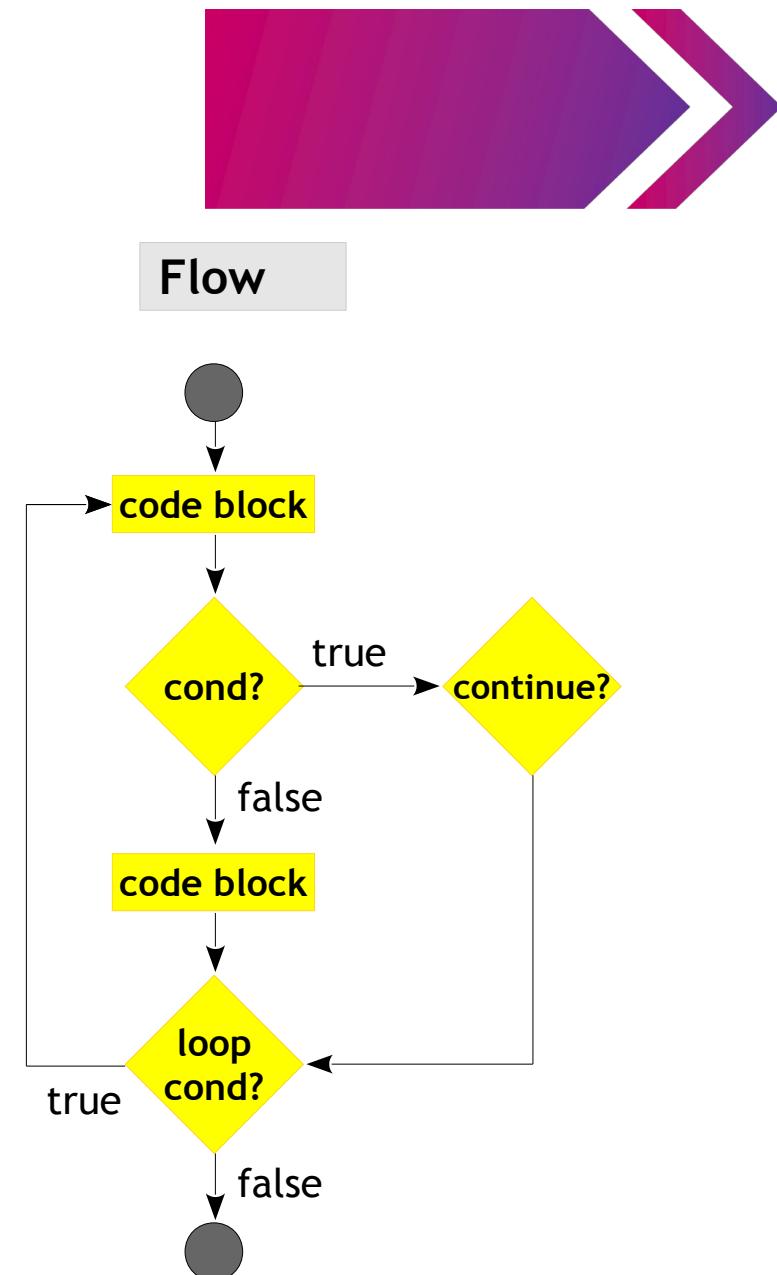
Advanced C

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);
```



Advanced C

Conditional Constructs - continue

016_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 an 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

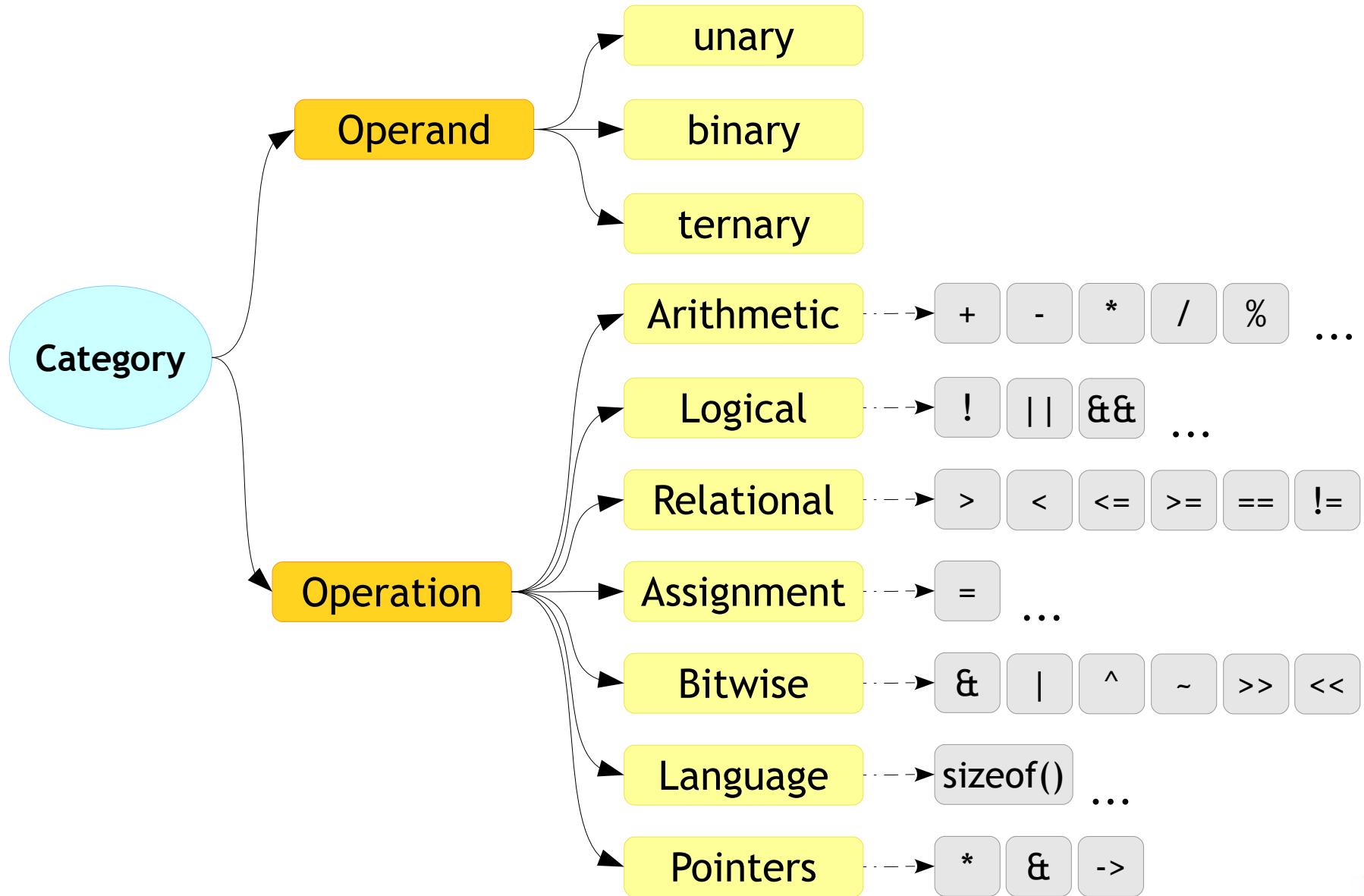


Advanced C Operators



- 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

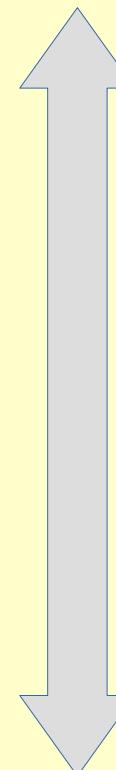
Advanced C Operators



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	

017_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()

018_example.c

```
#include <stdio.h>

int main()
{
    int num = 5;

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

    return 0;
}
```

019_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;
}
```

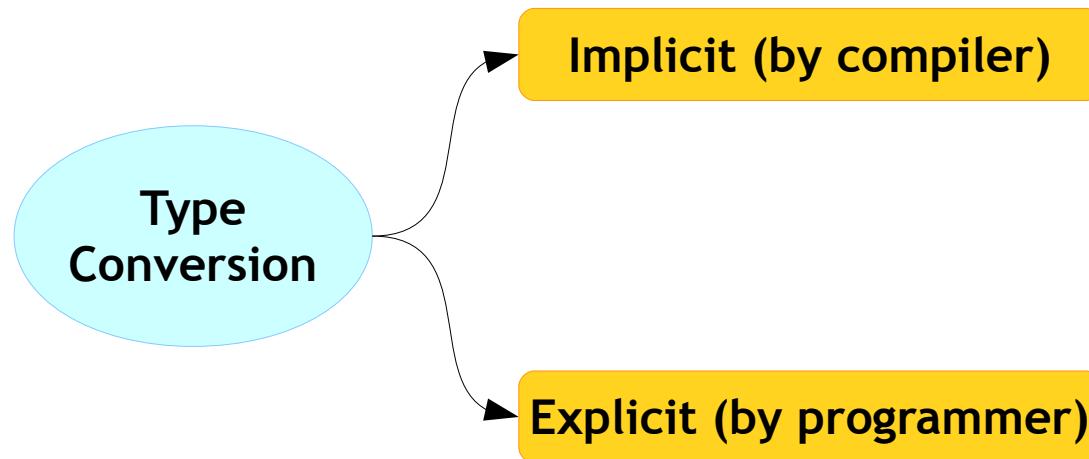
Advanced C

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

long double

double

float

unsigned long long

signed long long

unsigned long

signed long

unsigned int

signed int

unsigned short

signed short

unsigned char

signed char

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
```

020_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



021_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

022_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

023_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;
}
```

024_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



&

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

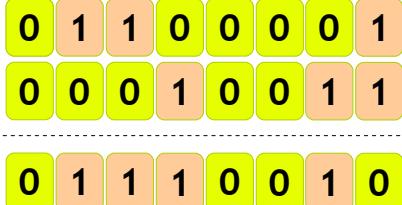
Operators - Bitwise



^

Bitwise XOR

Bitwise XORing of all the bits in two operands

Operand	Value
A	0x61
B	0x13
$A \wedge B$	0x72
	

~

Compliment

Complimenting all the bits of the operand

Operand	Value
A	0x61
$\sim A$	0x9E
	

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)



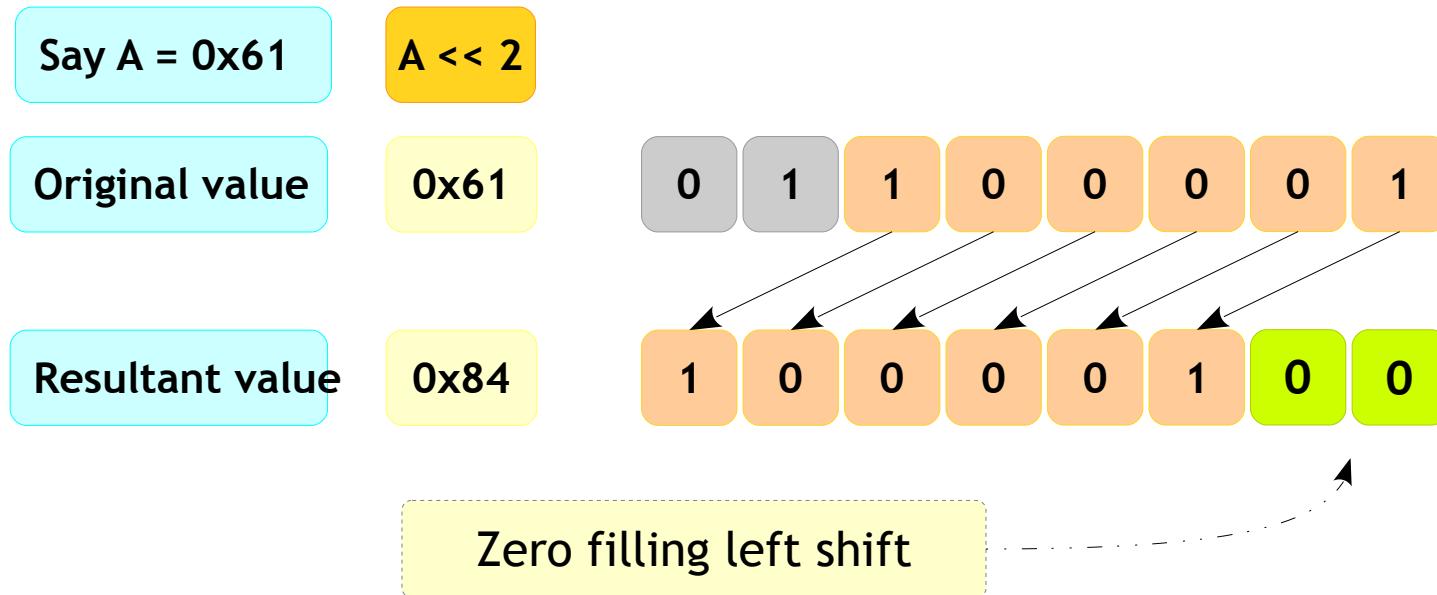
Advanced C

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



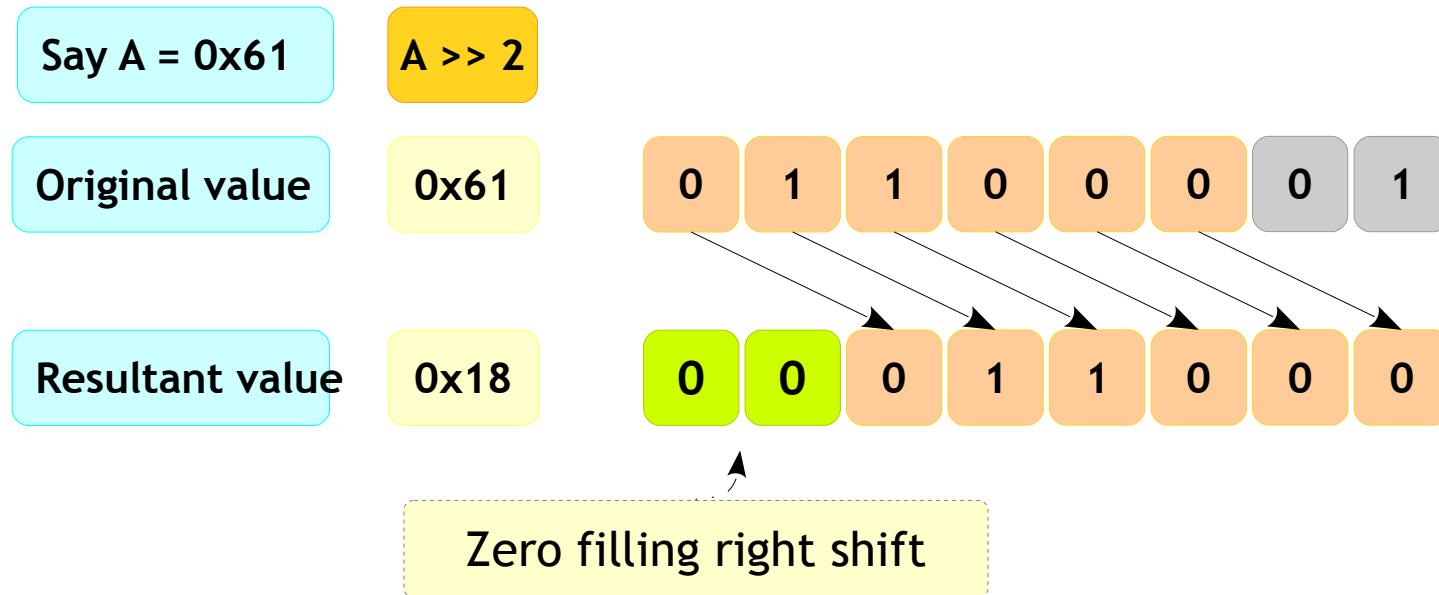
Advanced C

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

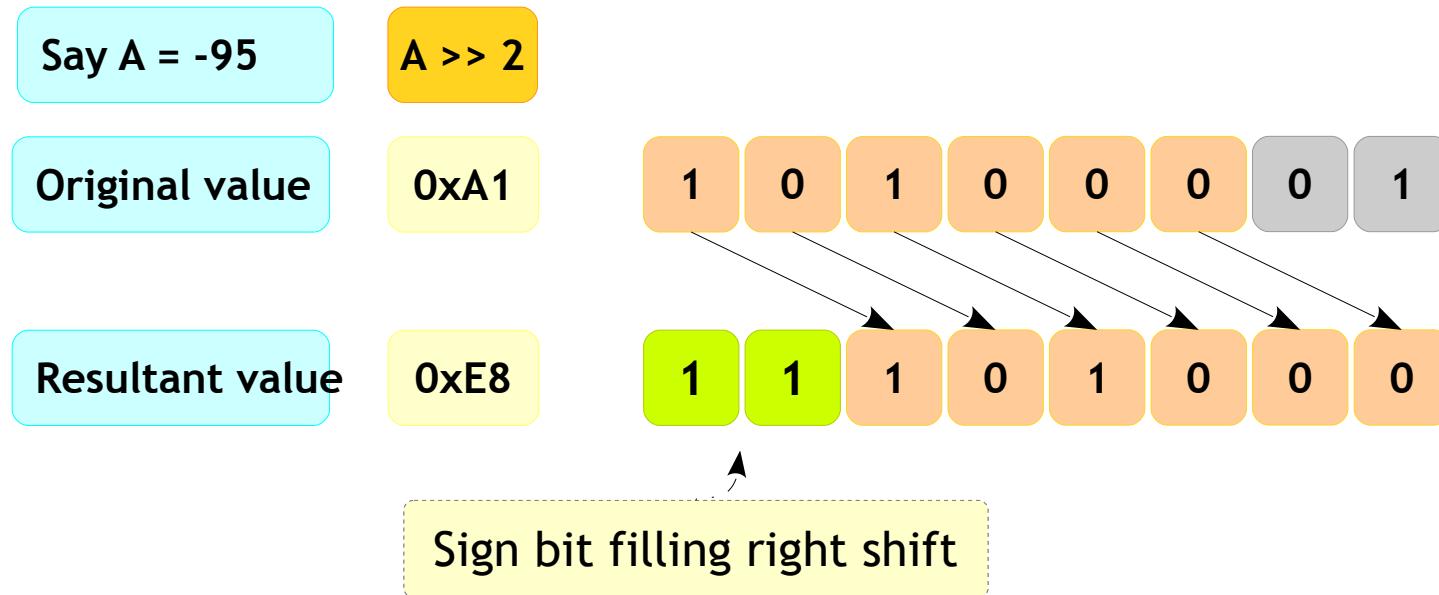


Advanced C

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

025_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

026_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

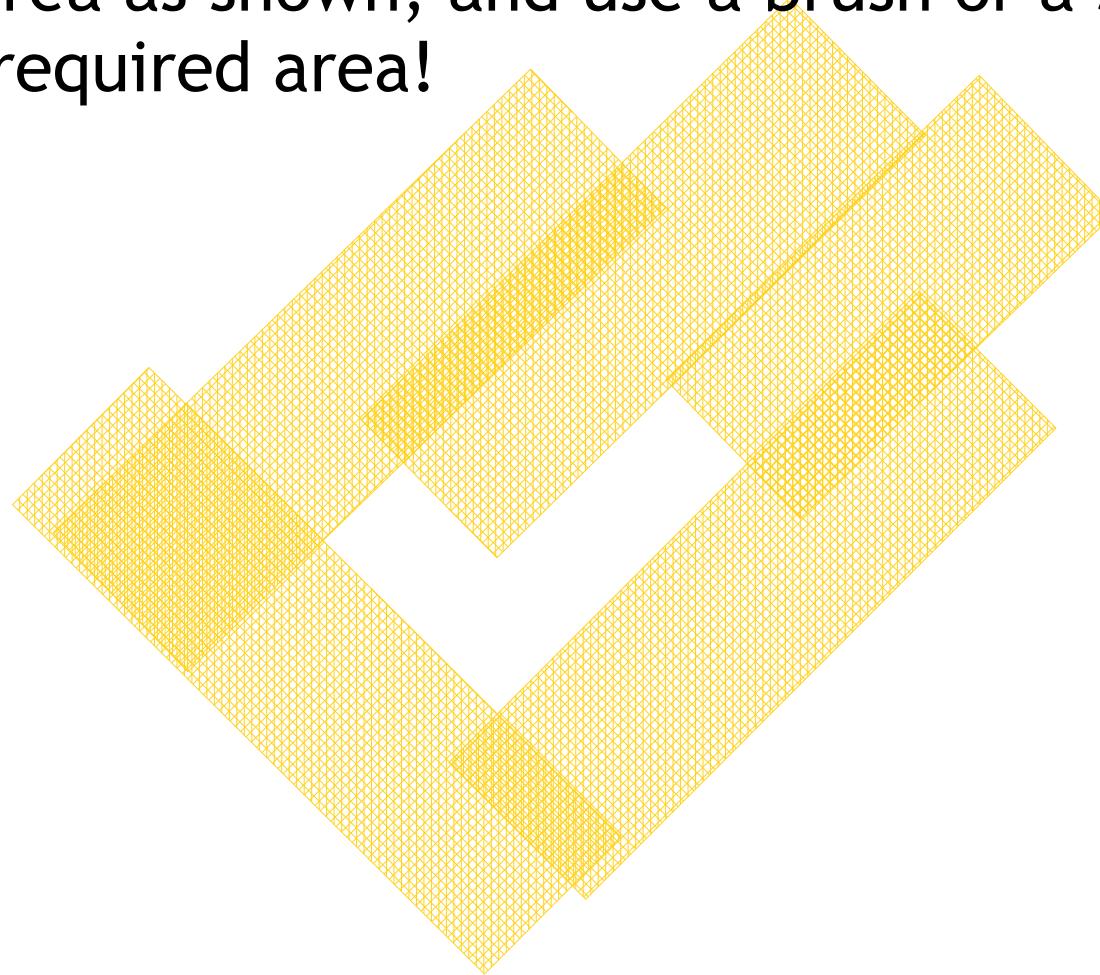
- If you want to create the below art assuming your are not a good painter, What would you do?



Advanced C

Operators - Bitwise - Mask

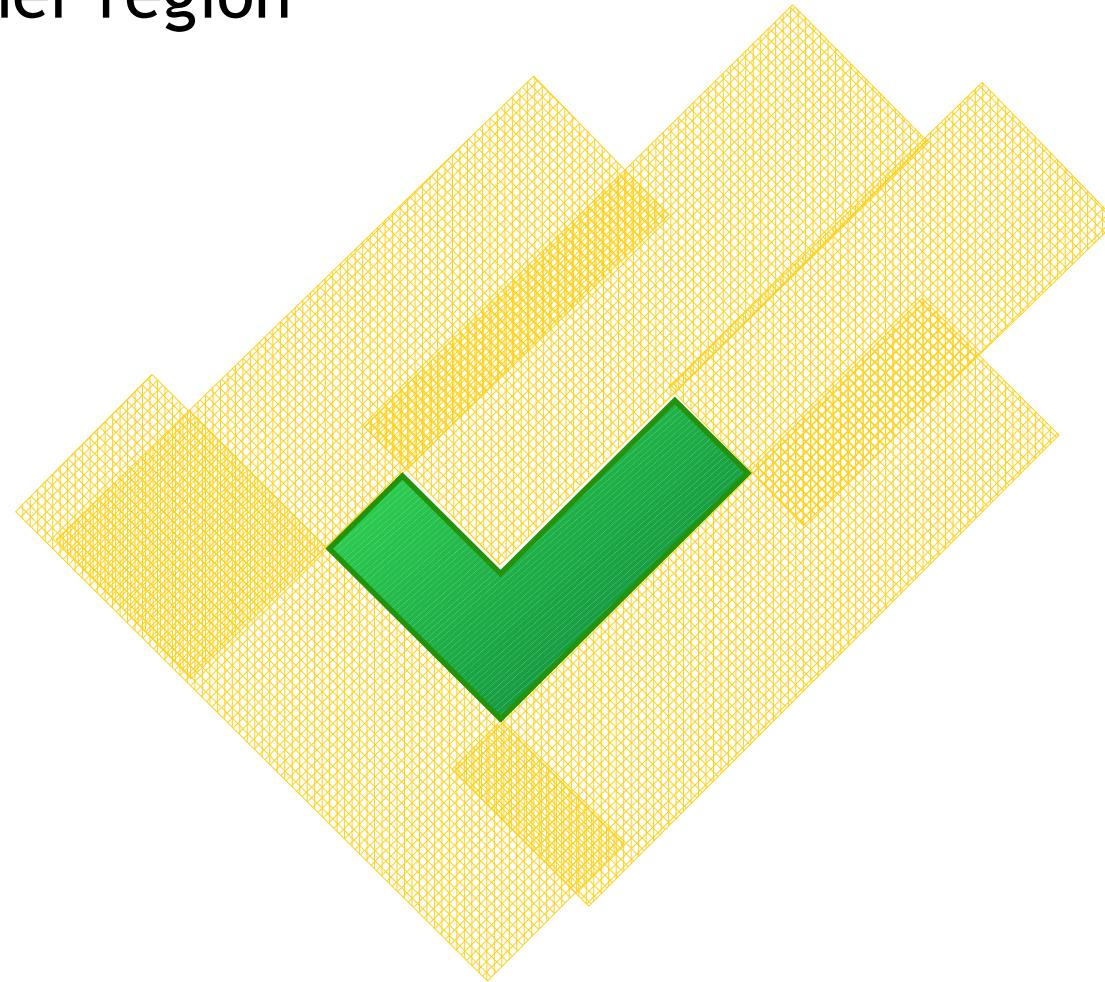
- Mask the area as shown, and use a brush or a spray paint to fill the required area!



Advanced C

Operators - Bitwise - Mask

- Fill the inner region



Advanced C

Operators - Bitwise - Mask



- Remove the mask tape



Advanced C

Operators - Bitwise - Mask

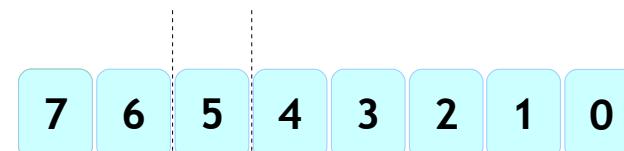
- So masking, technically means unprotecting the required bits of register and perform the actions like
 - Set Bit
 - Clear Bit
 - Get Bit
 - etc,..

Advanced C

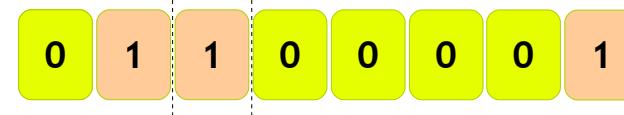
Operators - Bitwise - Mask



& Operator



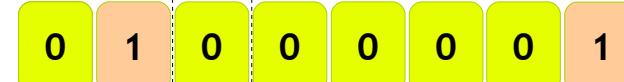
Bit Position



The register to be modified

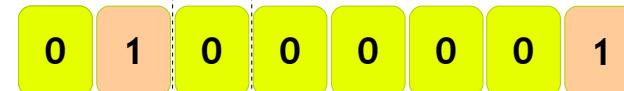


Mask to **CLEAR** 5th bit position



The result

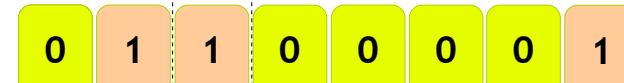
| Operator



The register to be modified



Mask to **SET** 5th 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;
```

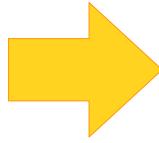
027_example.c

```
#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



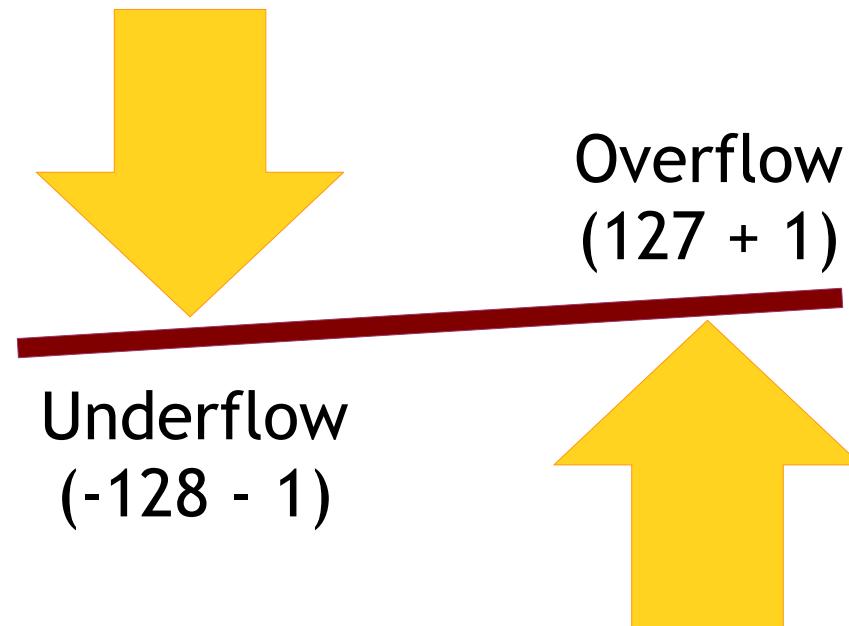
- The left operand of a comma operator is evaluated as a void expression (result discarded)
- Then the right operand is evaluated; the result has its type and value
- Comma acts as separator (not an operator) in following cases -
 - Arguments to function
 - Lists of initializers (variable declarations)
- 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

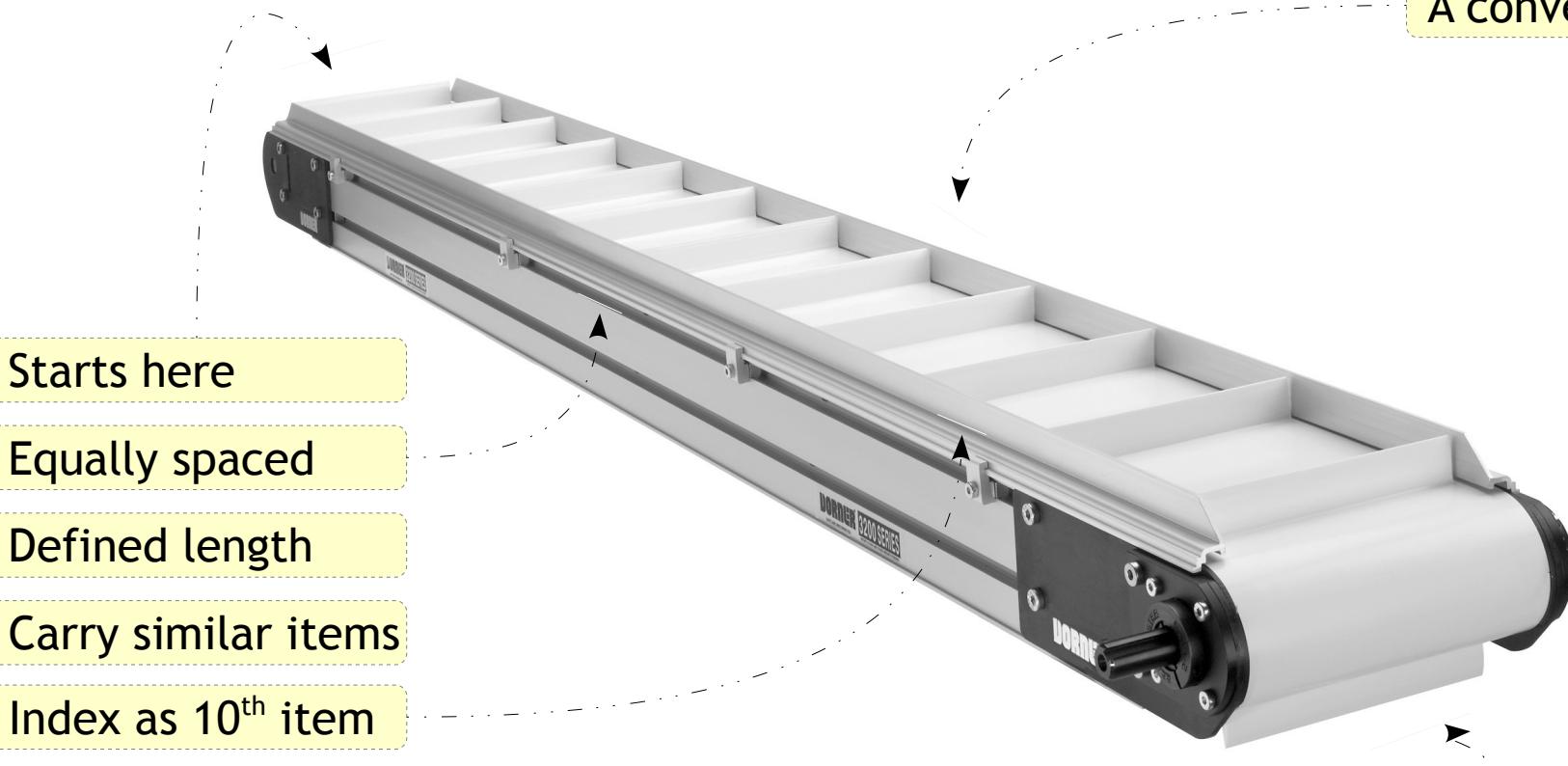
Arrays



Advanced C

Arrays - Know the Concept

A conveyor belt



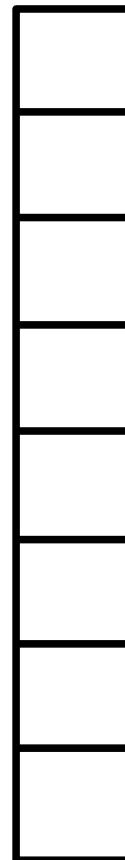
Advanced C

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

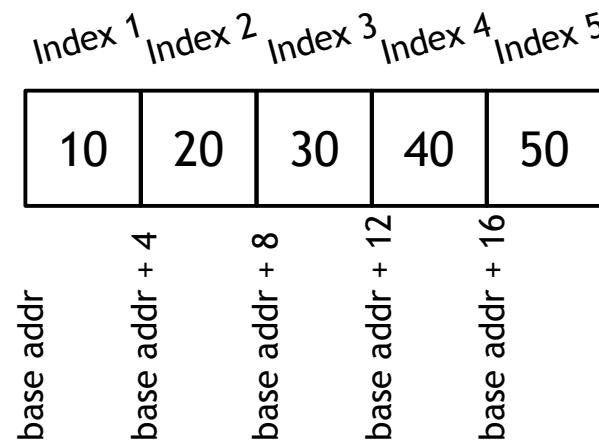
```
data_type name[SIZE];
```

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?

028_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

029_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

030_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

031_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?

032_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?



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.

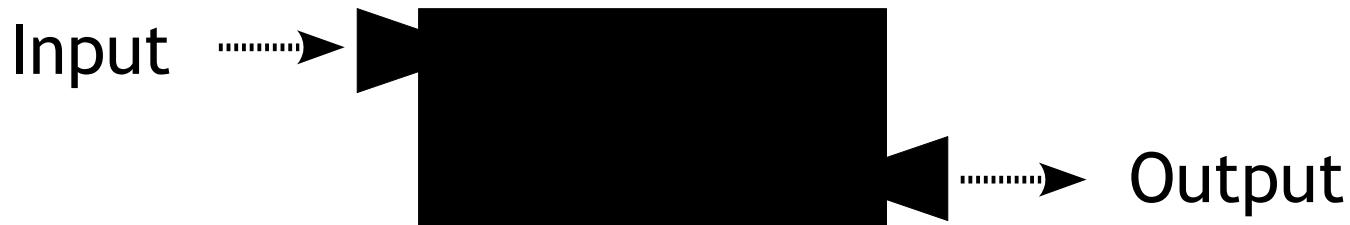
"the function $(bx + c)$ "

Source: Google

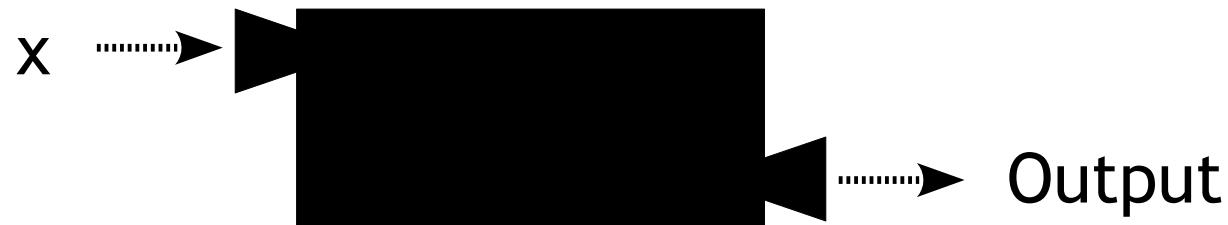
- In programming languages it can be something which performs a specific service
- Generally a function has 3 properties
 - Takes Input
 - Perform Operation
 - Generate Output

Advanced C

Functions - What?



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



$$x = 2$$



$$2 + 1$$



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

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

Return data type as int

First parameter with int type

Second parameter with int type

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

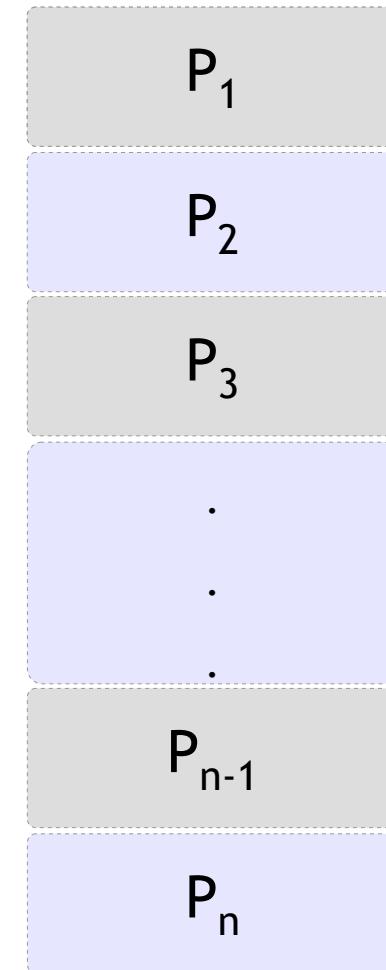
Advanced C

Function and the Stack

Linux OS



User Space



The User space contains many processes

Every process will be scheduled by the kernel

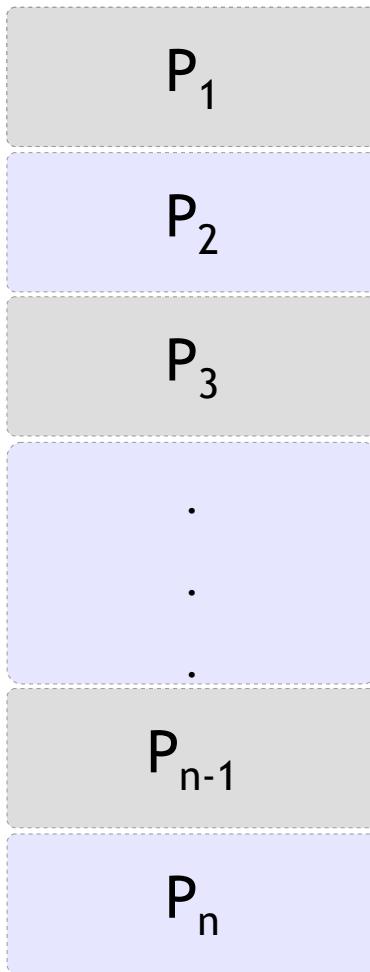
Each process will have its memory layout discussed in next slide

Advanced C

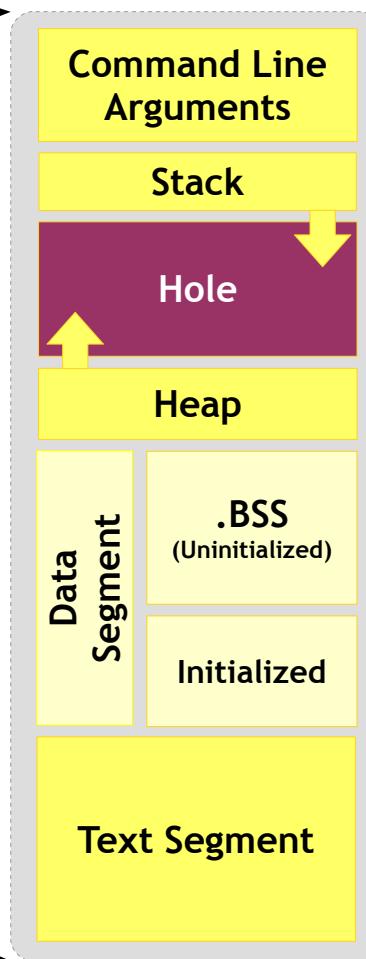
Function and the Stack



User Space



Memory Segments



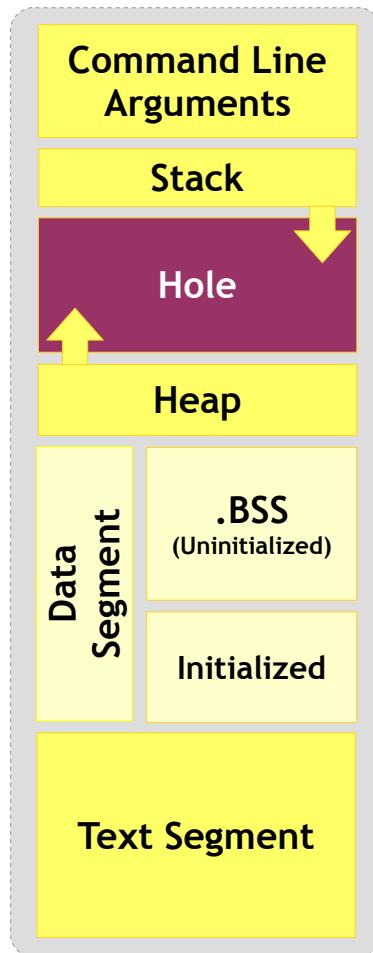
The memory segment of a program contains four major areas.

- 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

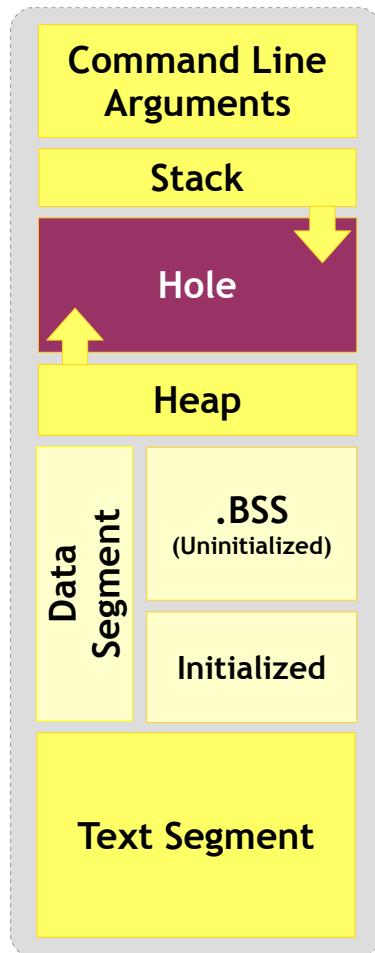
A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack

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

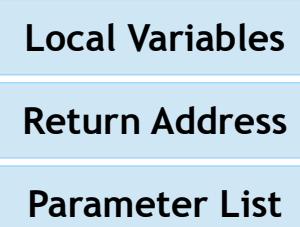
Advanced C

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(s)

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">This method copies the actual value of an argument into the formal parameter of the function.In this case, changes made to the parameter inside the function have no effect on the actual argument.	<ul style="list-style-type: none">This method copies the address of an argument into the formal parameter.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.

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;
}

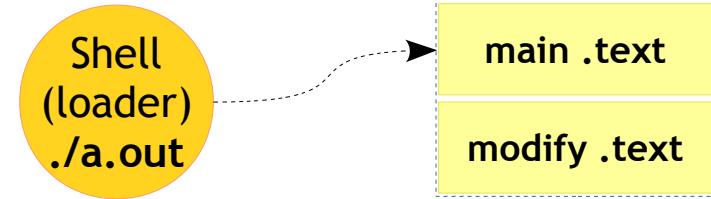
int main()
{
    int num = 10;

    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;
}

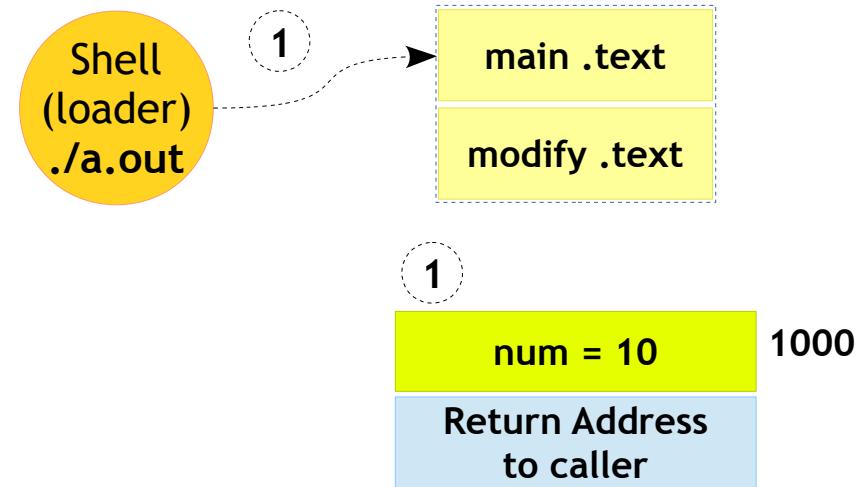
int main()
{
    int num = 10;

    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;
}

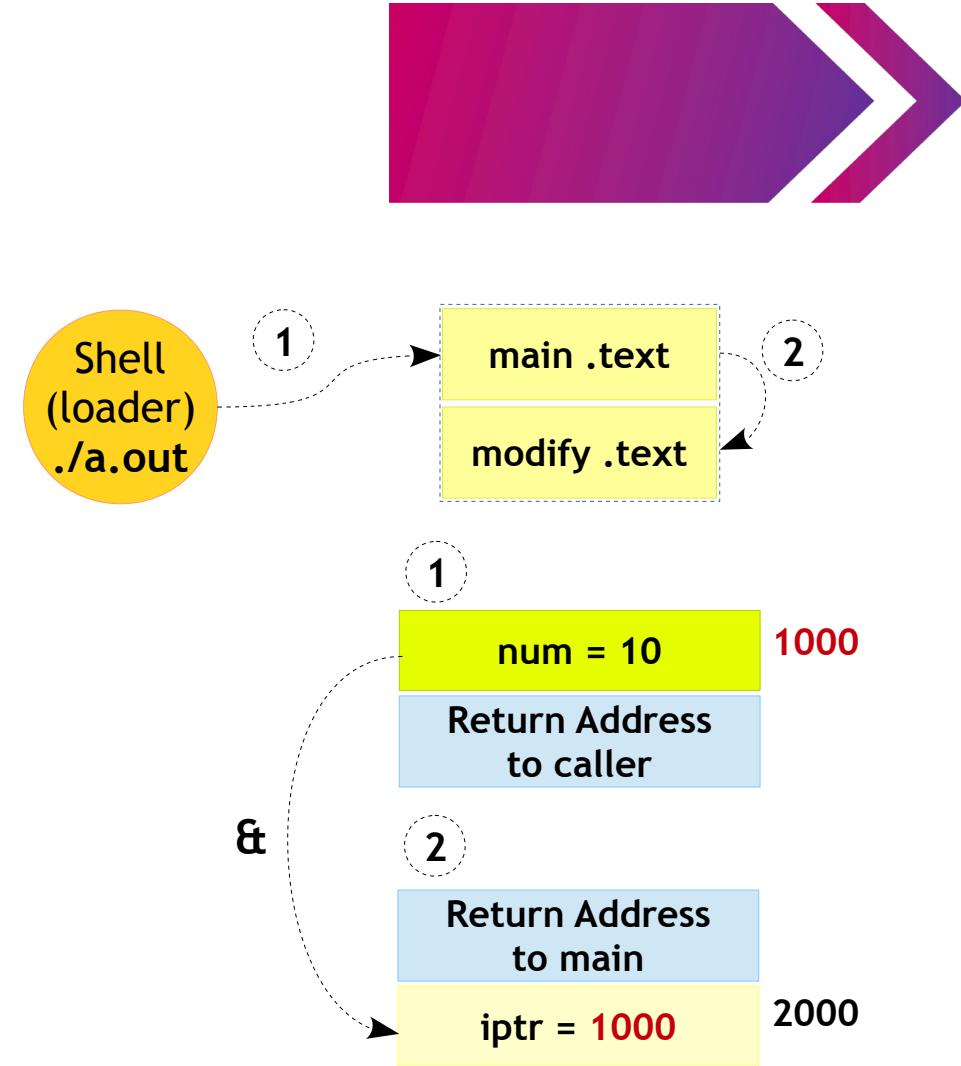
int main()
{
    int num = 10;

    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;
}

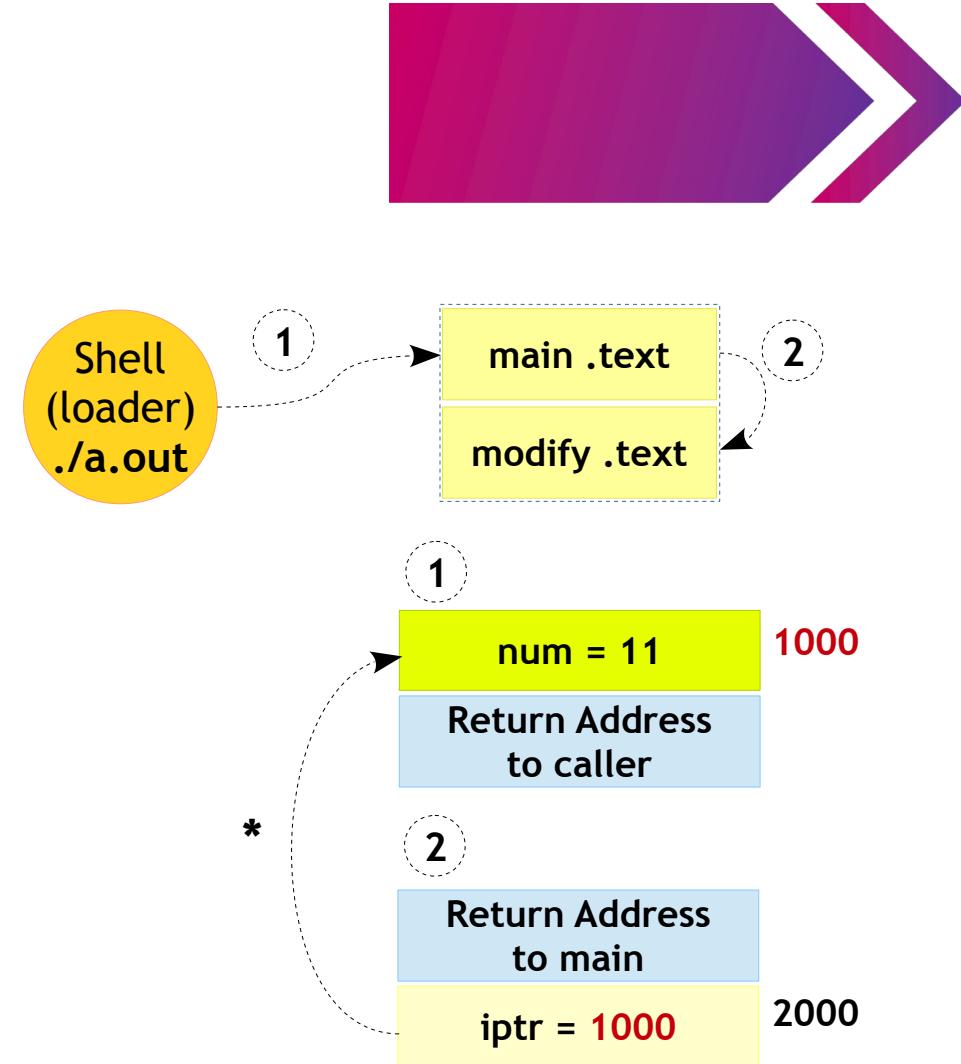
int main()
{
    int num = 10;

    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 - Implicit int rule

006_example.c

```
#include <stdio.h>

int main()
{
    dummy(20);

    return 0;
}

/* minimum valid function */
dummy()
{
}
```

- Compilers can assume that return and function parameter types are integers
- The rule was introduced in C89/90
- This rule is discontinued in C99
- But, compilers still follow the above rule to maintain backward compatibility

Advanced C

Functions - Prototype - What?

- Function prototype is signature of function specifying
 - Number of function parameters and their types
 - Return type of function

Advanced C

Functions - Prototype - Why?



- Need of function prototype -
 - Functions can be used in many files
 - Functions can be compiled and added to library for reuse purpose
 - Compiler needs to know the signature of function before it comes across the invocation of function
 - In absence of prototype, compilers will apply “Implicit int rule” which might lead to discrepancy with function parameters and return type in actual definition

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

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[iter]);
    }
}
```

Advanced C

Functions - Passing Array

008_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

009_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

010_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

011_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 - Local Return

012_example.c

```
#include <stdio.h>

int *func(void)
{
    int a = 10;

    return &a;
}

int main()
{
    int *ptr;

    ptr = func();

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

    return 0;
}
```

Advanced C

Functions - Void Return

013_example.c

```
#include <stdio.h>

void func(void)
{
    printf("Welcome!\n");

    return; // Use of return is optional
}

int main()
{
    func();

    return 0;
}
```

Advanced C

Functions - Void Return

014_example.c

```
#include <stdio.h>

int main()
{
    printf("%s\n", func()); // Error, invalid use of a function returning void

    return 0;
}

void func(void)
{
    char buff[] = "Hello World";

    return buff; // some compilers might report error in this case
}
```

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

015_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;

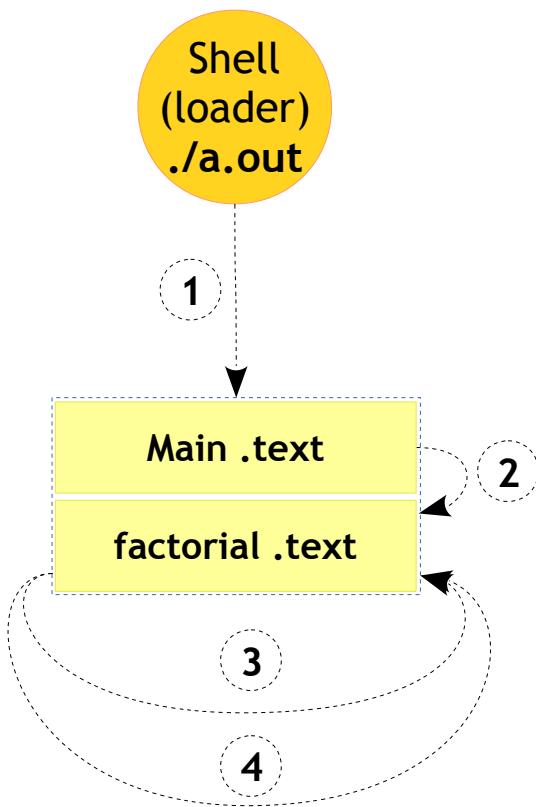
    ret = factorial(3);
    printf("Factorial of 3 is %d\n", ret);

    return 0;
}
```

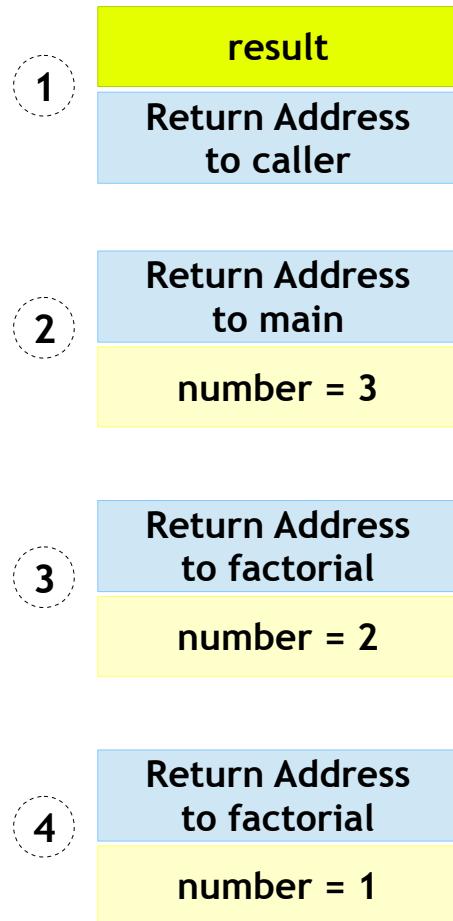
n	$!n$
0	1
1	1
2	2
3	6
4	24

Embedded C

Functions - Recursive - Example Flow



Stack Frames



Value with calls

factorial(3)

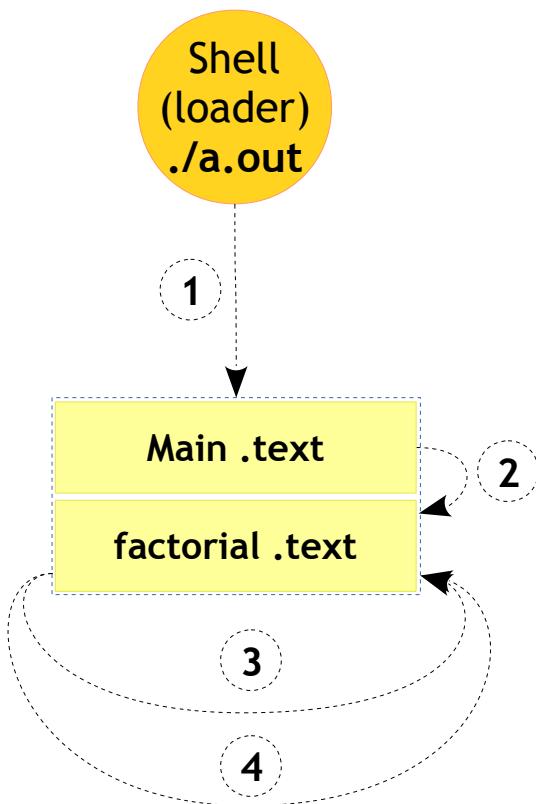
number != 1
number * factorial(number -1)
3 * factorial(3 -1)

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

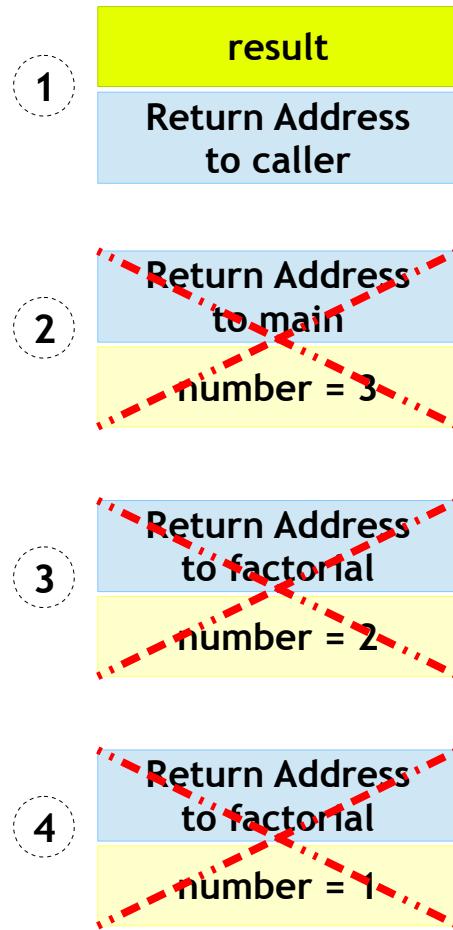
number == 1

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 numbers starting from 1
- Write a program to find x raise to the power of y (x^y)
 - Example : $2^3 = 8$
- Write a program to find the sum of digits of a given number
 - Example : if given number is 10372, the sum of digits will be $1+0+3+9+2 = 15$

Standard I/O Functions



Pointers



Advanced C

Pointers - Jargon

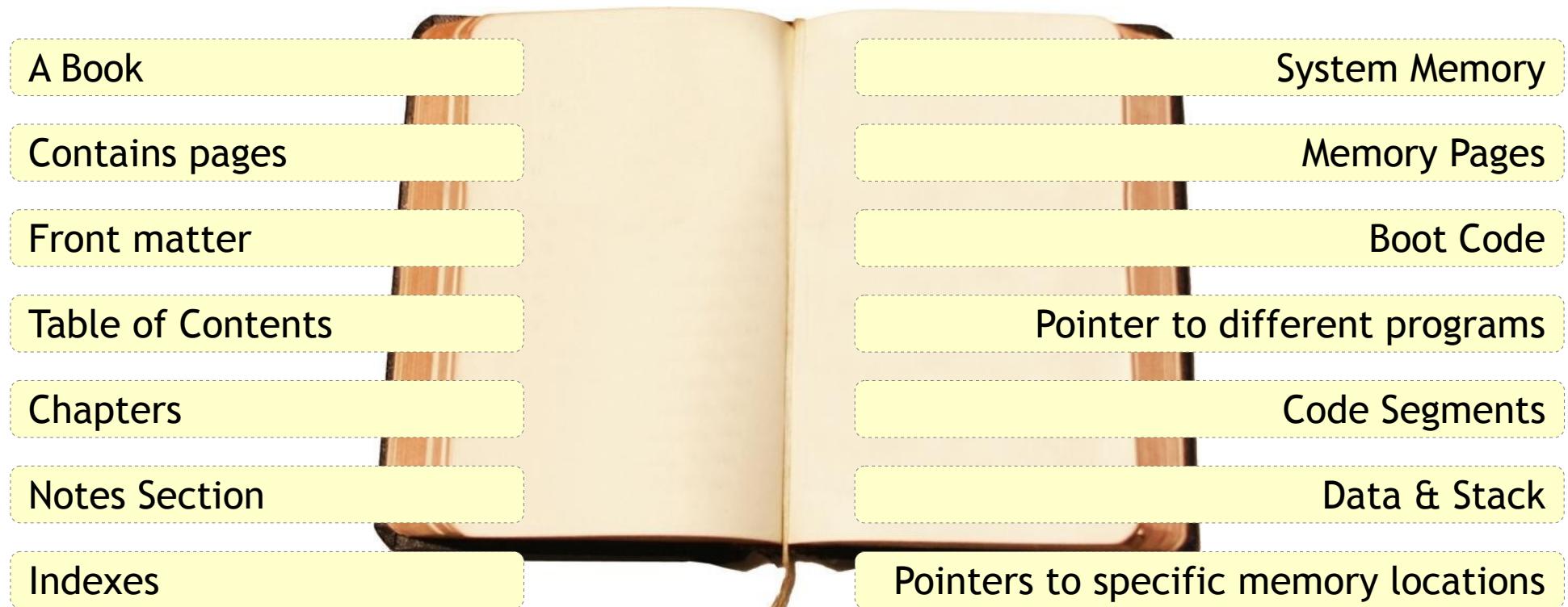


- What's a Jargon?
 - Jargon may refer to terminology used in a certain profession, such as computer jargon, or it may refer to any nonsensical language that is not understood by most people.
 - Speech or writing having unusual or pretentious vocabulary, convoluted phrasing, and vague meaning.
- Pointer 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

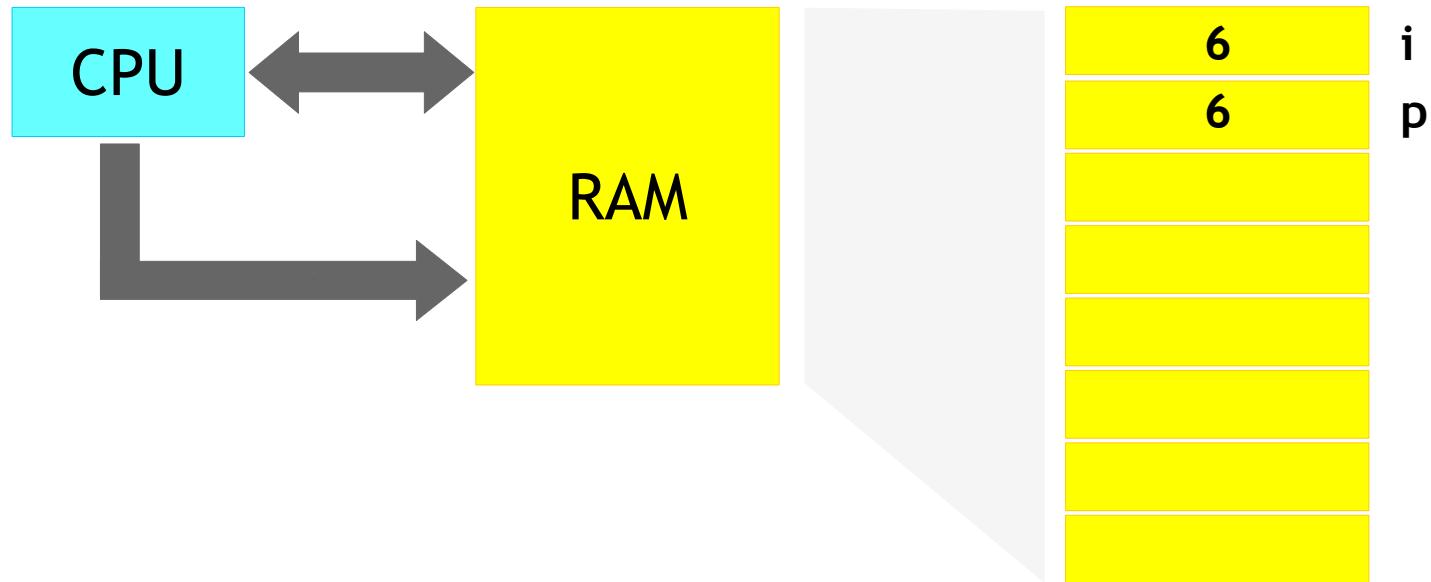
Advanced C

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

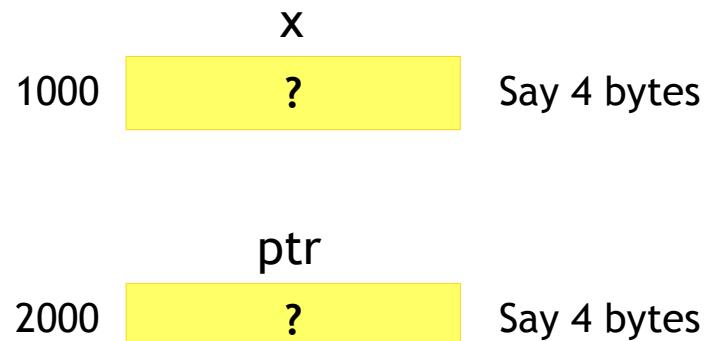
001_example.c

```
#include <stdio.h>

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

    x = 5;
    ptr = &x;

    return 0;
}
```



Advanced C

Pointers - Rule 1 in detail

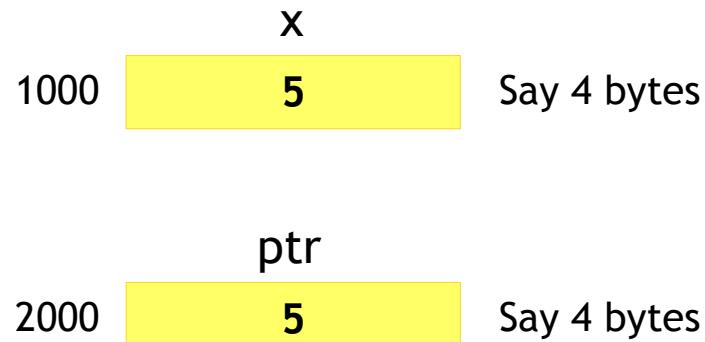
001_example.c

```
#include <stdio.h>

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

    → x = 5;
    → ptr = 5;

    return 0;
}
```



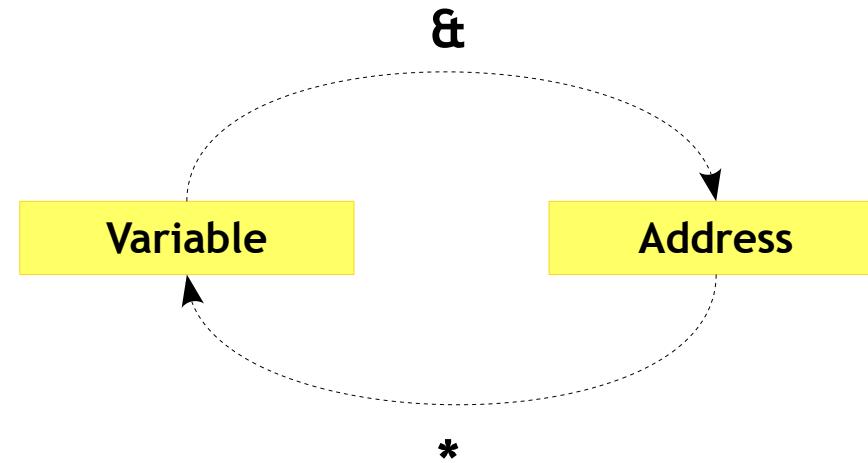
- So pointer is an integer
- But remember the “They may not be of same size”
 - 32 bit system = 4 Bytes
 - 64 bit system = 8 Bytes

Advanced C

Pointers - The 7 Rules - Rule 2



- Rule : “Referencing and Dereferencing”



Advanced C

Pointers - Rule 2 in detail

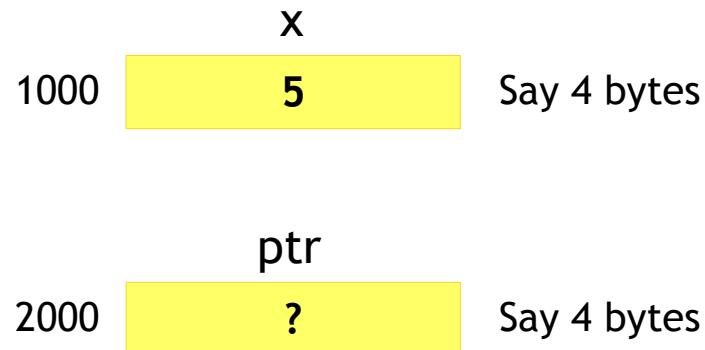
002_example.c

```
#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

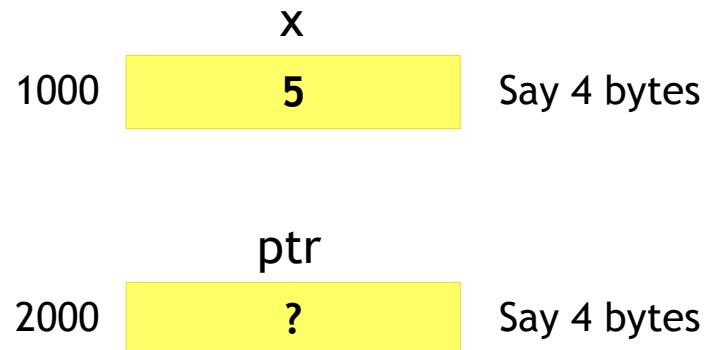
002_example.c

```
#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

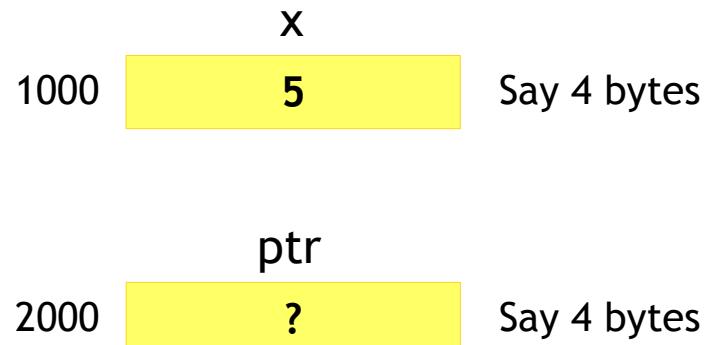
002_example.c

```
#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

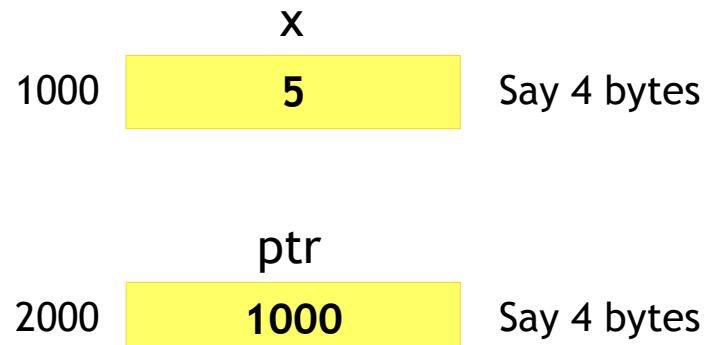
002_example.c

```
#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

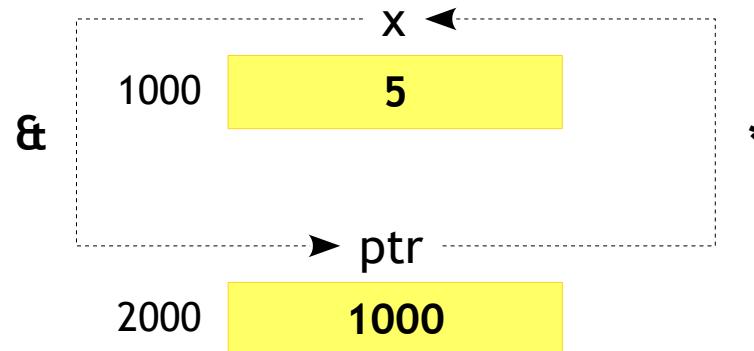
002_example.c

```
#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”

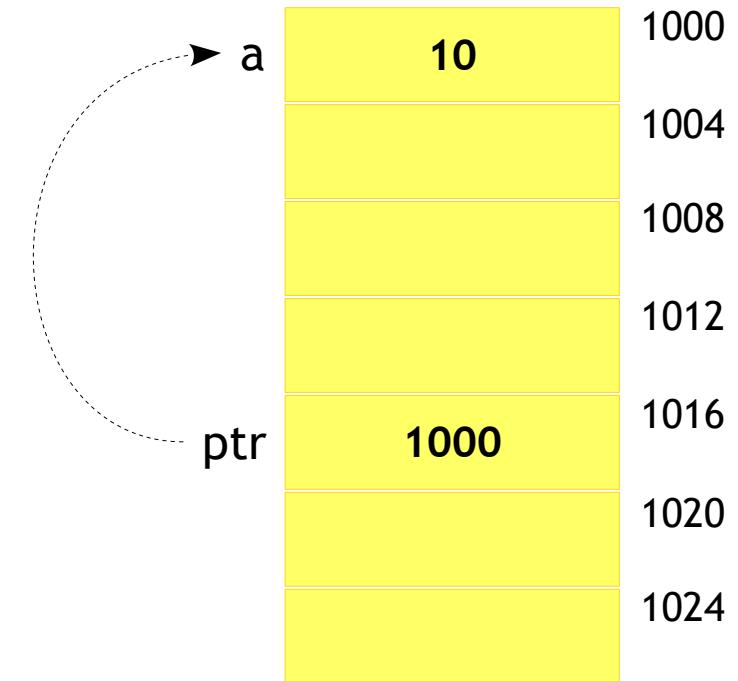
Example

```
#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

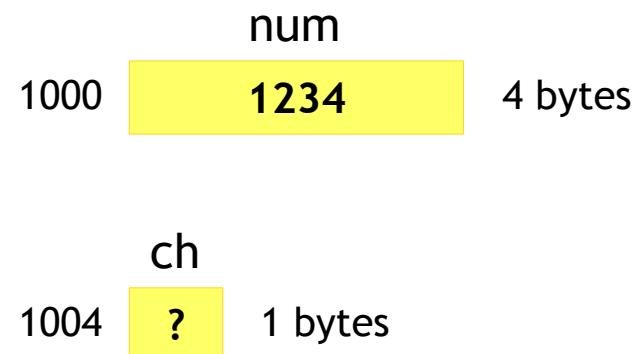
- Does address has a type?

Example

```
#include <stdio.h>

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

    return 0;
}
```



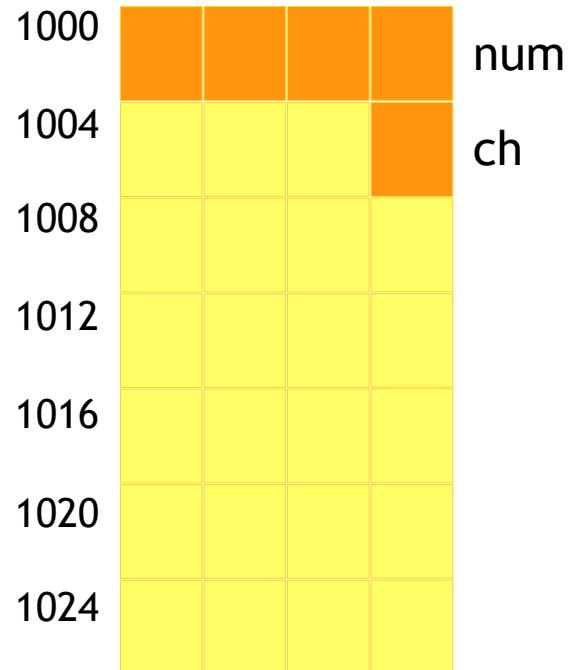
- So from the above diagram can we say $\&\text{num} \rightarrow 4 \text{ bytes}$ and $\&\text{ch} \rightarrow 1 \text{ byte}$?

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

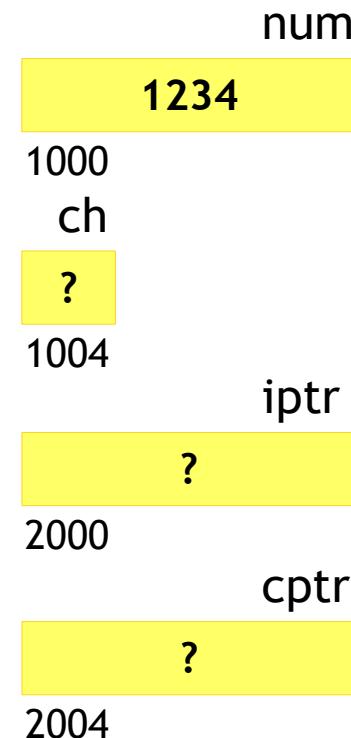
Example

```
#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



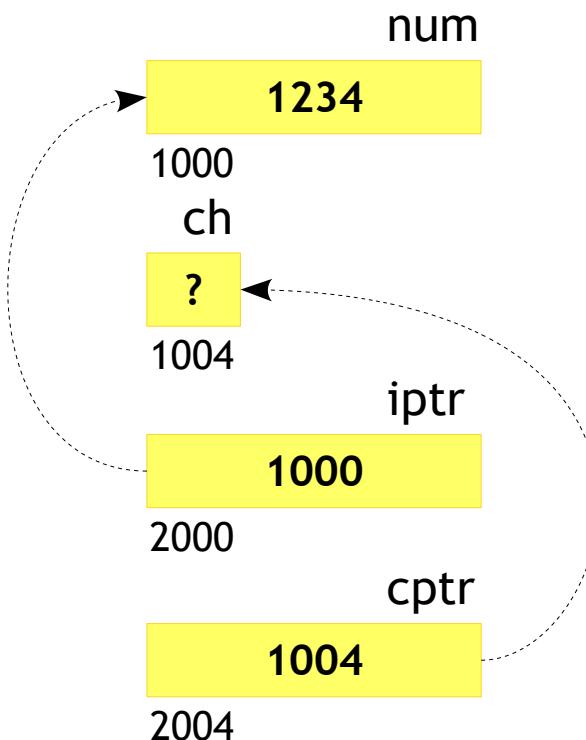
Example

```
#include <stdio.h>

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

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

    return 0;
}
```

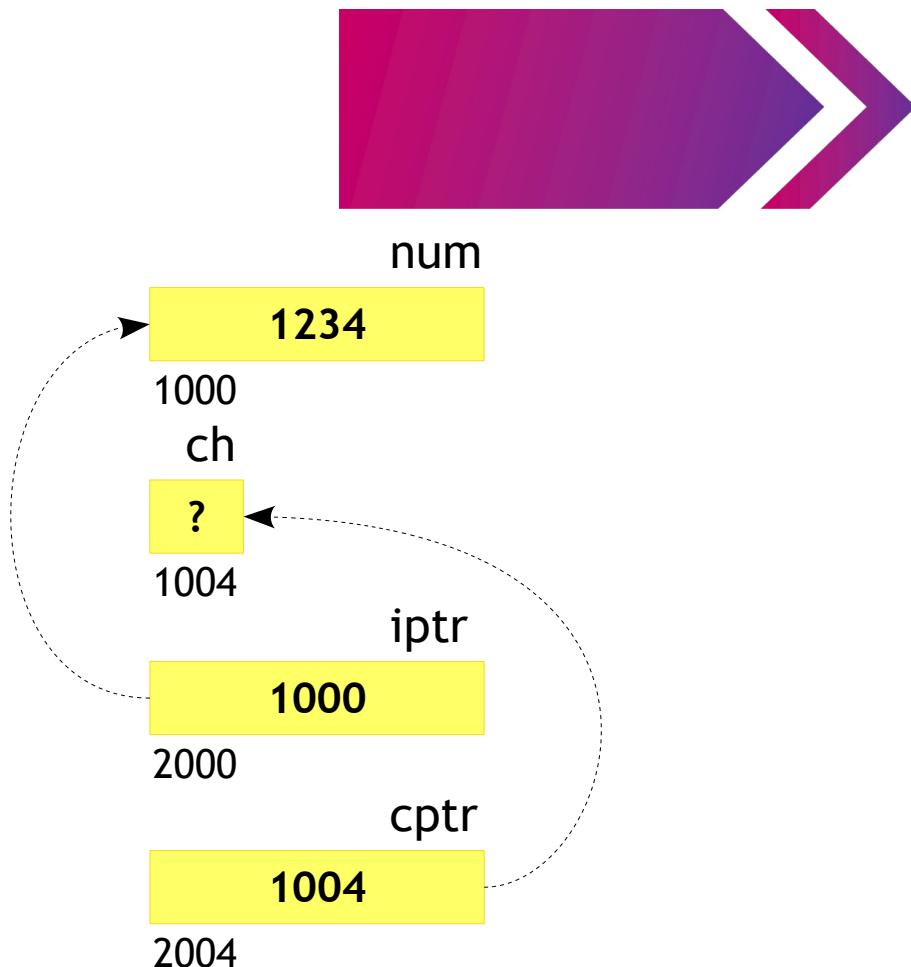


- Lets consider the above examples to understand it
- Say we have a integer and a character pointer

Advanced C

Pointers - Rule 4 in detail

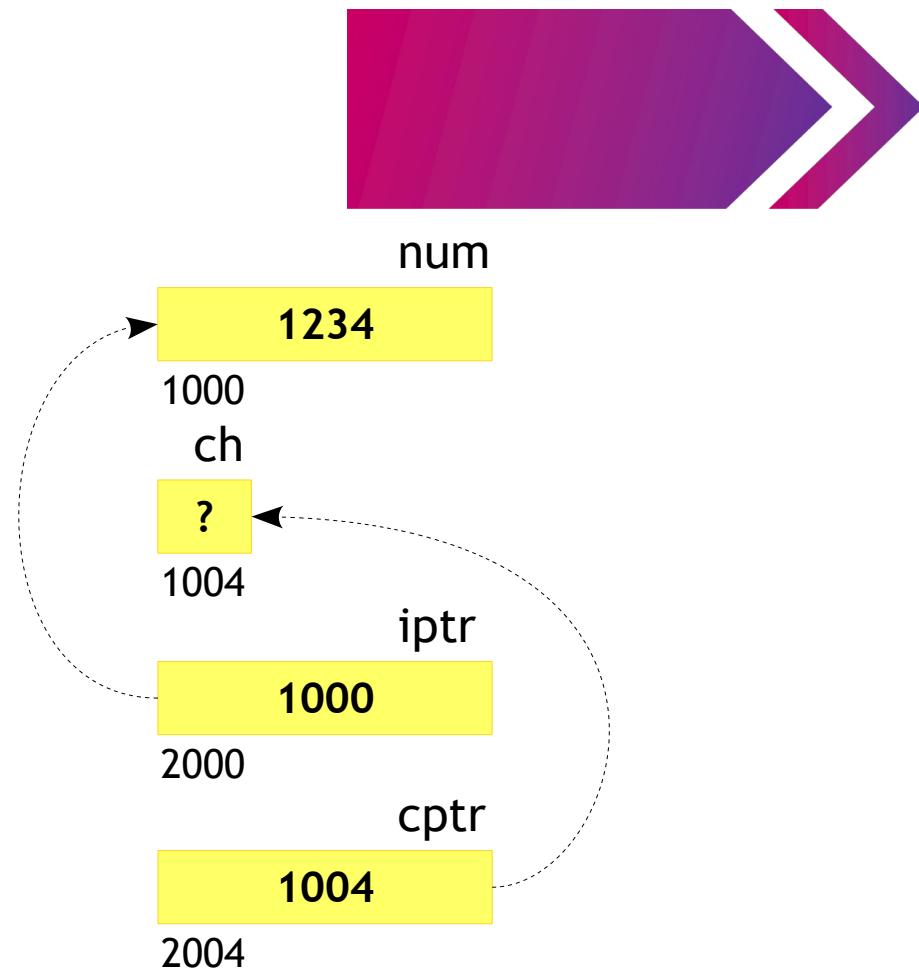
- With just the address, can we know what data is stored?
- How would we know how much data to fetch for the address it is pointing to?
- Eventually the answer would be NO!!



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

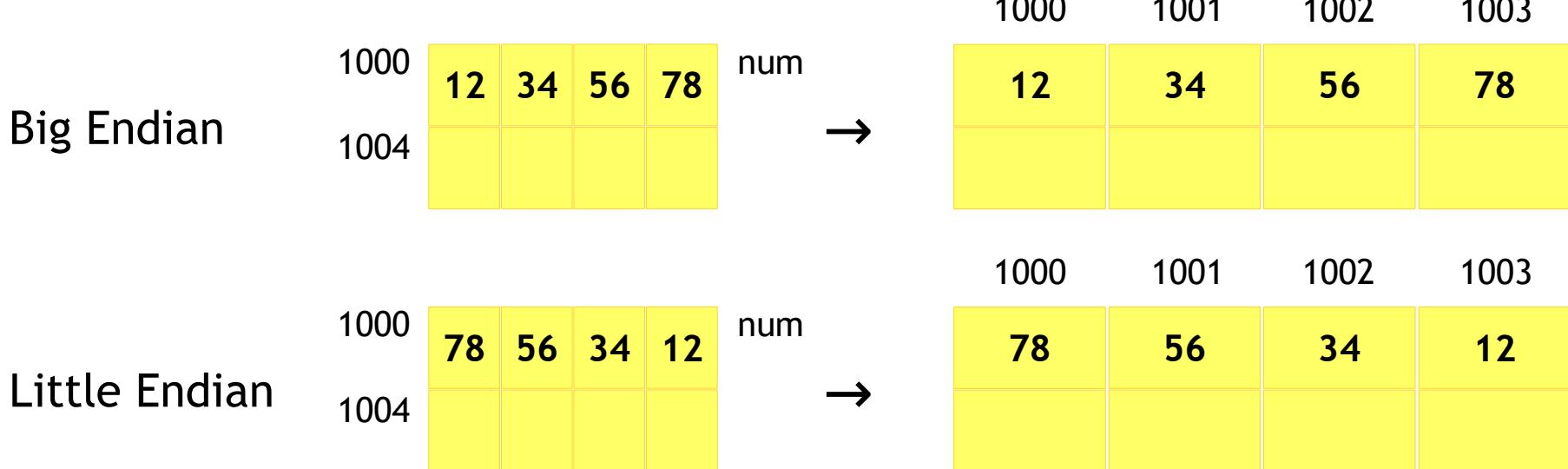
Example

```
#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



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;

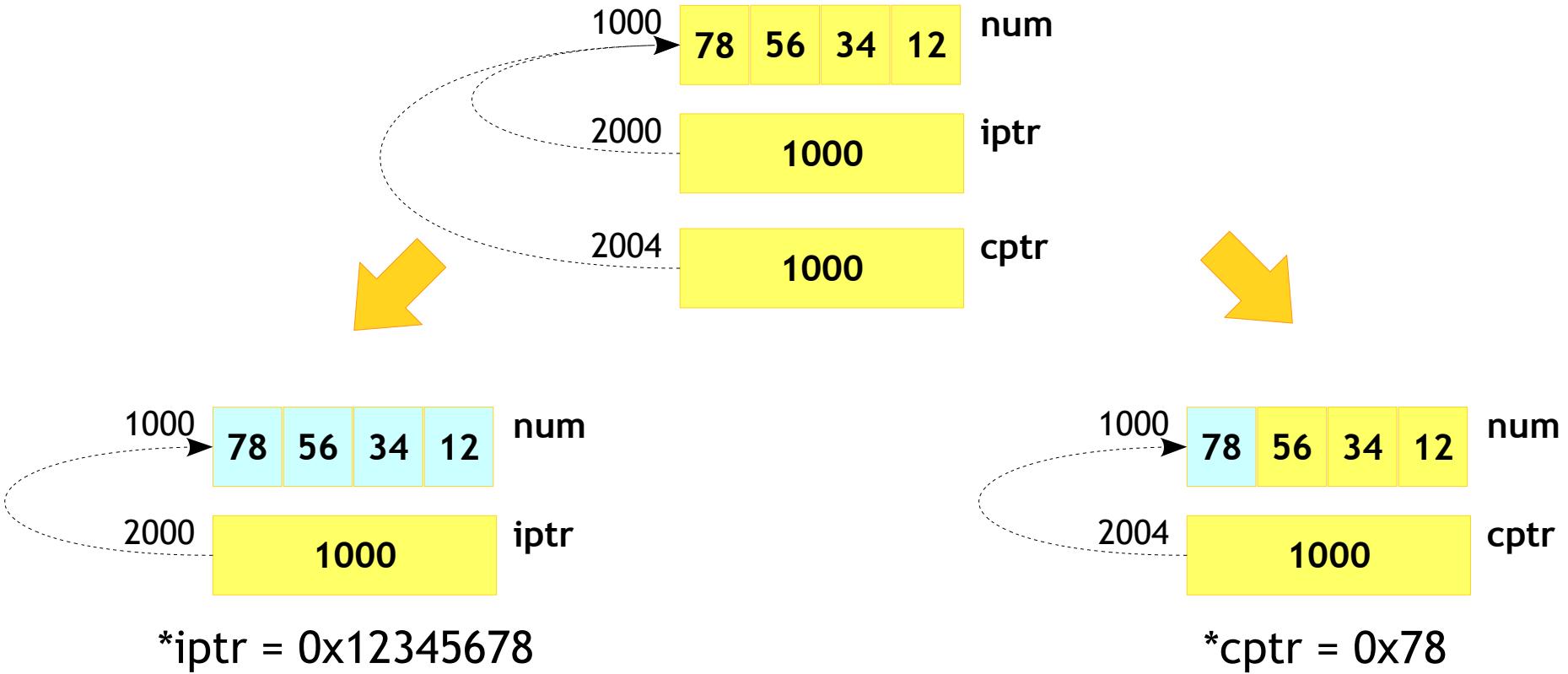
    iptr = &num;
    cptr = &num;

    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>

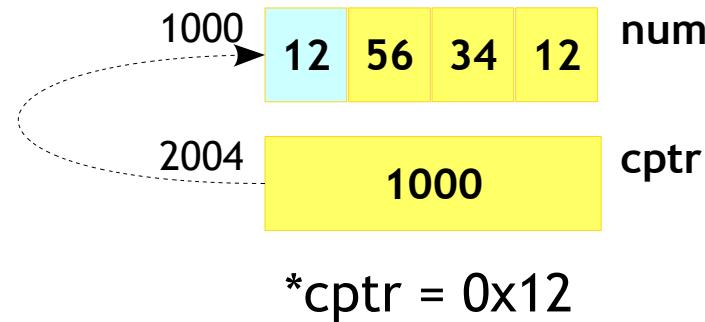
int main()
{
    int num = 0x12345678;
    char ch;

    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;
}
```

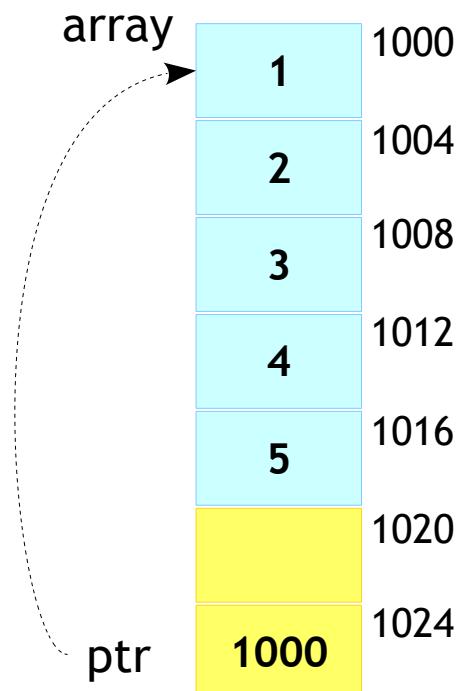
- So,

Address of array = 1000

Base address = 1000

$\&\text{array}[0]$ = 1 → 1000

$\&\text{array}[1]$ = 2 → 1004



Advanced C

Pointers - The Rule 5 in detail

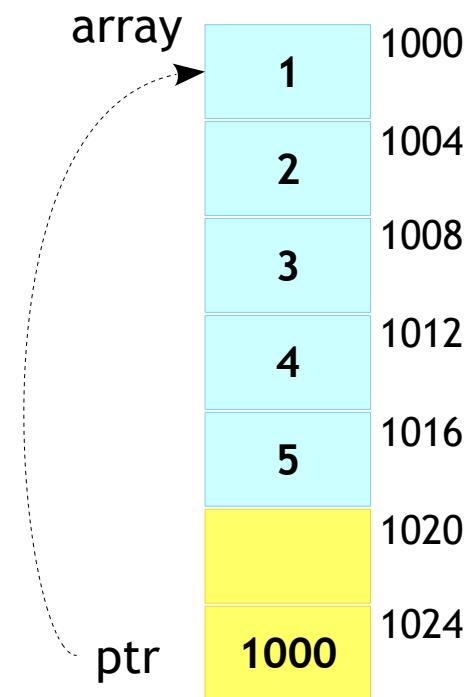
007_example.c

```
#include <stdio.h>

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

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

    return 0;
}
```



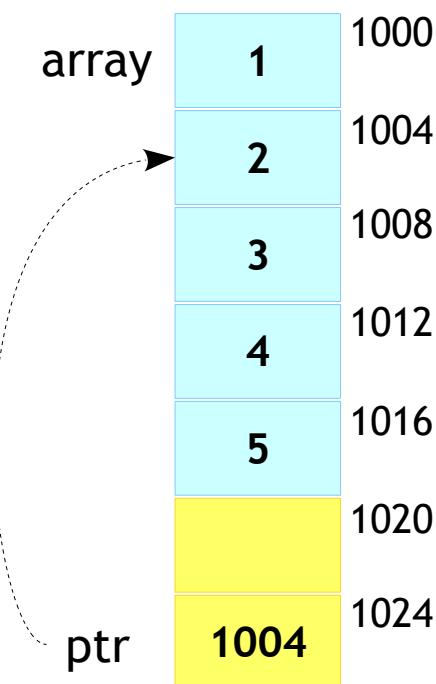
- This code should print 1 as output since its points to the base address
- Now, what should happen if we do

```
ptr = ptr + 1;
```

Advanced C

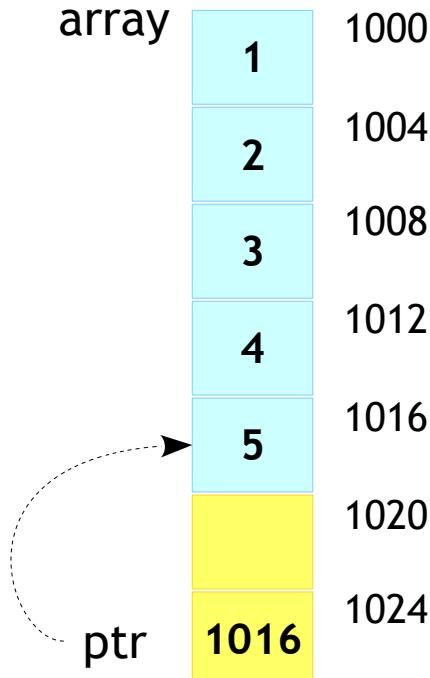
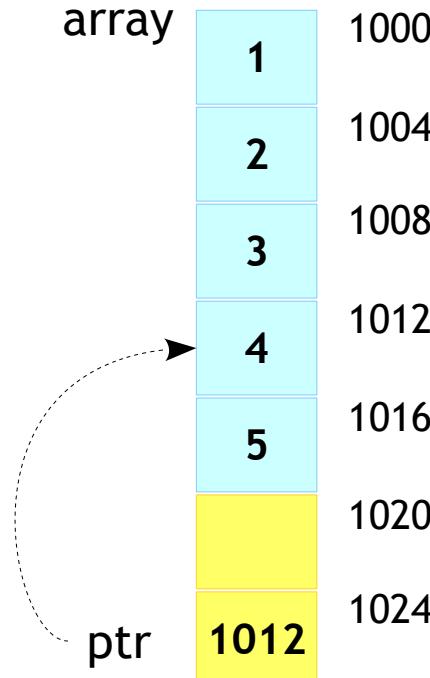
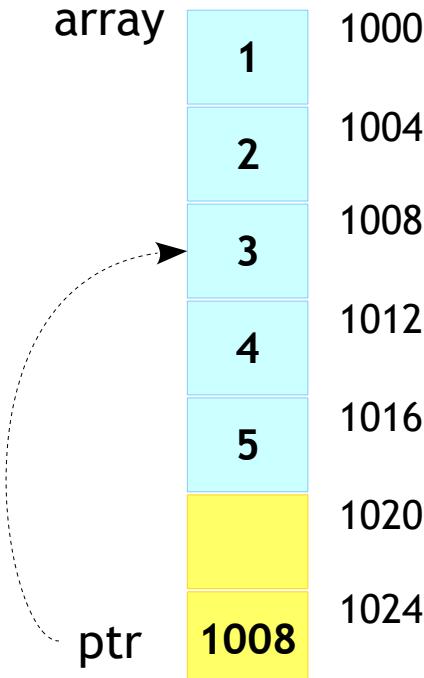
Pointers - The Rule 5 in detail

- `ptr = ptr + 1;`
- The above line can be described as follows
- `ptr = ptr + 1 * 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

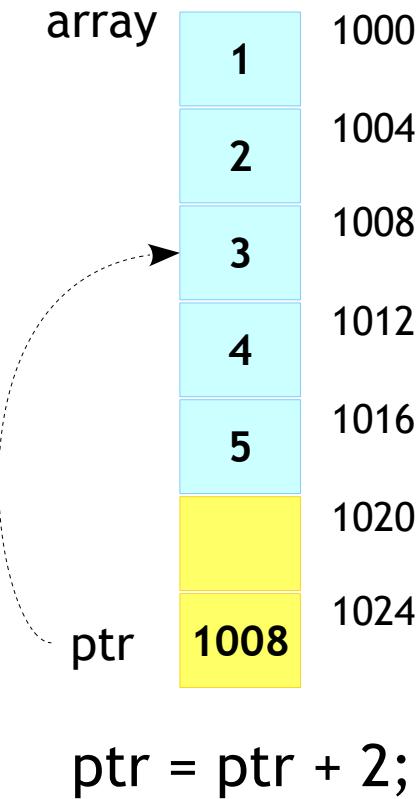
Pointers - The Rule 5 in detail



- 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

$*(\text{ptr} + i) \rightarrow \text{array}[i]$

- This can be written as following too!!

$\text{array}[i] \rightarrow *(\text{array} + i)$

- 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>

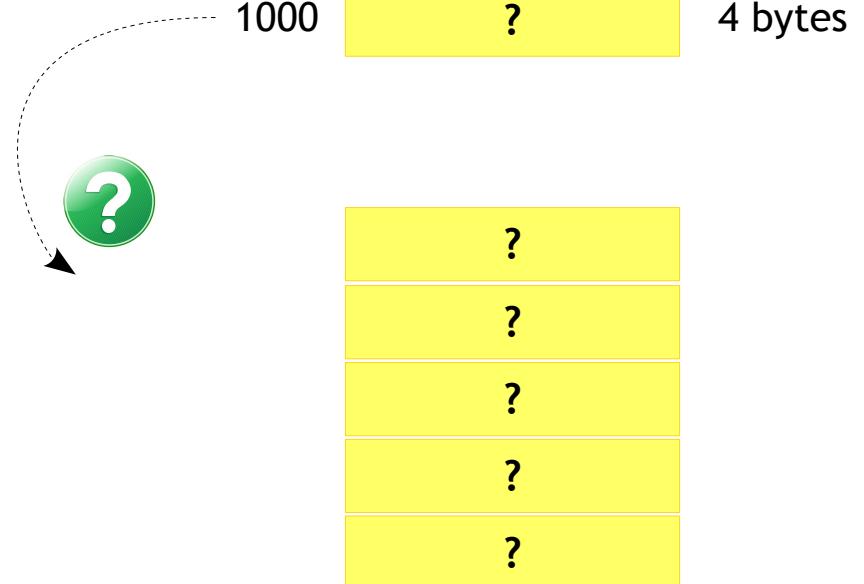
int main()
{
    int *num;

    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.
- ??, 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

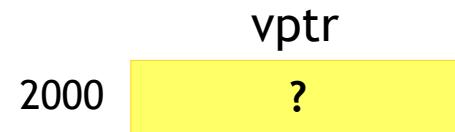
- A generic pointer which can point to data in memory
- The data type has to be mentioned while accessing the memory which has to be done by type casting

Example

```
#include <stdio.h>

int main()
{
    void *vptr;

    return 0;
}
```



Advanced C

Pointers - Void Pointer

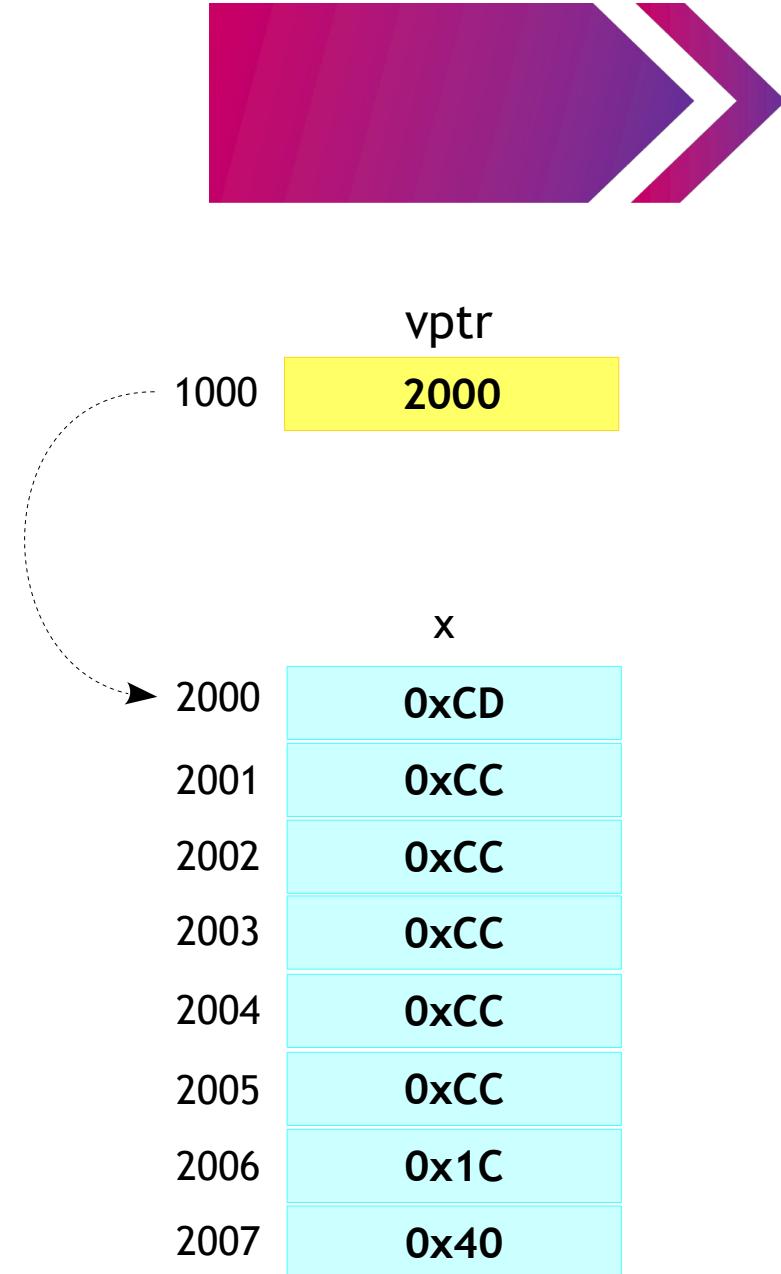
008_example.c

```
#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

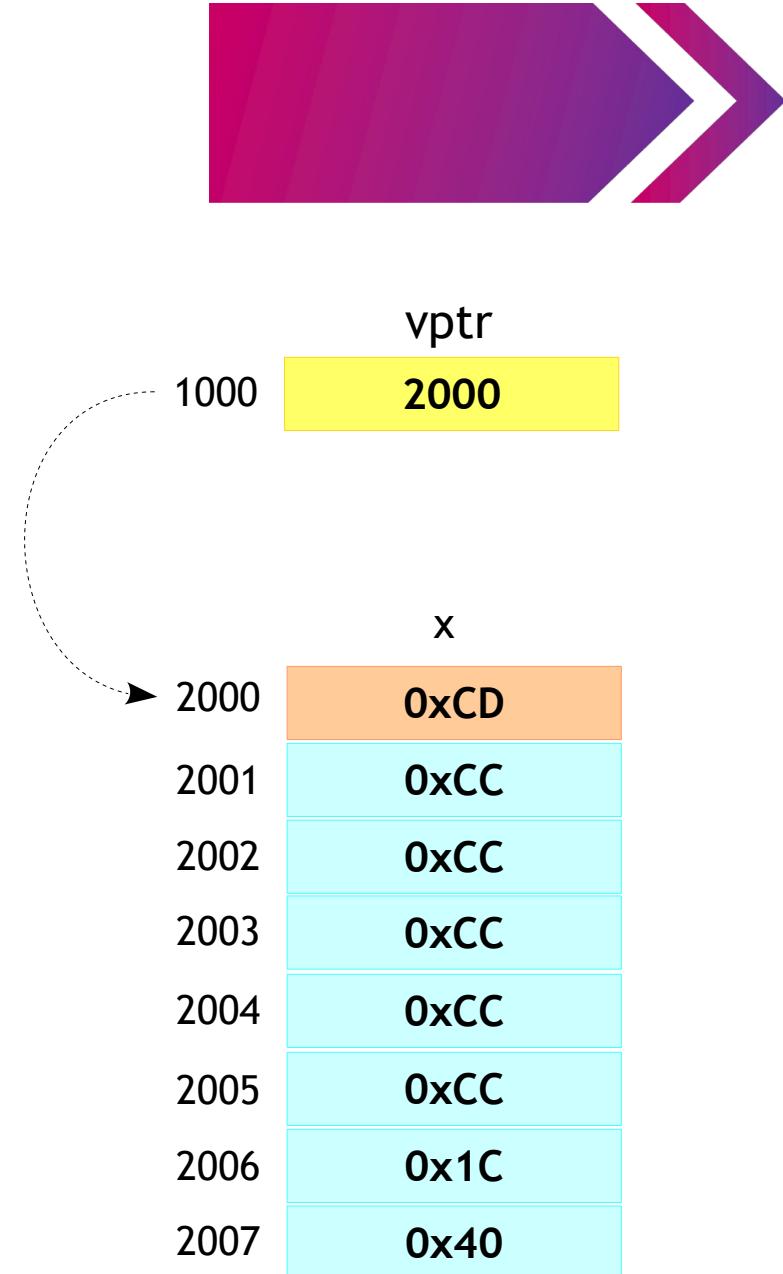
008_example.c

```
#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

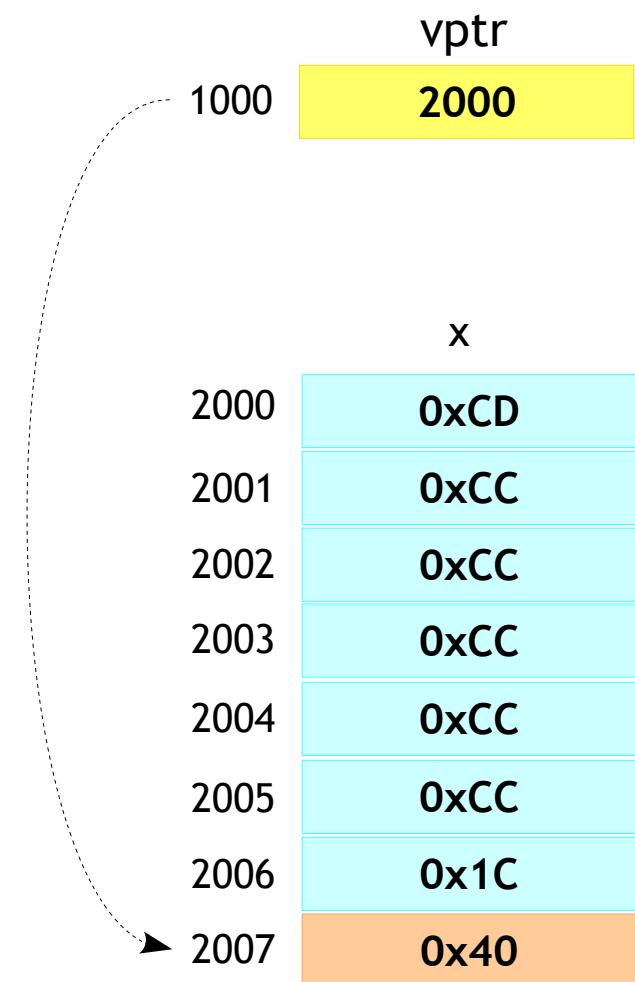
008_example.c

```
#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;
}
```



Advanced C

Pointers - Void Pointer

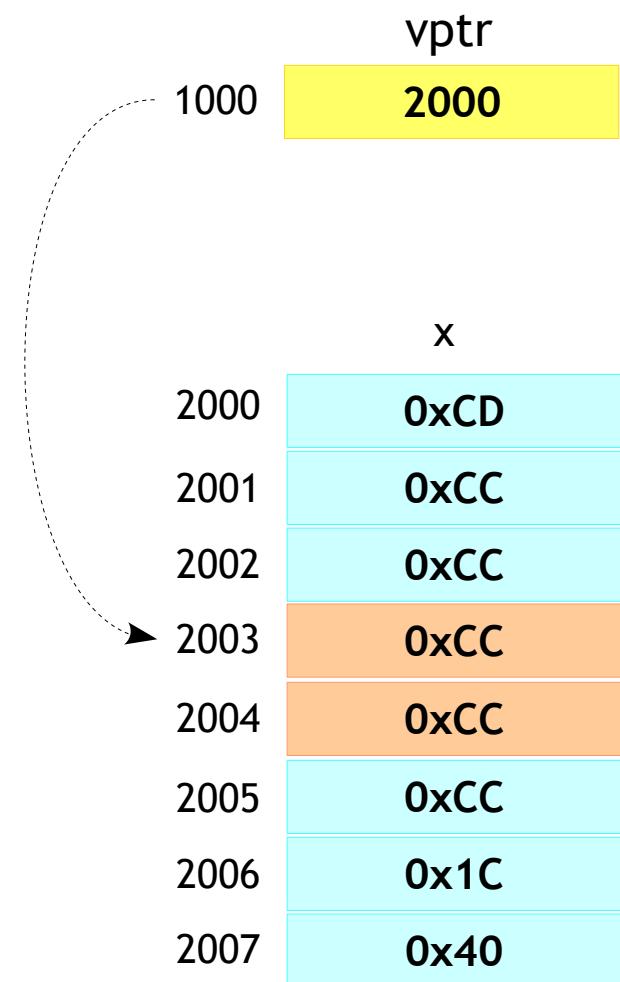
008_example.c

```
#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

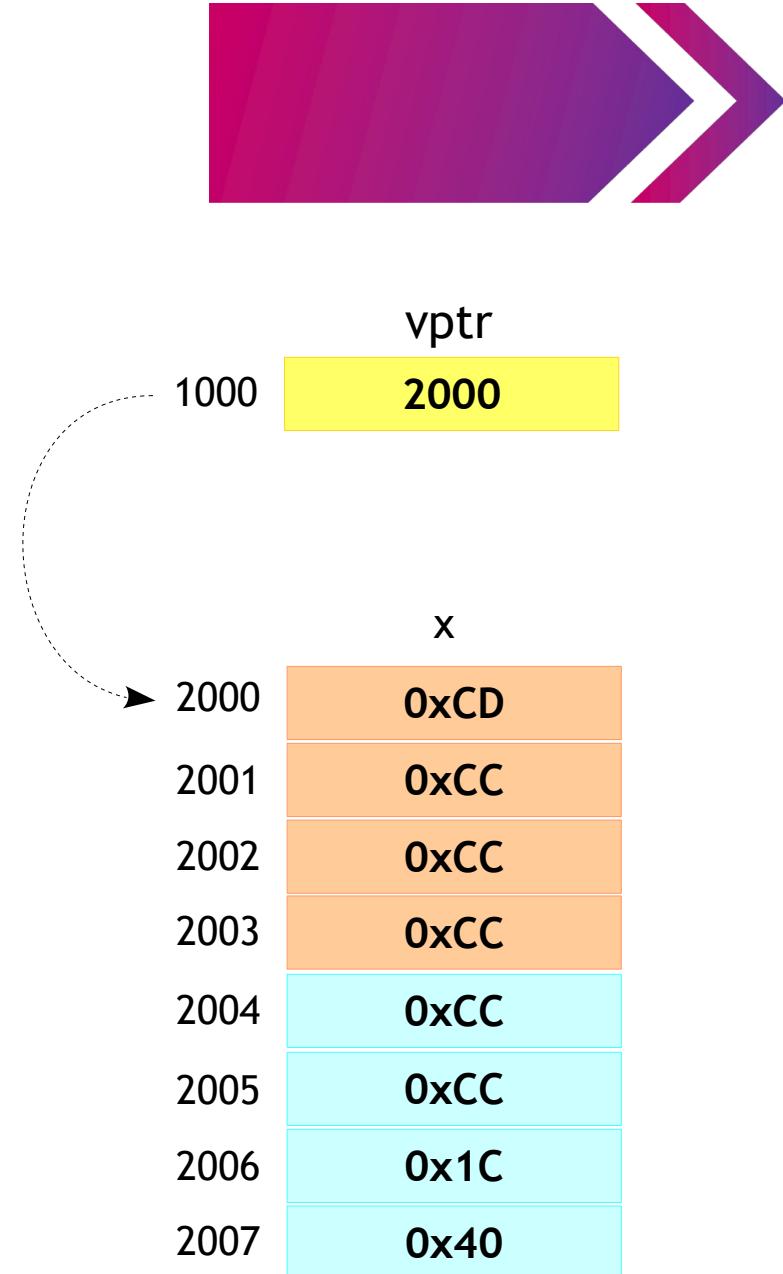
008_example.c

```
#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

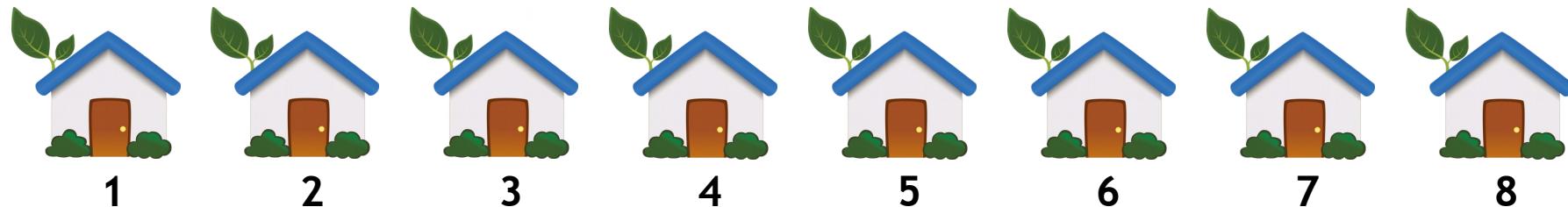
Pointers - Rule 7 in detail



- 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

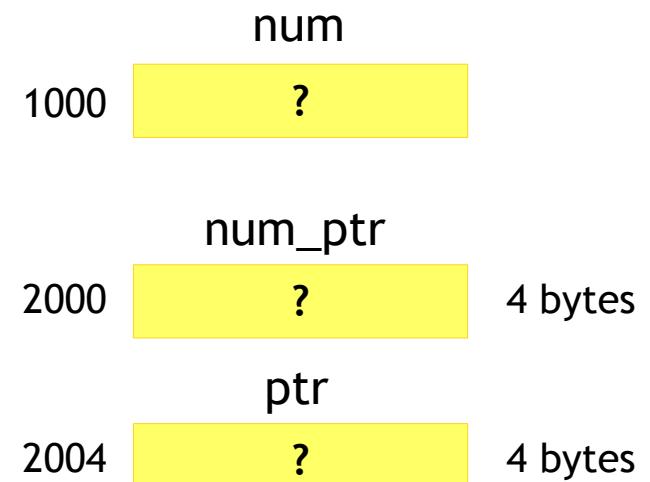
```
#include <stdio.h>

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

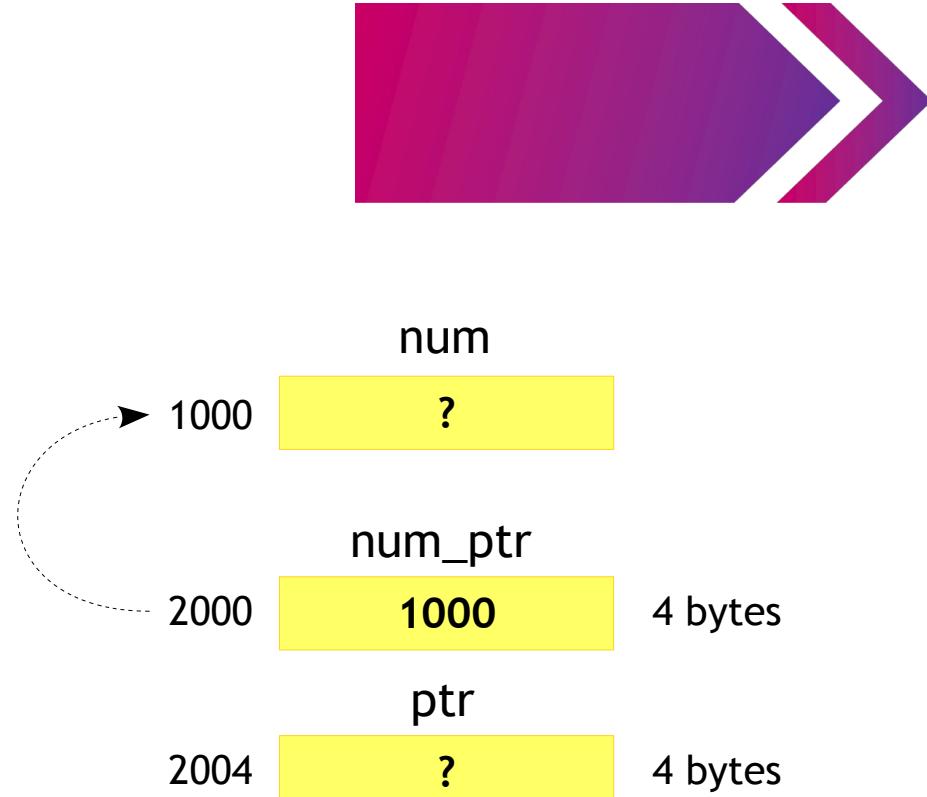
Example

```
#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

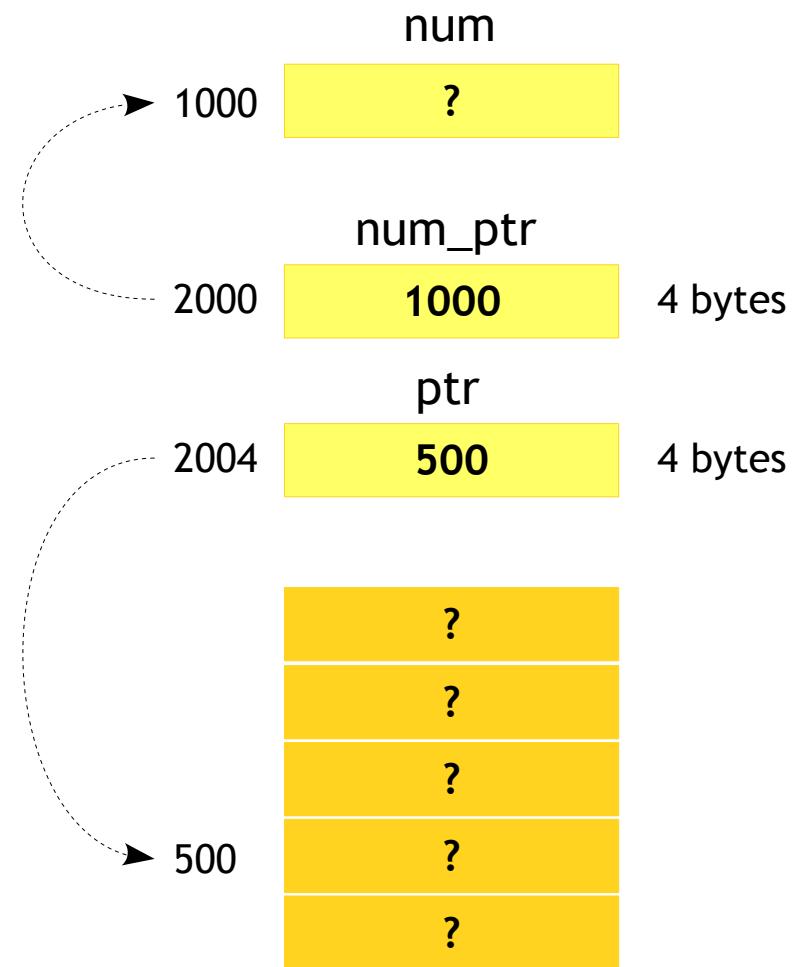
Example

```
#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

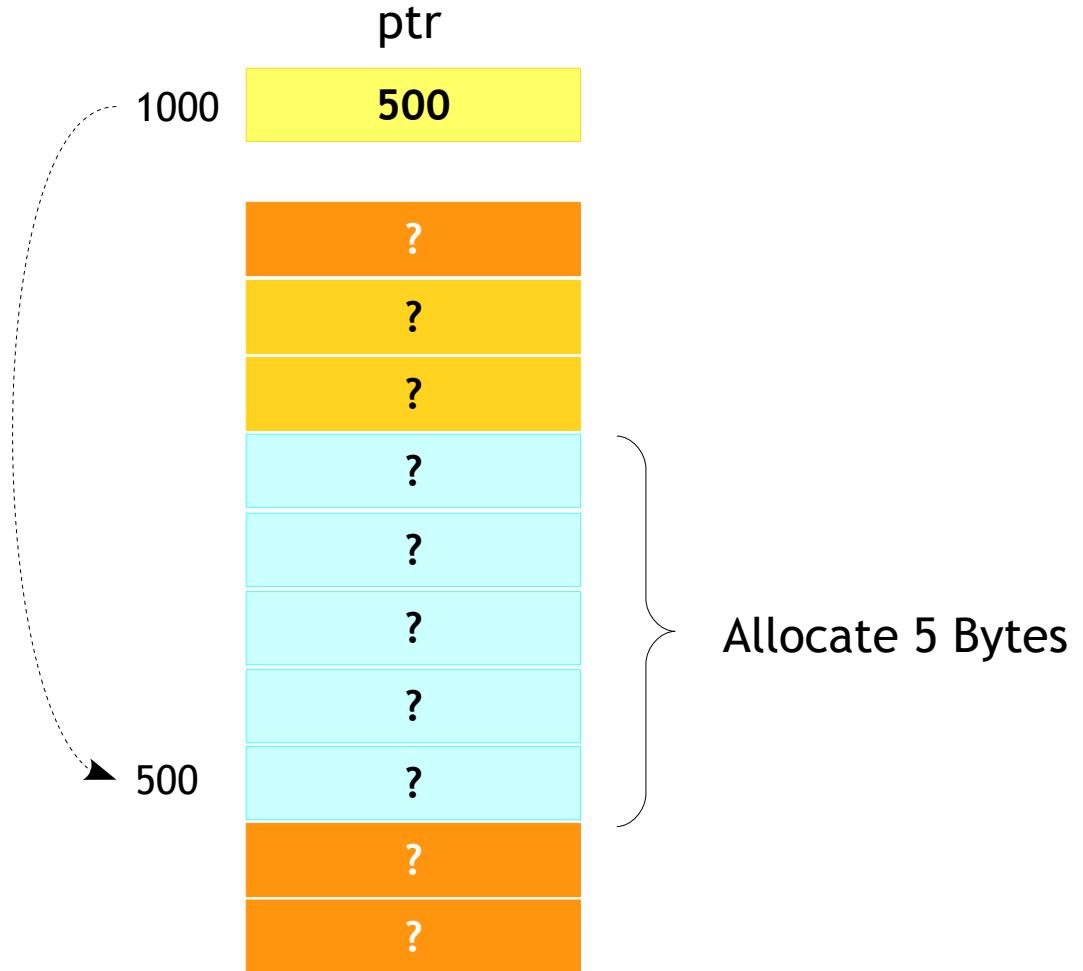
Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    return 0;
}
```



Advanced C

Pointers - Rule 7 - Dynamic Allocation - malloc

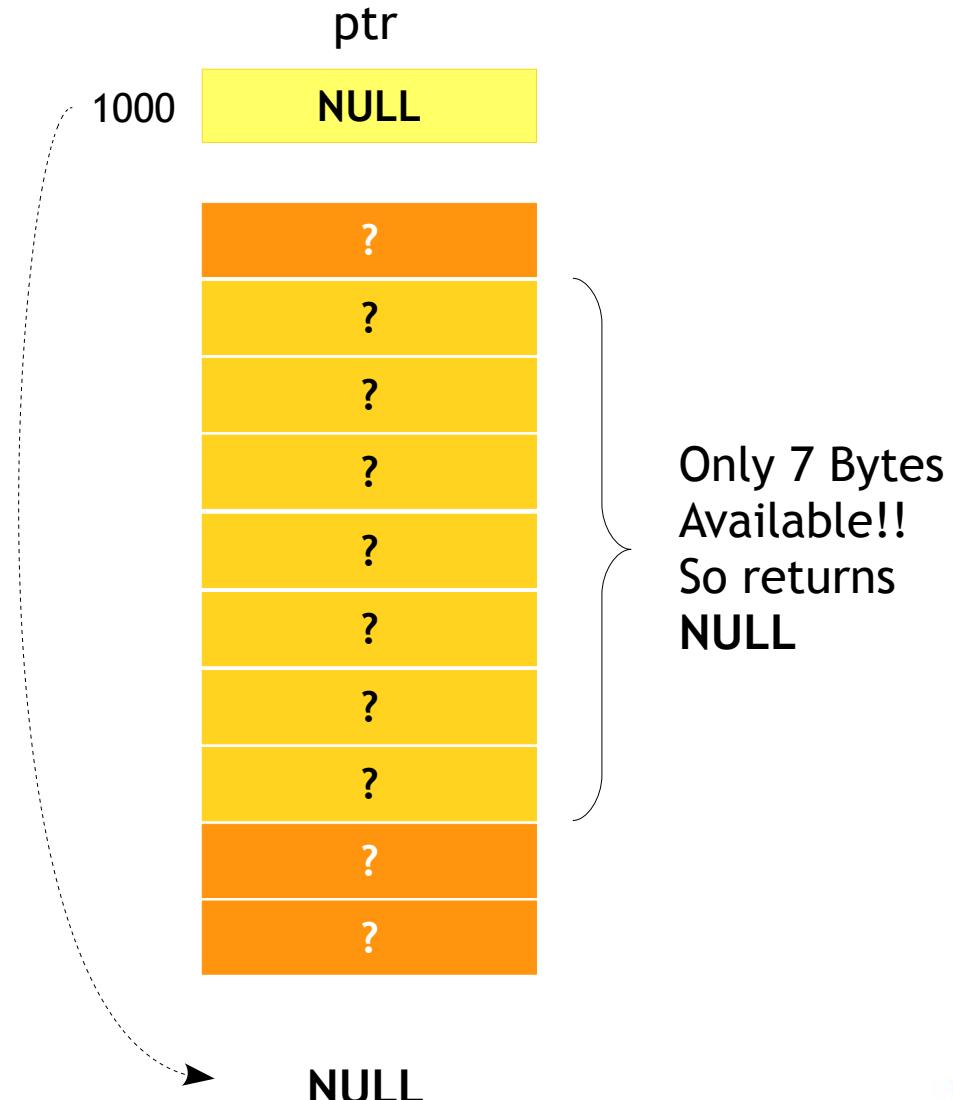
Example

```
#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

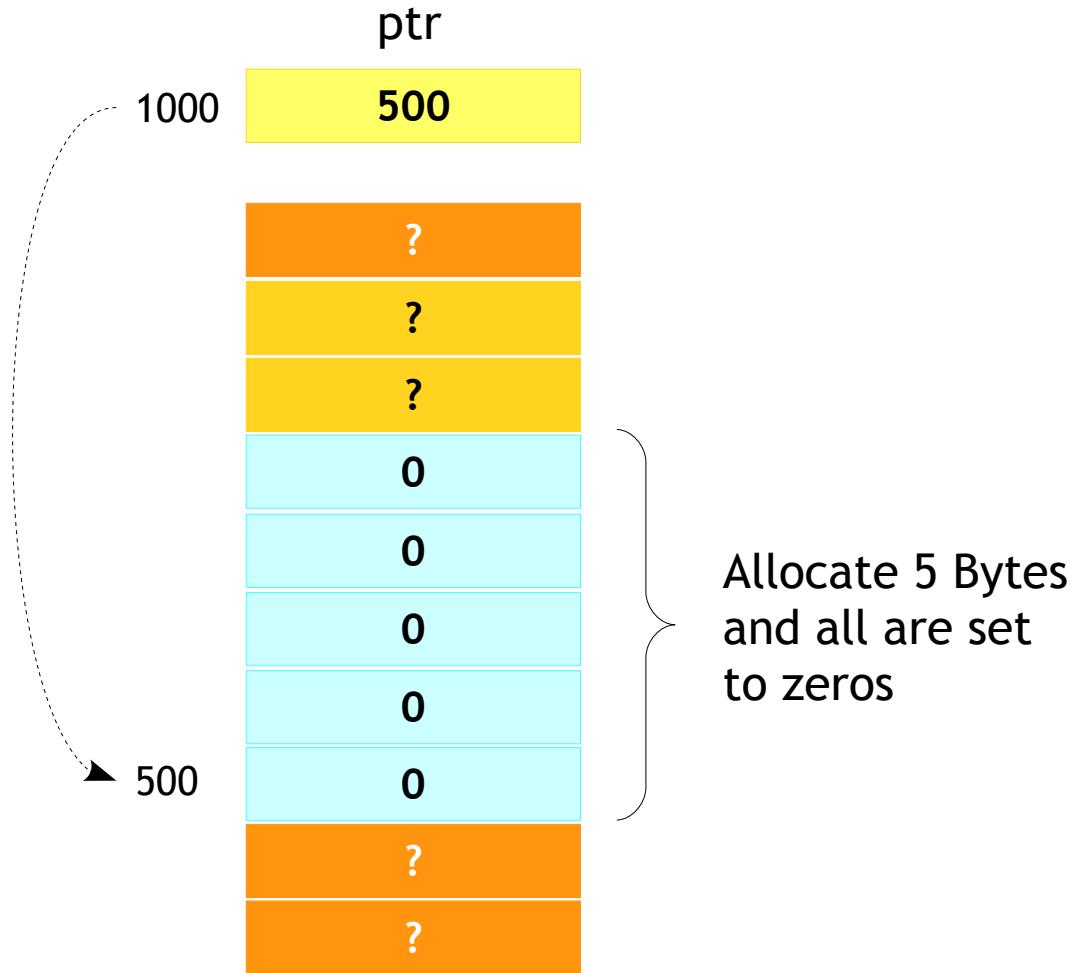
Example

```
#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

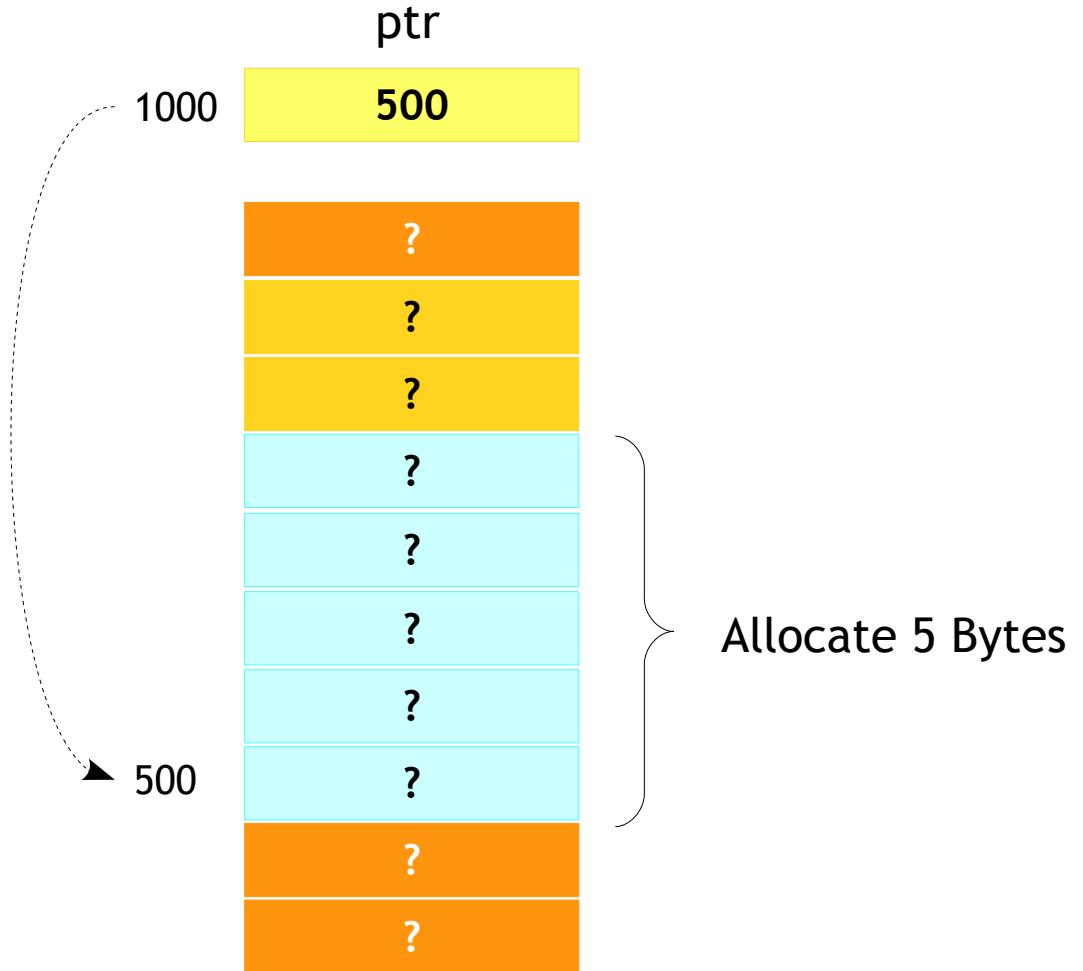
```
#include <stdio.h>

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

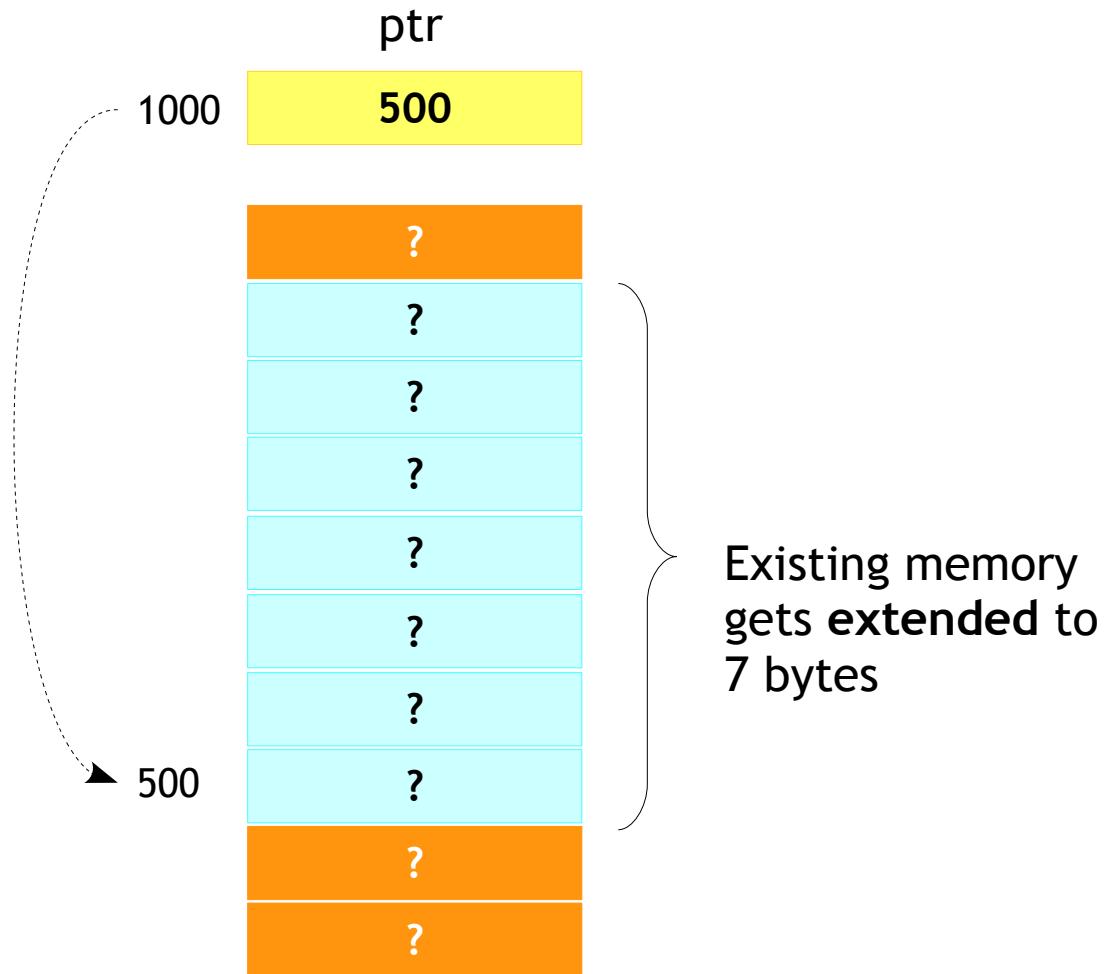
```
#include <stdio.h>

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

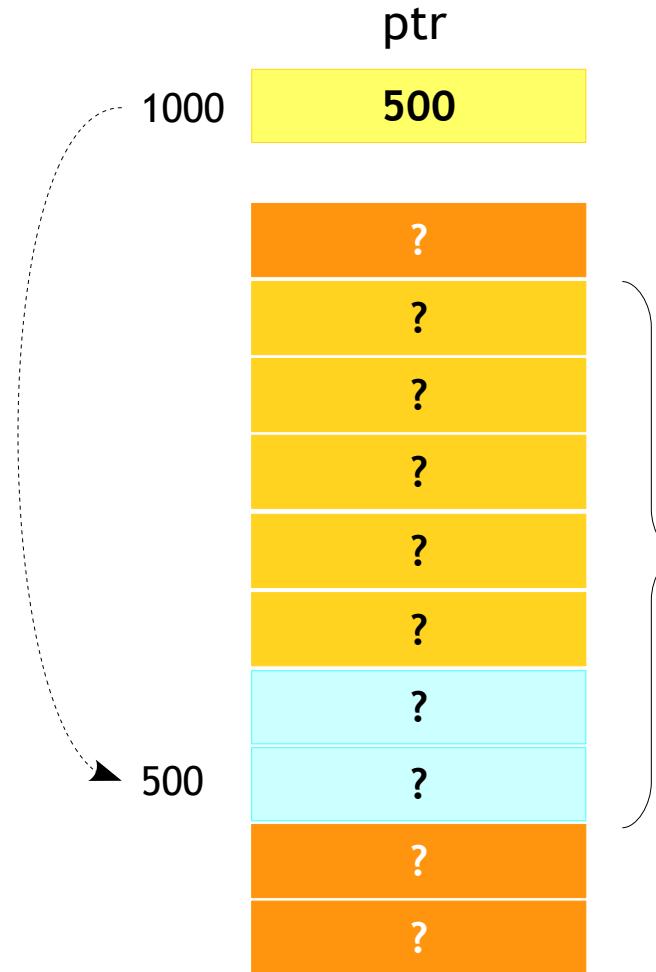
```
#include <stdio.h>

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>

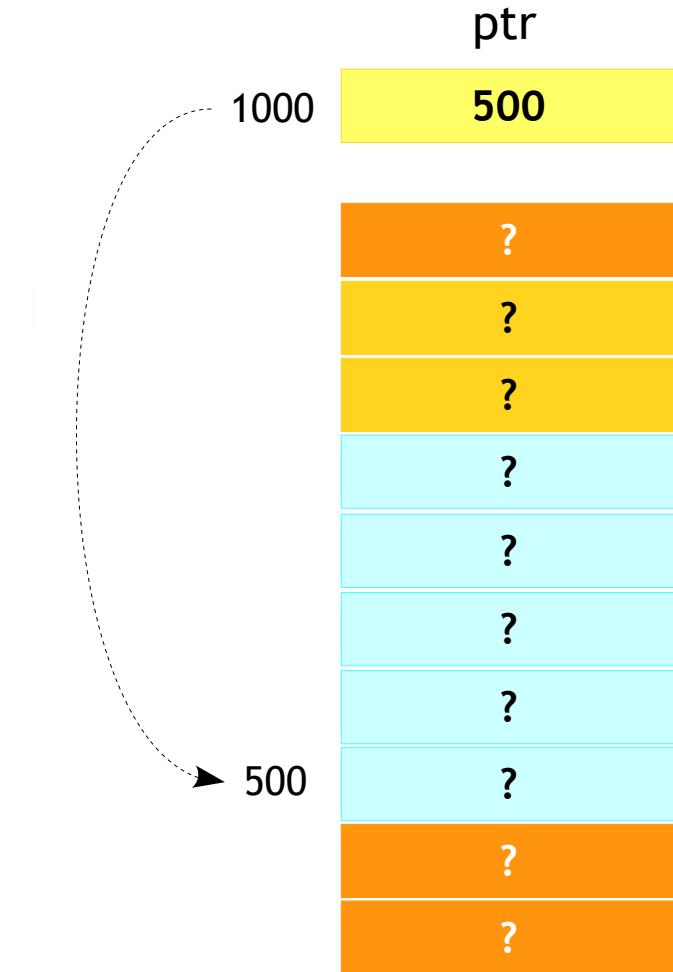
int main()
{
    char *ptr;
    int iter;

→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>

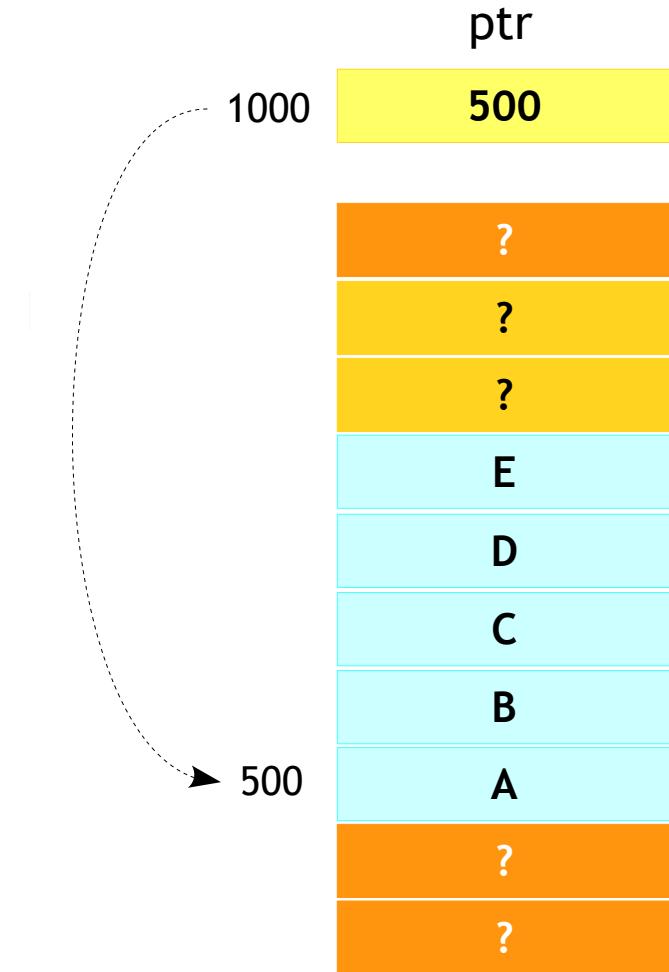
int main()
{
    char *ptr;
    int iter;

    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

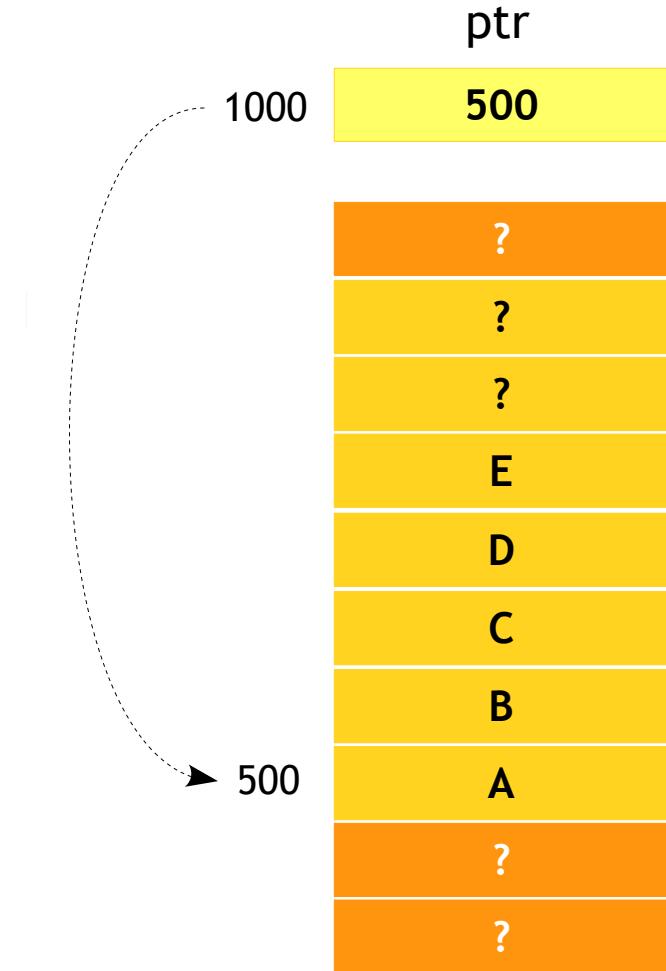
```
#include <stdio.h>

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

    ptr = malloc(5);

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

→ [ free(ptr);
    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

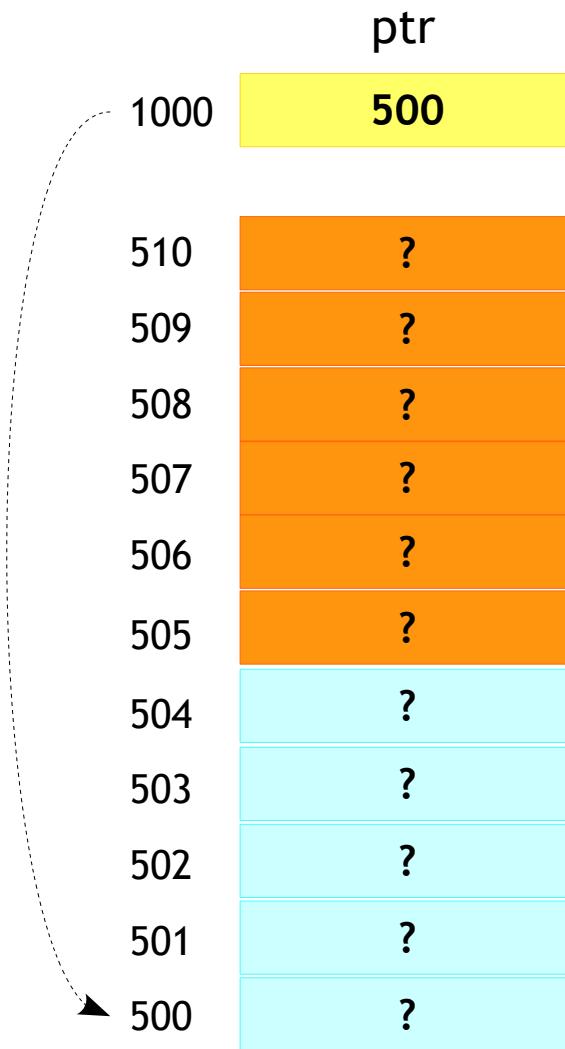
```
#include <stdio.h>

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

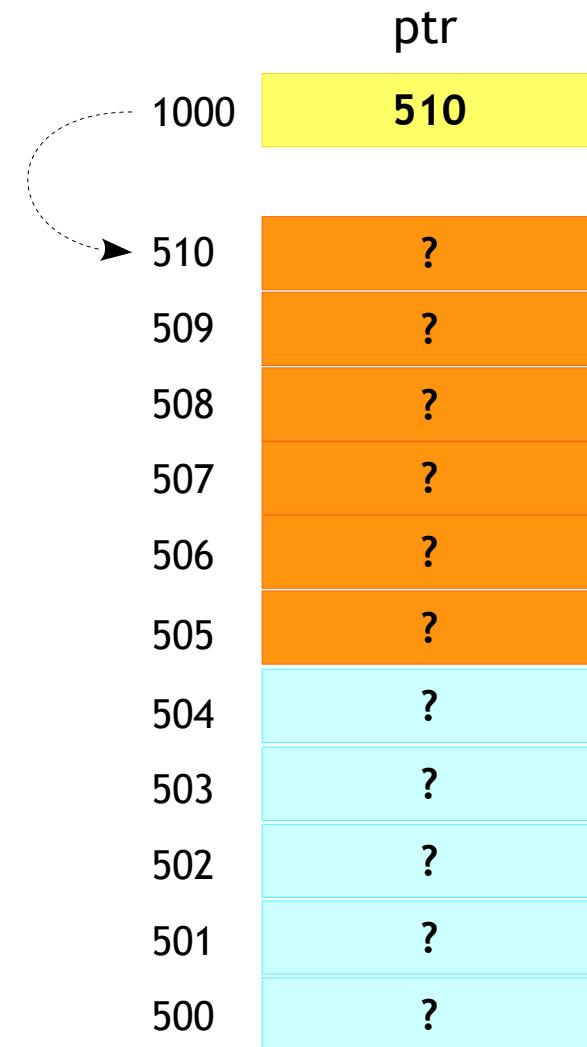
Example

```
#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

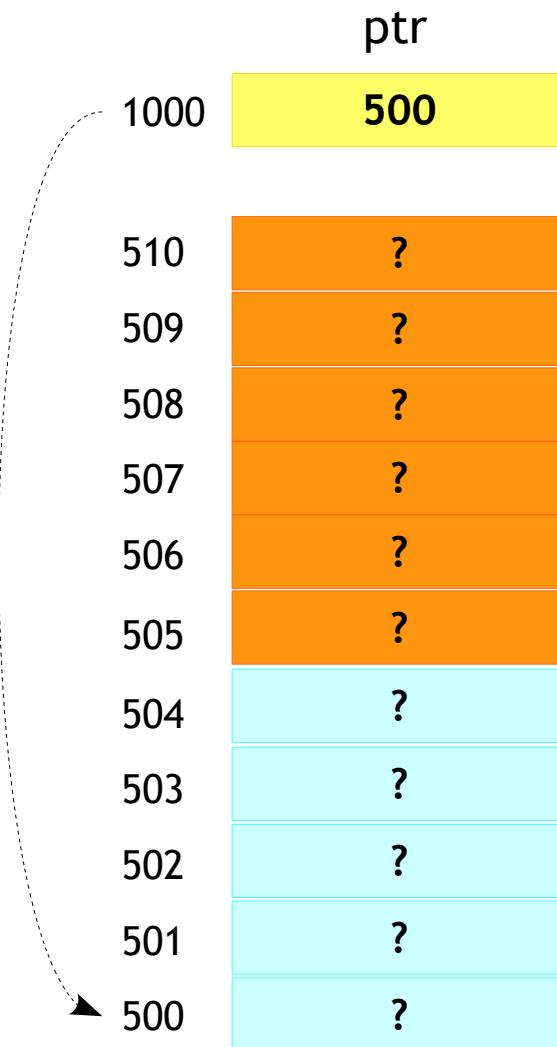
```
#include <stdio.h>

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;
}
```

Standard I/O



Advanced C

Standard I/O - Why should I know this? - DIY

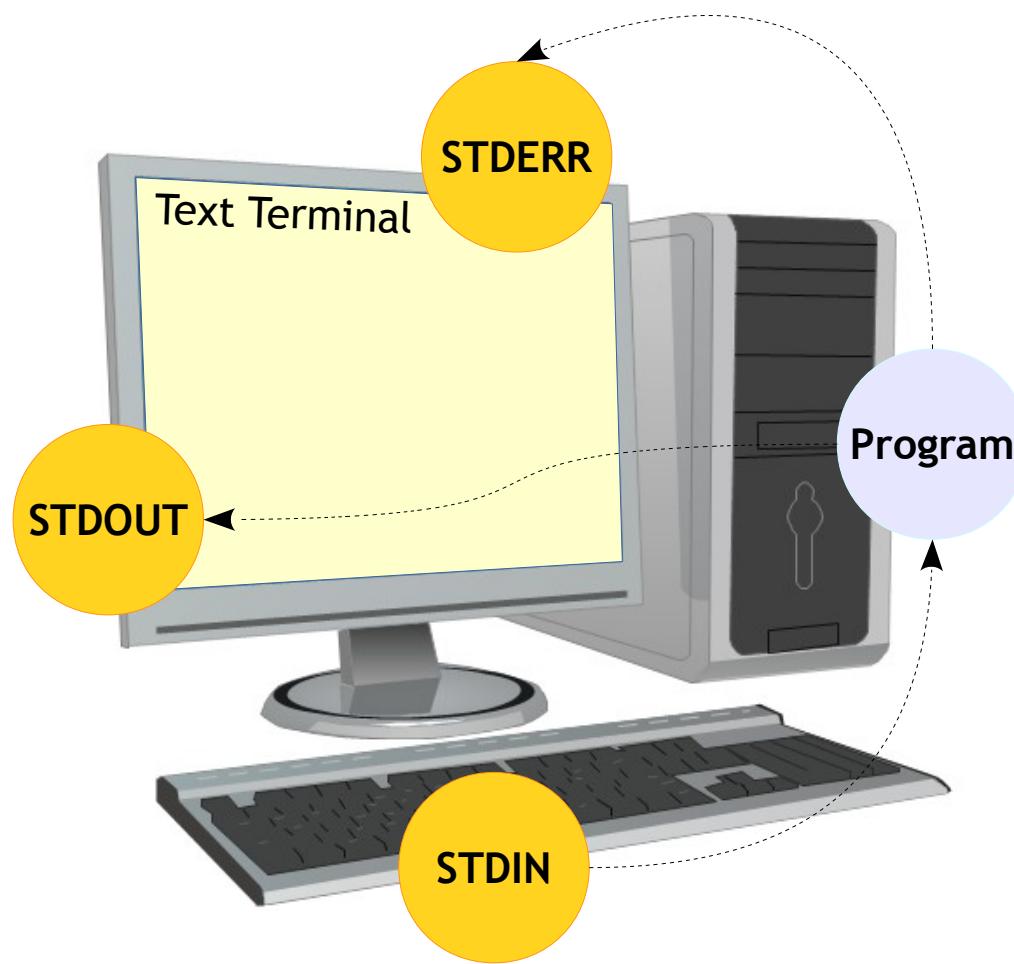
Screen Shot

```
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

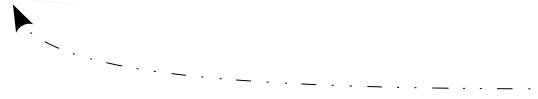
Advanced C

Standard I/O - printf()



Prototype

```
int printf(const char *format, ...);
```



What is this!?



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

What should be written here and how many?

```
int printf("%c %d %f", );
```

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?

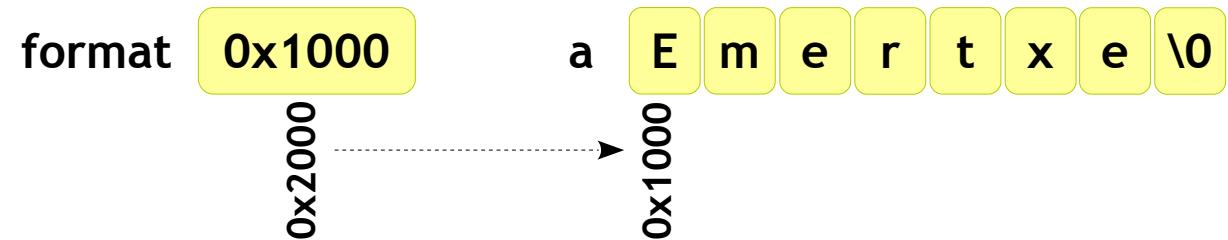
And strings are nothing but array of characters terminated by null

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	<code>printf("%3.1d", 1)</code> <code>printf("%3.2d", 1)</code> <code>printf("%3.3d", 1)</code>	1 01 001
%0.[x]f	<code>printf("%0.3f", 1.0)</code> <code>printf("%0.10f", 1.0)</code>	1.000 1.0000000000
%[x].[x]s	<code>printf("%12.8s", "Hello World")</code>	Hello Wo

Advanced C

Standard I/O - printf() - Flags

010_example.c

Flag	Example	Expected Output
%[#]x	printf("%#x %#X %#x", 0xA, 0xA, 10) printf("%#o", 8)	0xa 0XA 0xa 010
%[-x]d	printf("%-3d %-3d", 1, 1) printf("%-3d %-3d", 10, 10) printf("%-3d %-3d", 100, 100)	1 1 10 10 100 100
%[]3d	printf("% 3d", 100) printf("% 3d", -100)	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", &h, &m, &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-zA-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("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:~]
```



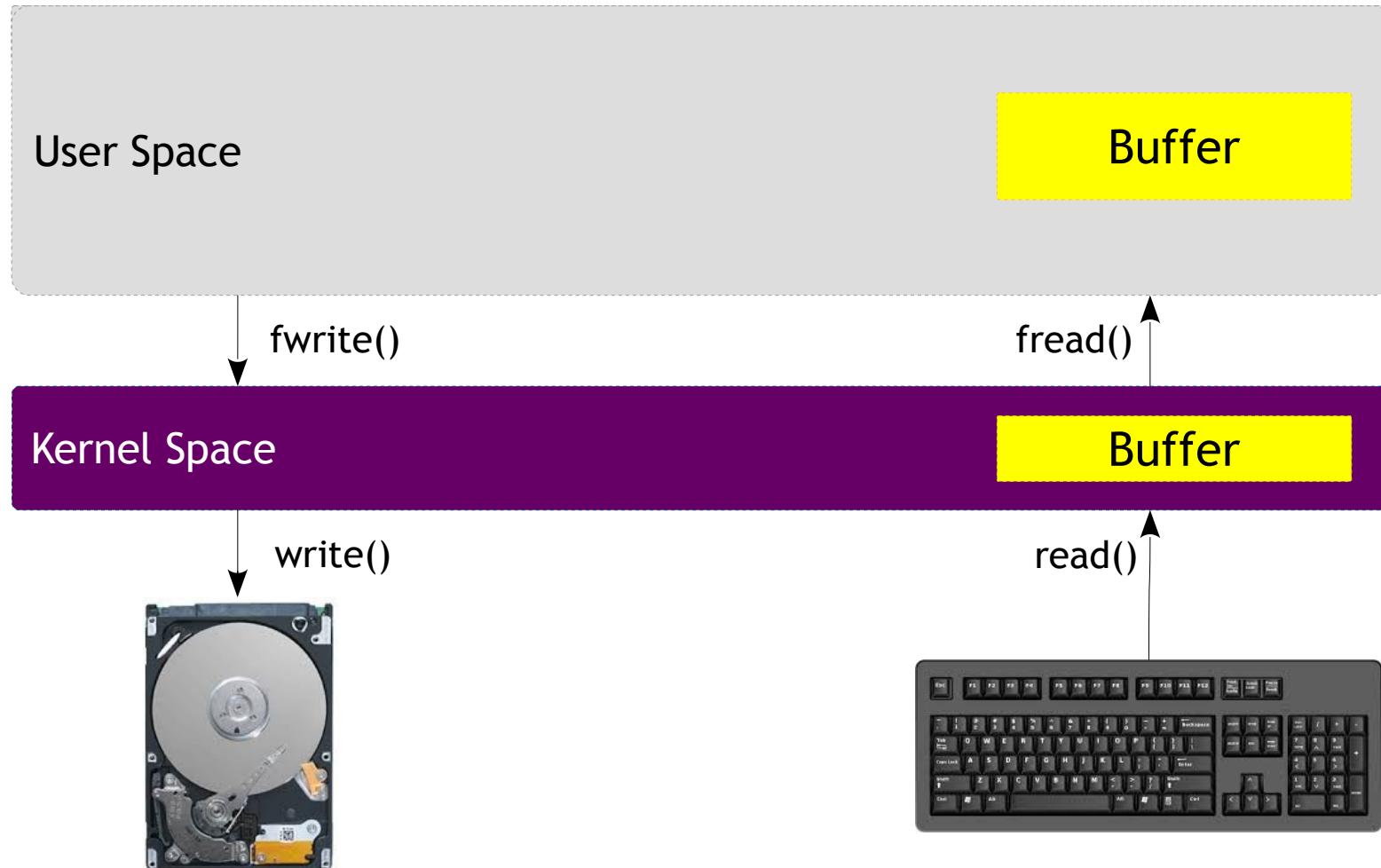
Advanced C

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 ;)

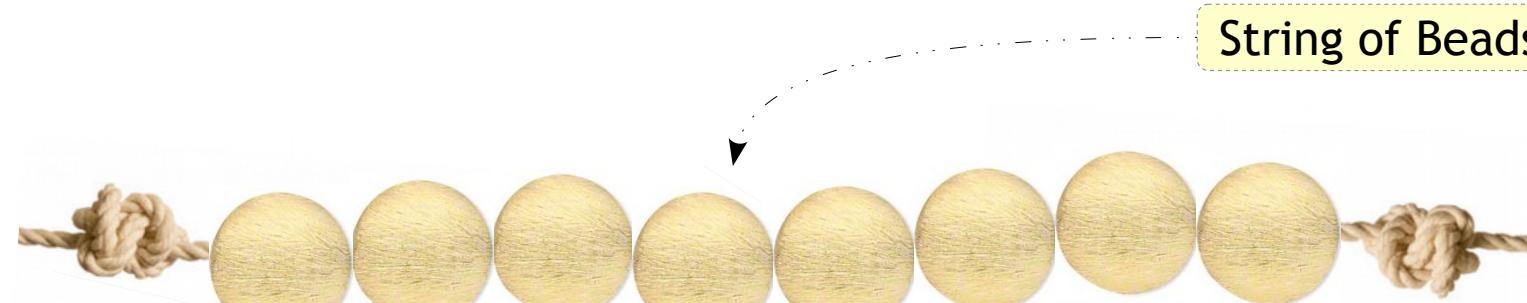


Advanced C Strings



A set of things tied or threaded together on a thin cord

Source: Google



String

Start of String

Bead

Contains n numbers of Beads

String of Beads

String ends here



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

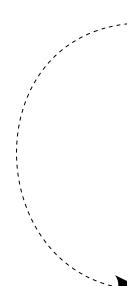
```
char str1[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```

```
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>

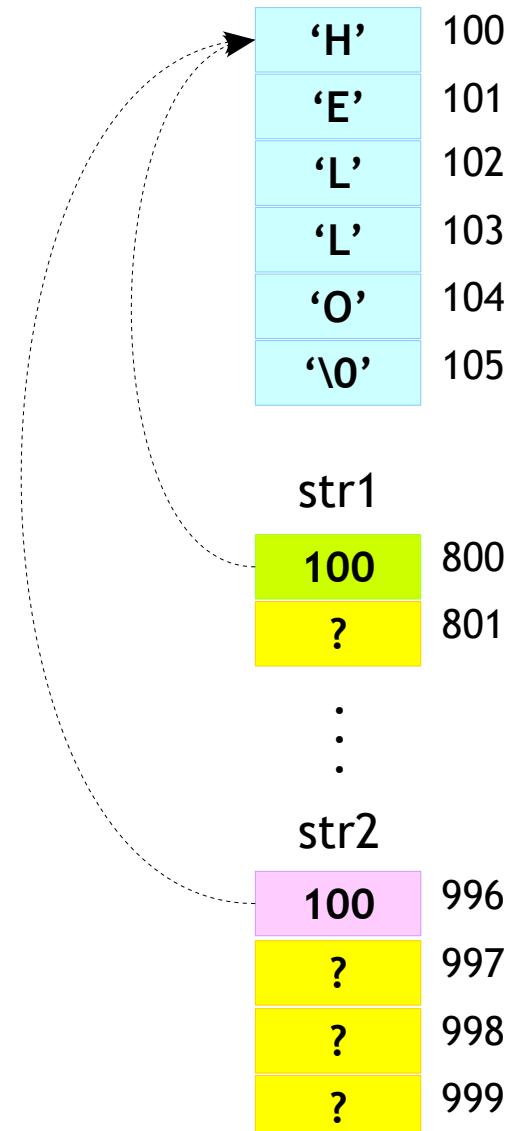
int main()
{
    char *str1 = "Hello";
    char *str2 = "Hello";

    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>	$\text{str1} < \text{str2} \rightarrow < 0$ $\text{str1} > \text{str2} \rightarrow > 0$ $\text{str1} = \text{str2} \rightarrow = 0$
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 - Quiz - Pointer vs Array



Pointer	Array
A single variable designed to store address	A bunch of variables; Each variable is accessed through index number
Size of pointer depends on size of address (Ex - 32 bit or 64 bit)	Size of Array depends on number of elements and their type
Pointer is a lvalue - <ul style="list-style-type: none">• Pointers can be modified (can be incremented/decremented or new addresses can be stored)	Array name is not lvalue - <ul style="list-style-type: none">• Array name represents either whole array (when operand to sizeof operator) or base address (address of first element in the array)• Can't be modified (Array can't be incremented/decremented or base address can't be changed)

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”



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

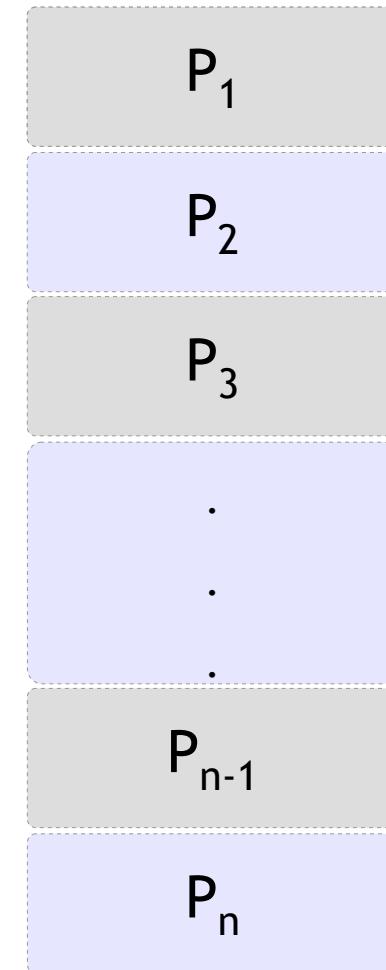
Memory Segments



Linux OS



User Space



The User space contains many processes

Every process will be scheduled by the kernel

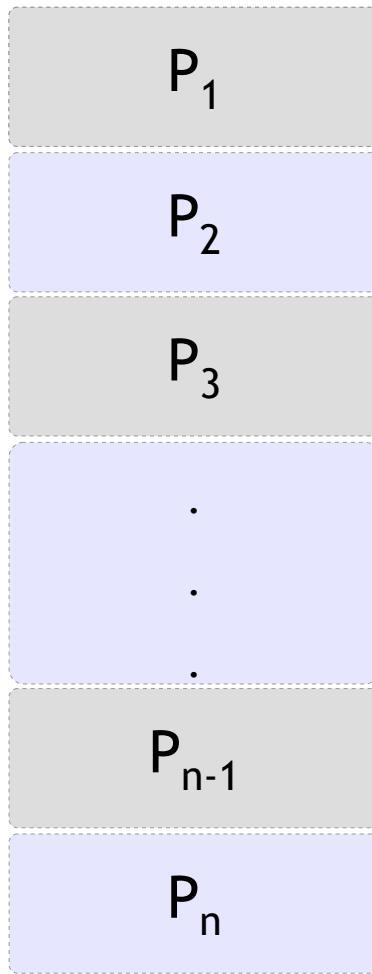
Each process will have its memory layout discussed in next slide

Advanced C

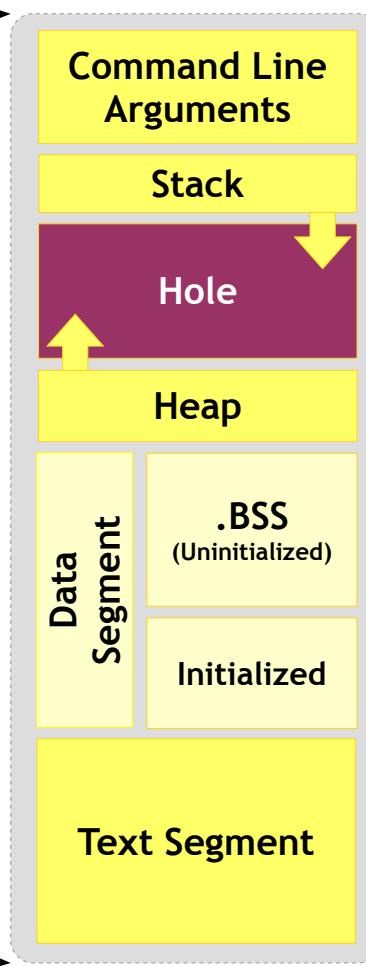
Memory Segments



User Space



Memory Segments



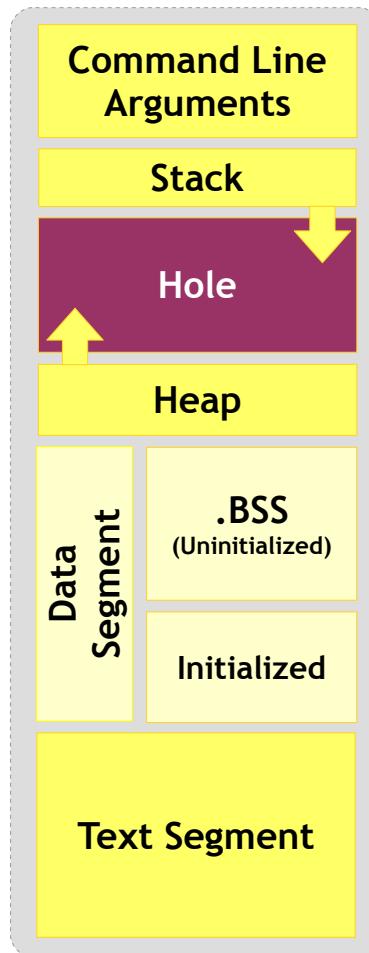
The memory segment of a program contains four major areas.

- 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

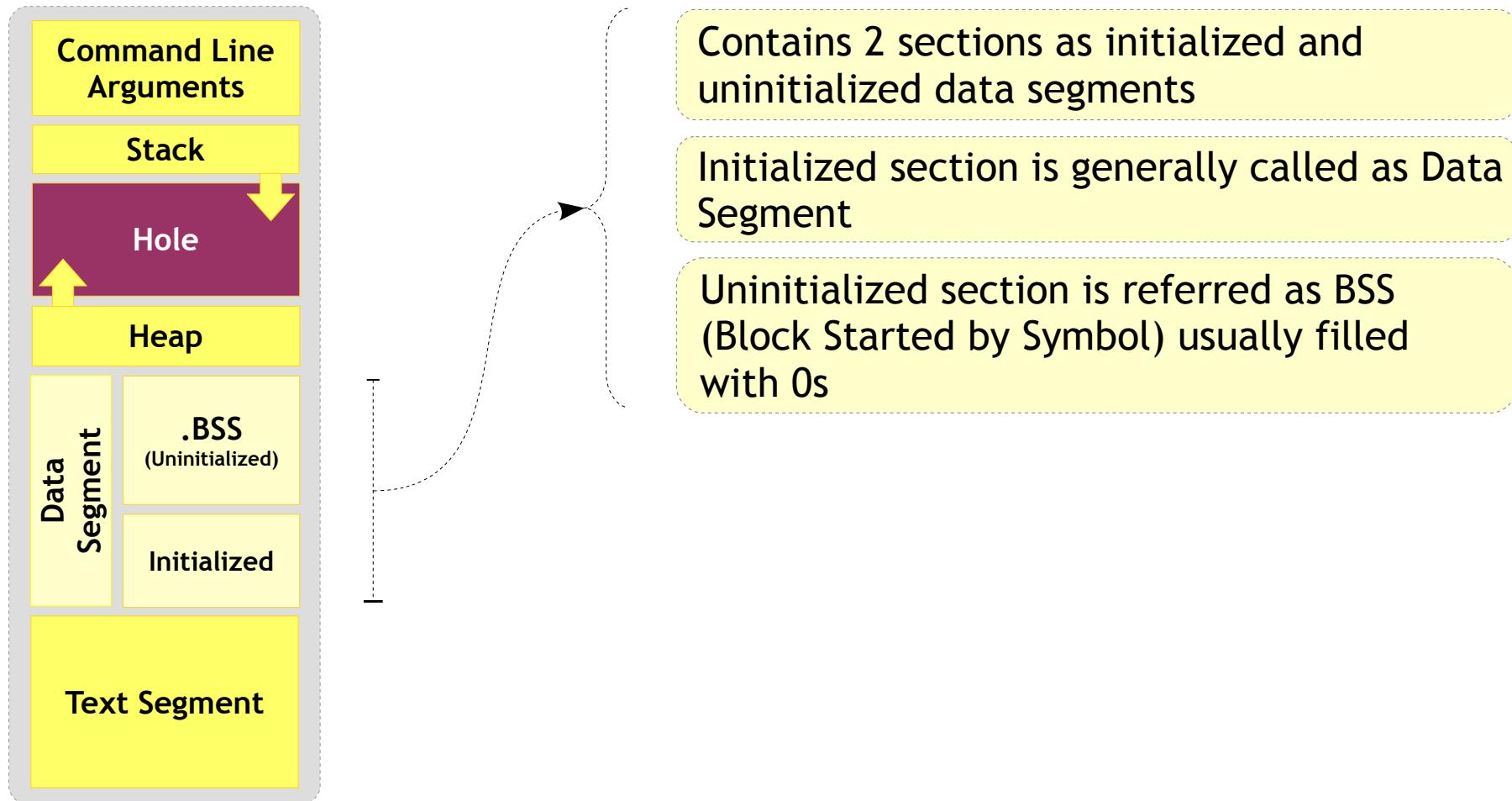
In memory, this is place below the heap or stack to prevent getting over written

Is a read only section and size is fixed

Advanced C

Memory Segments - Data Segment

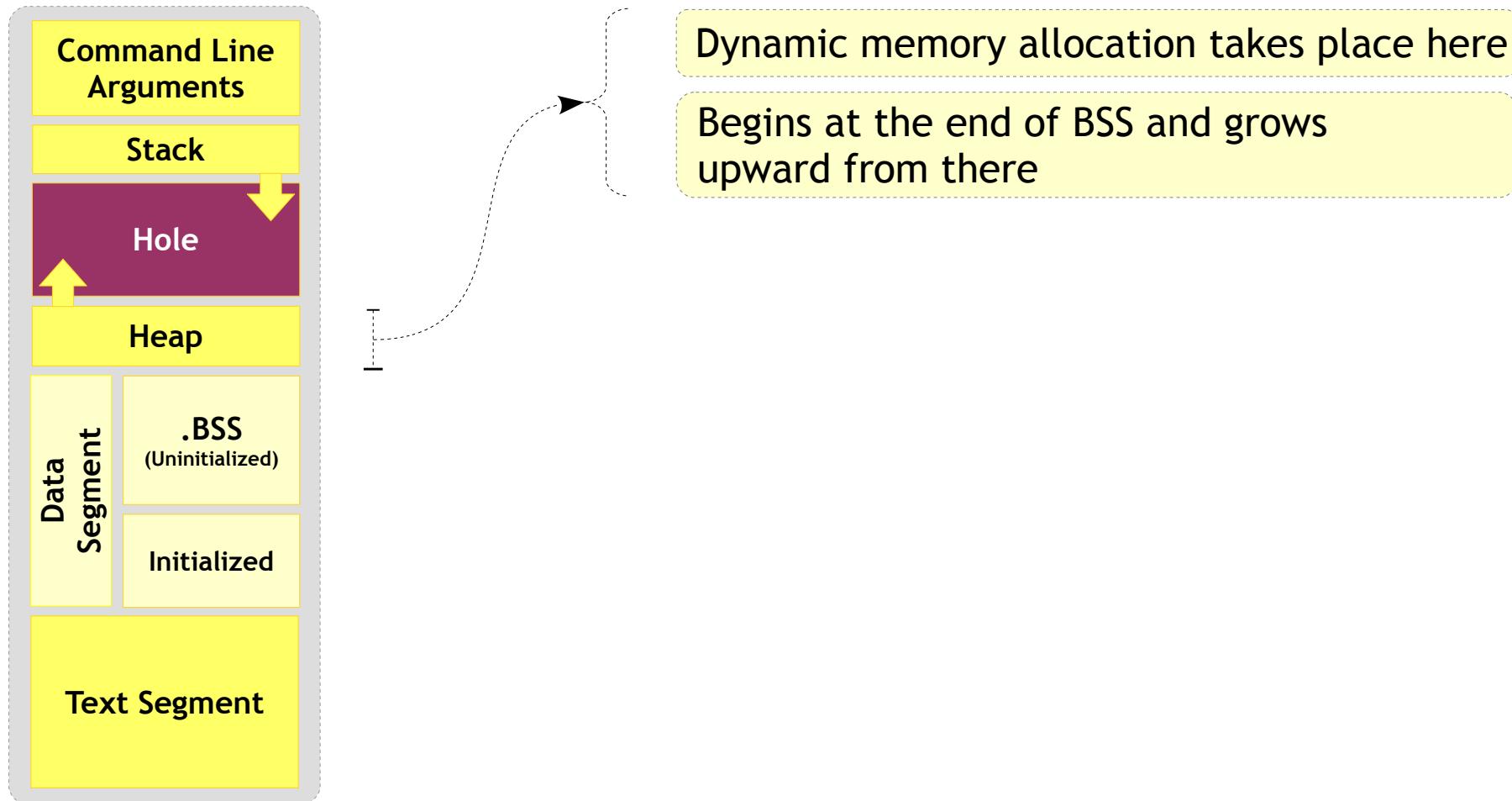
Memory Segments



Advanced C

Memory Segments - Data Segment

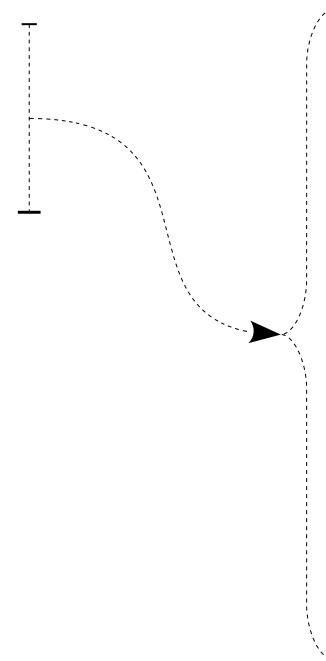
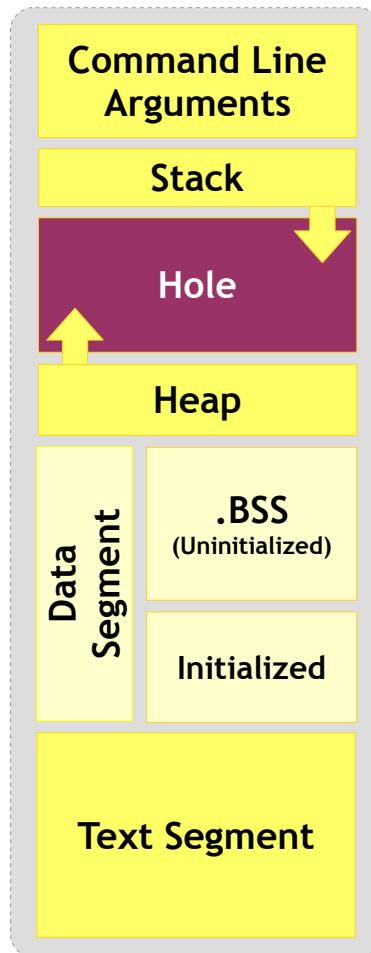
Memory Segments



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

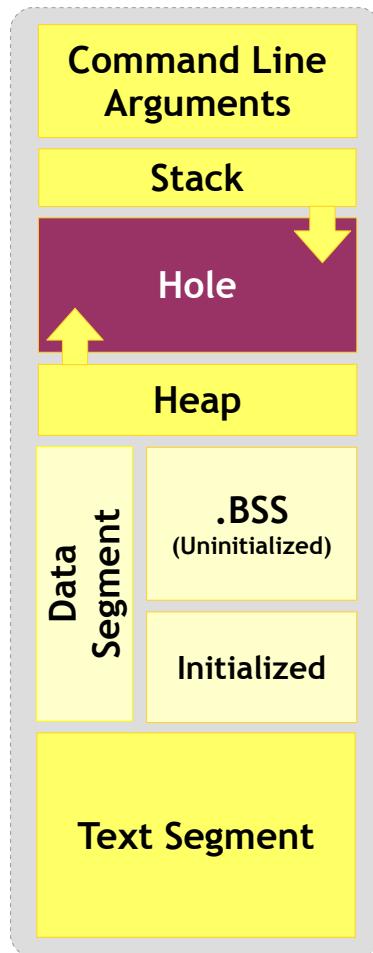
A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack

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

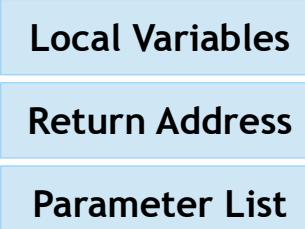
Advanced C

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("%i %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;

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

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 increase 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



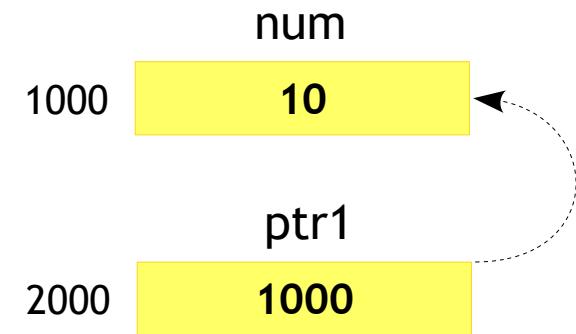
001_example.c

```
#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



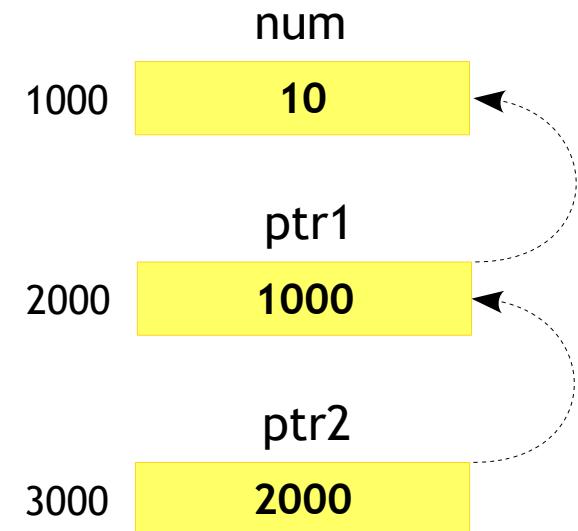
001_example.c

```
#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



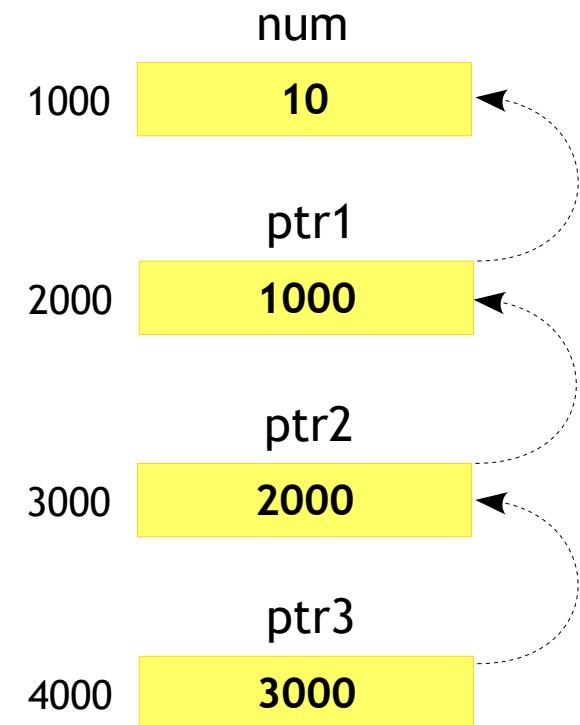
001_example.c

```
#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



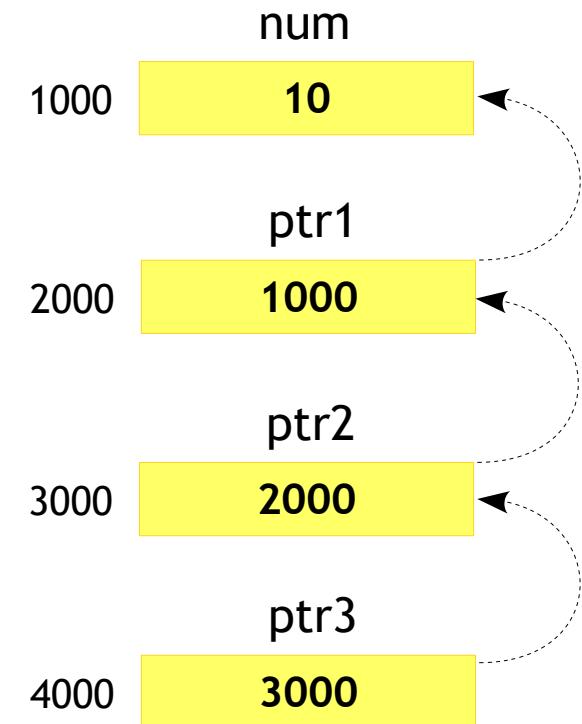
001_example.c

```
#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



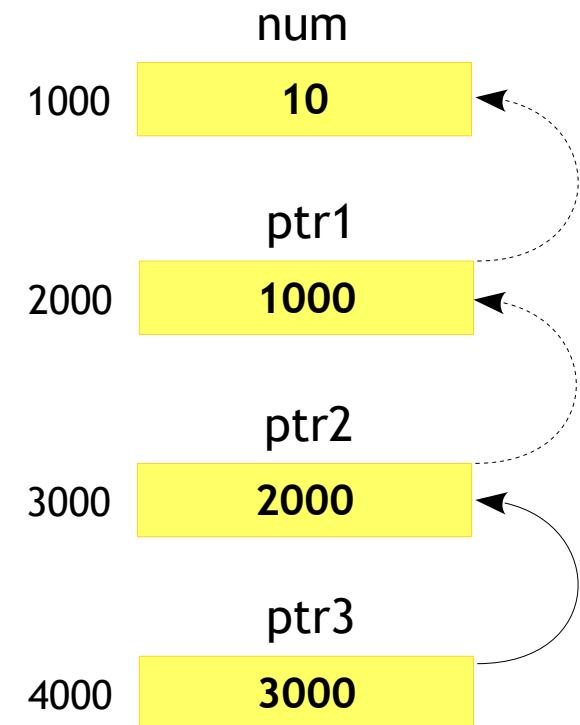
001_example.c

```
#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



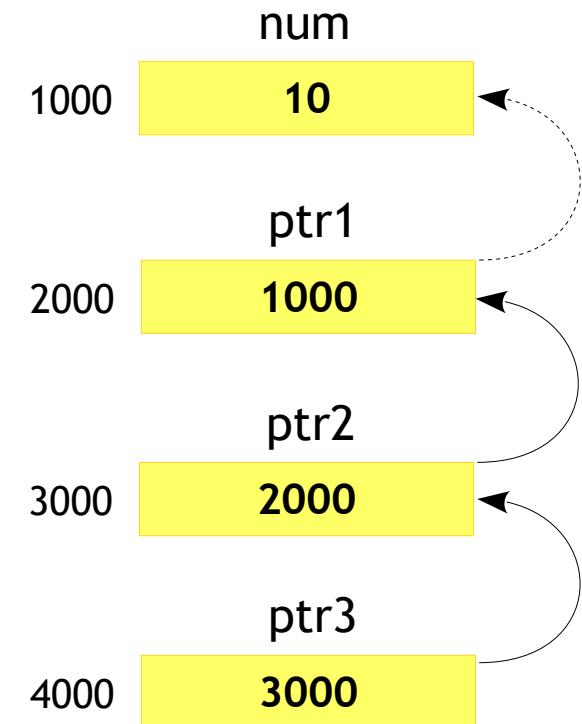
001_example.c

```
#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



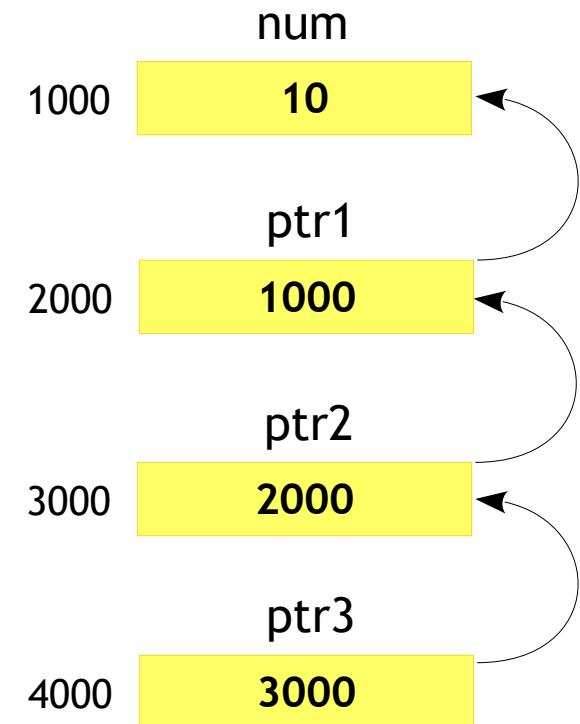
001_example.c

```
#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>

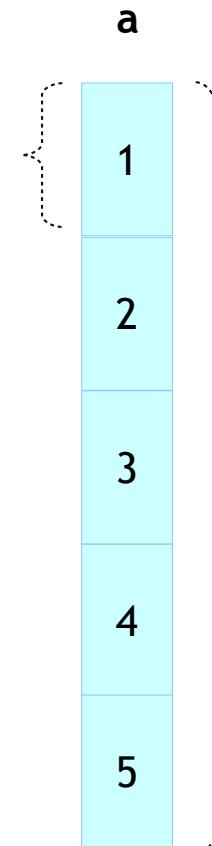
int main()
{
    int a[5] = {1, 2, 3, 4, 5};

    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

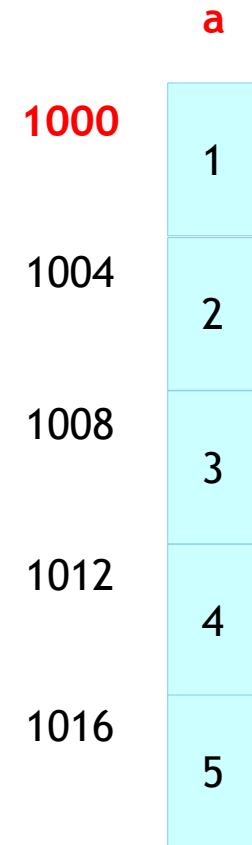
002_example.c

```
#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

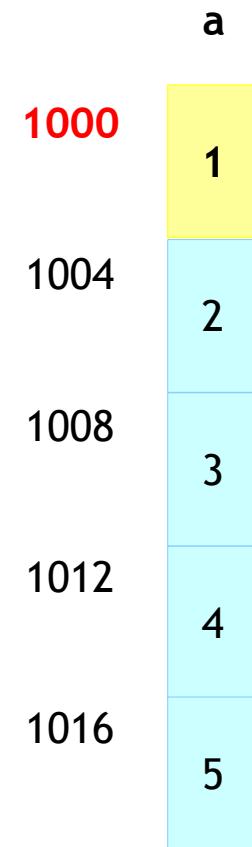
002_example.c

```
#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

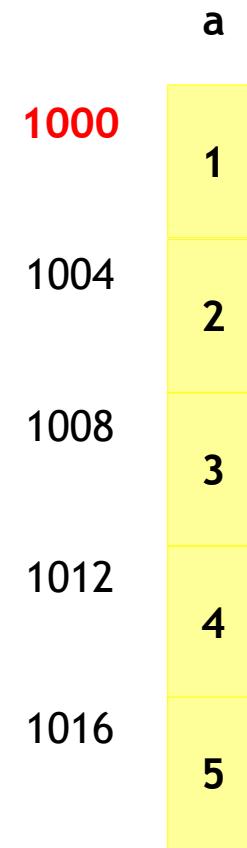
002_example.c

```
#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

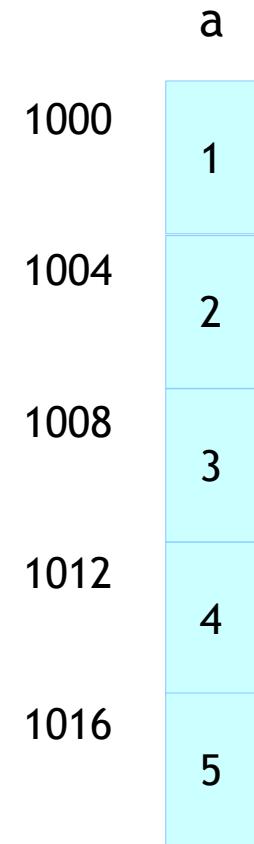
003_example.c

```
#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

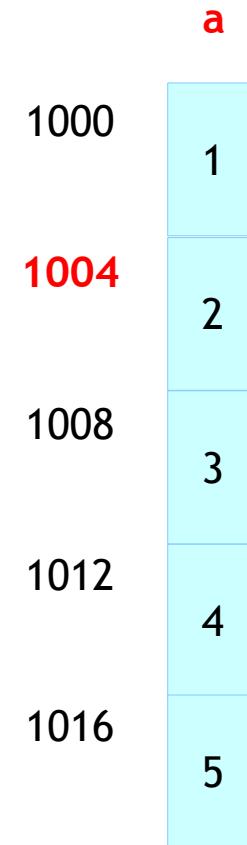
003_example.c

```
#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

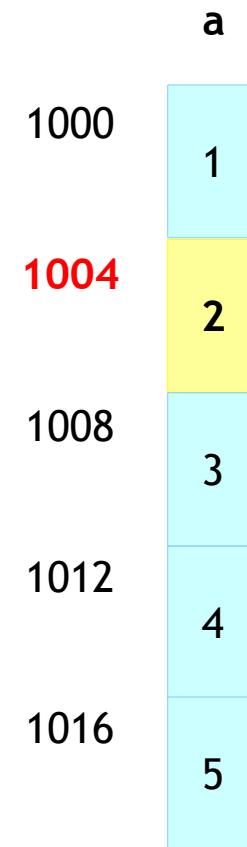
003_example.c

```
#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

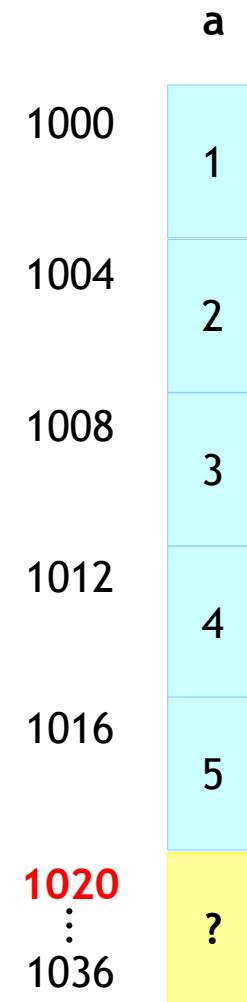
003_example.c

```
#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;
}
```



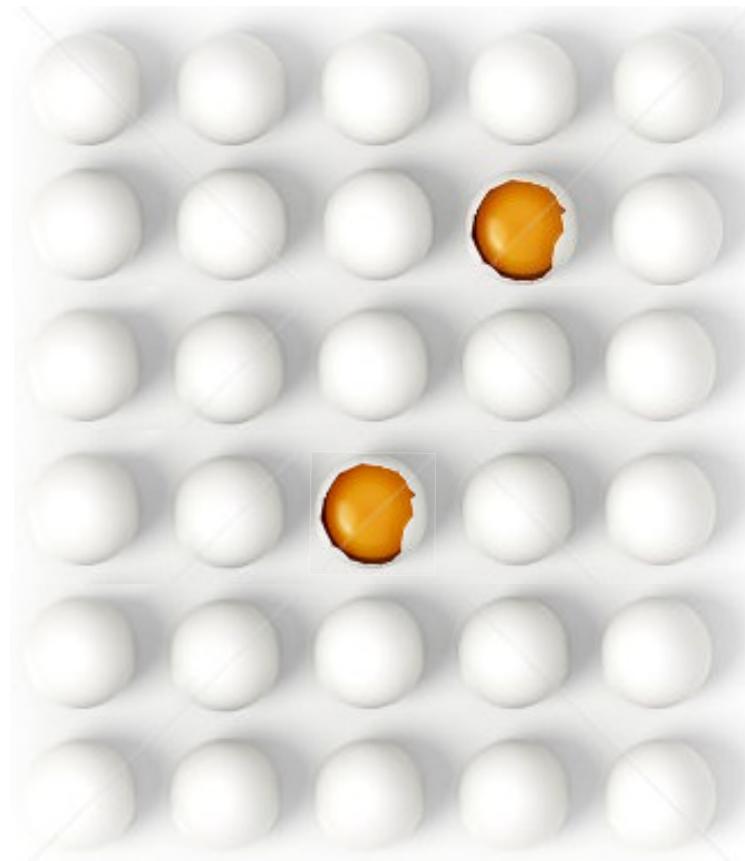
Advanced C

Arrays - Interpretations



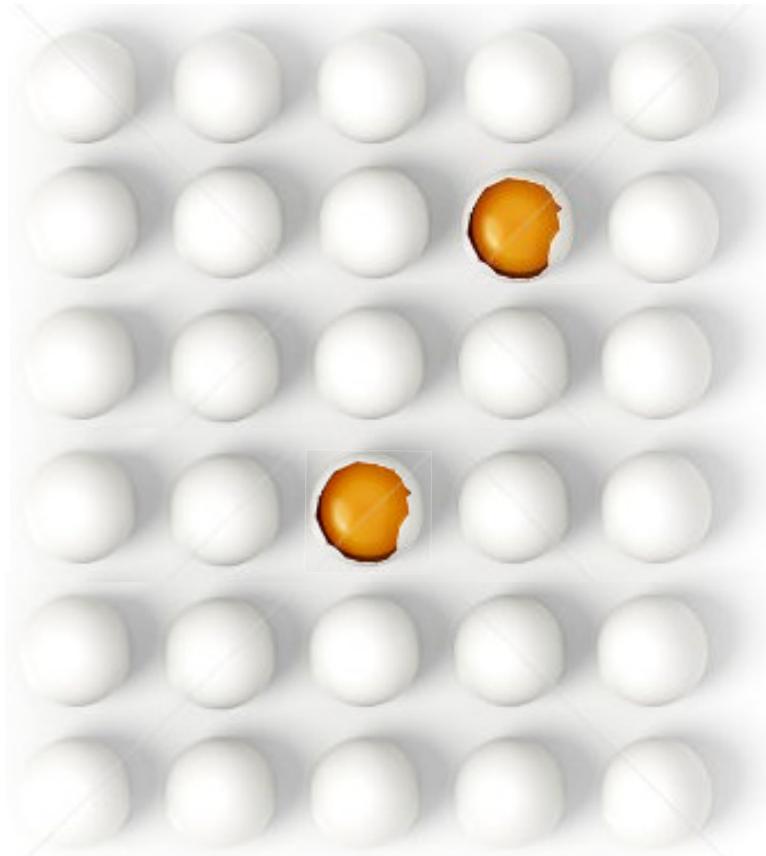
- So in summary, if we try to print the address of `a[]`
 - `a` - prints the value of the constant pointer
 - `&a[0]` - prints the address of the first element pointed by `a`
 - `&a` - prints the address of the whole array which pointed by `a`
- Hence all the lines will print `1000` as output
- These 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

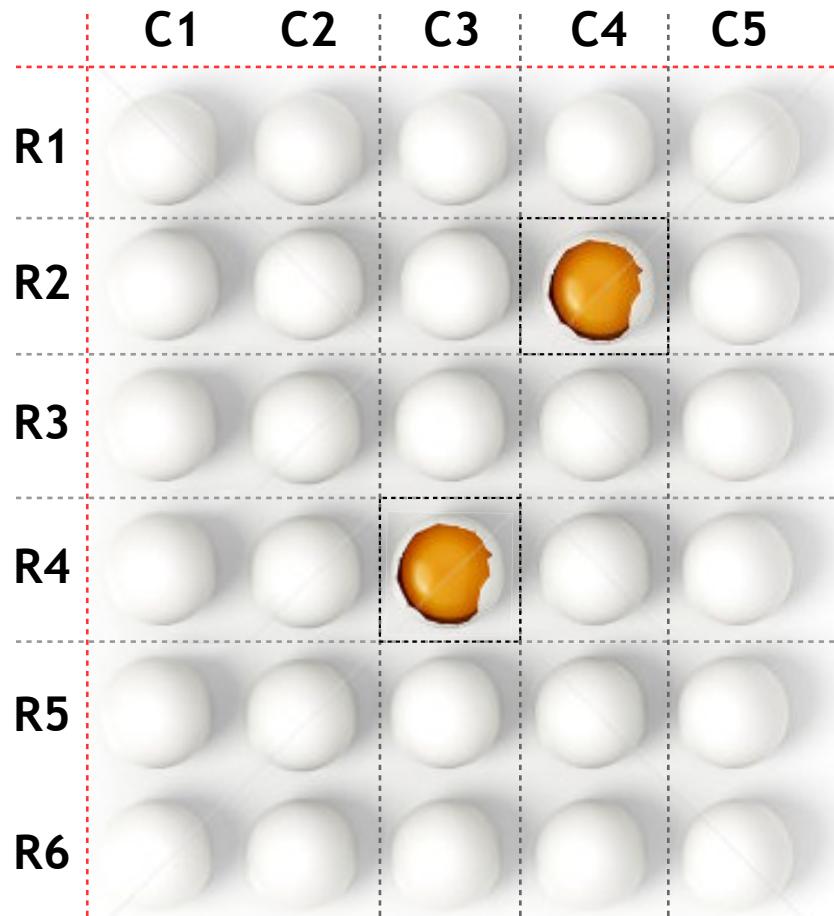
Arrays - 2D

	C1	C2	C3	C4	C5
R1					
R2					
R3					
R4					
R5					
R6					

- Now is it better to tell which one broken??

Advanced C

Arrays - 2D



- 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

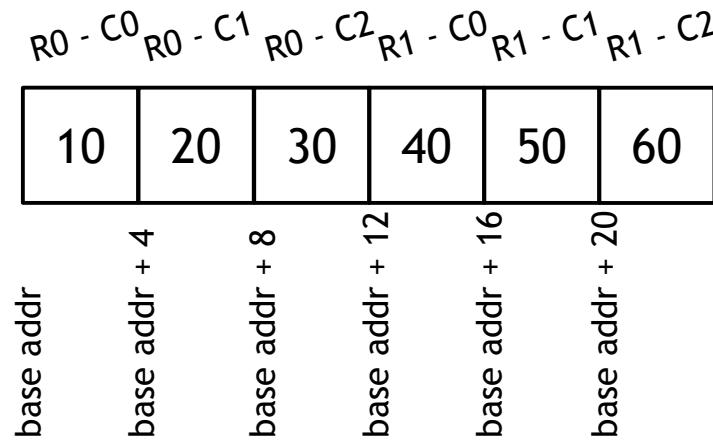
```
data_type name[ROW][COL];
```

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

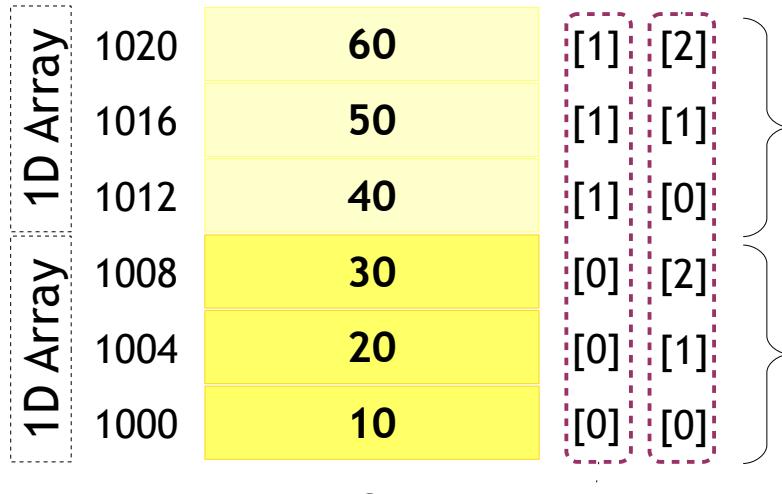
```
int a[2][3] = {{10, 20, 30}, {40, 50, 60}};
```



Advanced C

Arrays - 2D - Referencing

2 * 1D array linearly placed in memory



Index to access the
1D array

2nd 1D Array with base address 1012
 $a[1] = \&a[1][0] = a + 1 \rightarrow 1012$

1st 1D Array with base address 1000
 $a[0] = \&a[0][0] = a + 0 \rightarrow 1000$

Advanced C

Arrays - 2D - Dereferencing

Core Principle

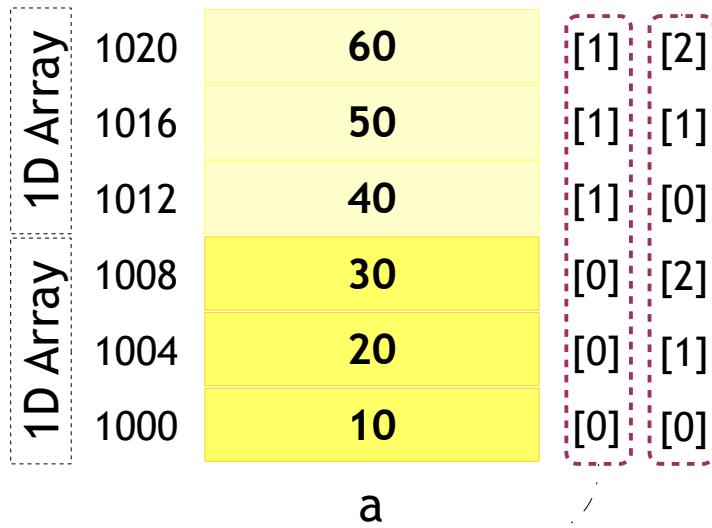
- Dereferencing n^{th} - dimensional array will return $(n - 1)^{\text{th}}$ -dimensional array
 - Example : dereferencing 2D array will return 1D array
- Dereferencing 1D array will return 'data element'
 - Example : Dereferencing 1D integer array will return integer

Array	Dimension
<code>&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,

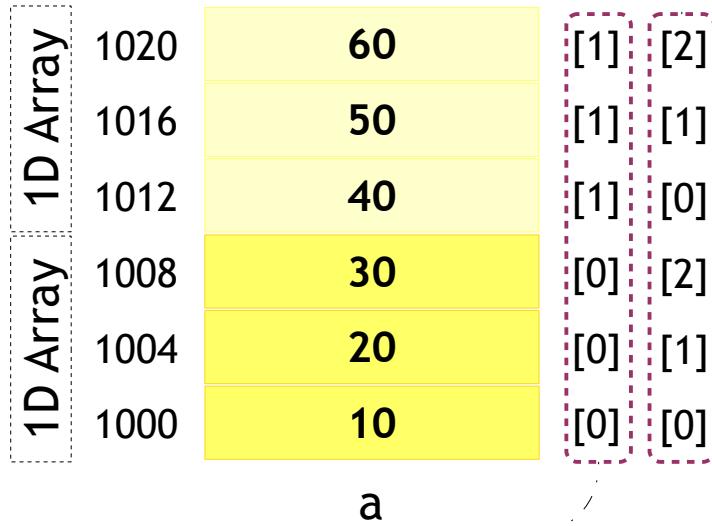
$a[0][1] =$

$$\begin{aligned} &= *(a[0] + (1 * \text{sizeof(type)})) \\ &= *(*a + (0 * \text{sizeof(1D array)})) + (1 * \text{sizeof(type)}) \\ &= *(*a + (0 * 12)) + (1 * 4) \\ &= *(*a + 0) + 4 \\ &= *(a + 0) + 4 \\ &= *(a + 4) \\ &= *(1000 + 4) \\ &= *(1004) \\ &= 20 \end{aligned}$$

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,

$a[1][1] =$

$$\begin{aligned} &= *(a[1] + (1 * \text{sizeof(type)})) \\ &= *(*(a + (1 * \text{sizeof(1D array)})) + (1 * \text{sizeof(type)})) \\ &= *(*(a + (1 * 12)) + (1 * 4)) \\ &= *(*(a + 12) + 4) \\ &= *(a + 12 + 4) \\ &= *(a + 16) \\ &= *(1000 + 16) \\ &= *(1016) \\ &= 50 \end{aligned}$$

Address of $a[r][c] = \text{value}(a) + r * \text{sizeof(1D array)} + c * \text{sizeof(type)}$

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;
}
```

a	
1000	10
b	
1004	20
c	
1008	30

Advanced C

Pointers - Array of pointers



Syntax

```
datatype *ptr_name[SIZE]
```

004_example.c

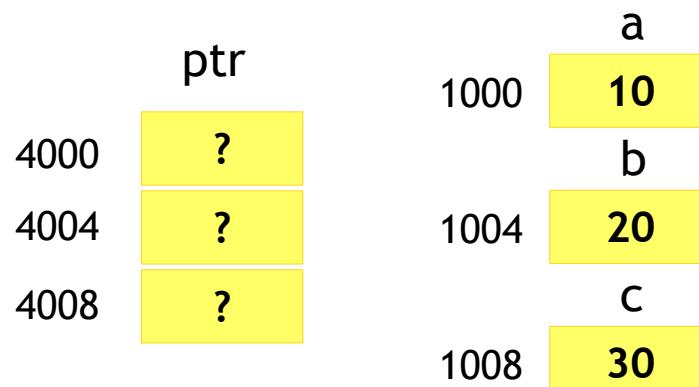
```
#include <stdio.h>

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

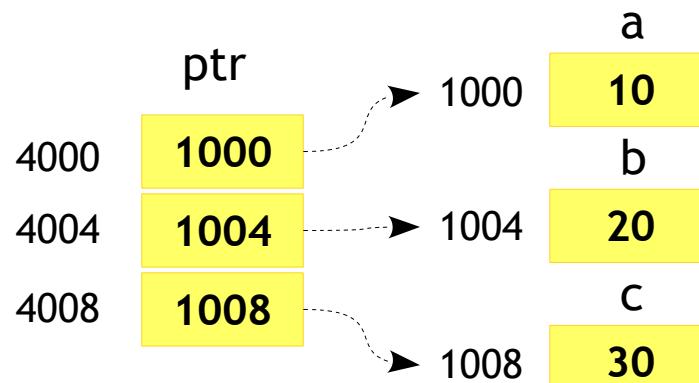
```
#include <stdio.h>

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;
}
```

	1000	1004
a	10	20
	1008	1012
b	30	40
	1016	1020
c	50	60

Advanced C

Pointers - Array of pointers



Syntax

```
datatype *ptr_name[SIZE]
```

005_example.c

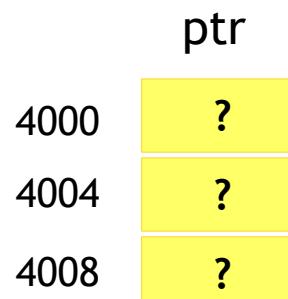
```
#include <stdio.h>
```

```
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;
}
```



	1000	1004
a	10	20
	1008	1012
b	30	40
	1016	1020
c	50	60

Advanced C

Pointers - Array of pointers



Syntax

```
datatype *ptr_name[SIZE]
```

005_example.c

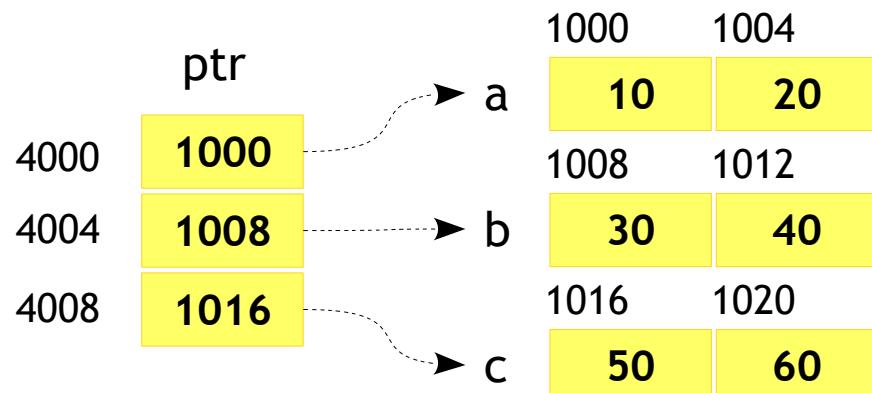
```
#include <stdio.h>

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;
}
```

a	
1000	10
b	
1004	20
c	
1008	30

Advanced C

Pointers - Array of pointers

006_example.c

```
#include <stdio.h>

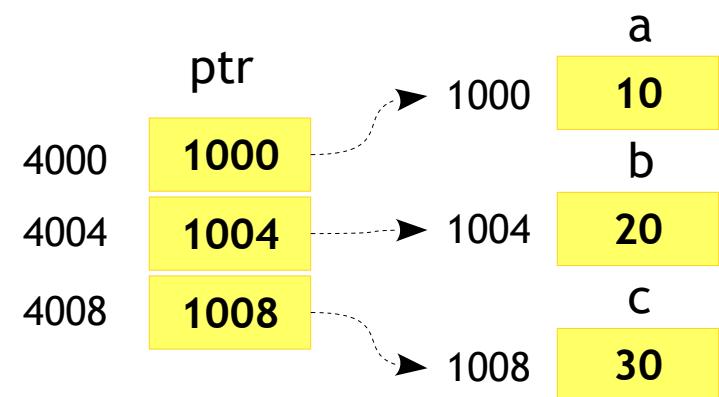
void print_array(int **p)
{
    int i;

    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>

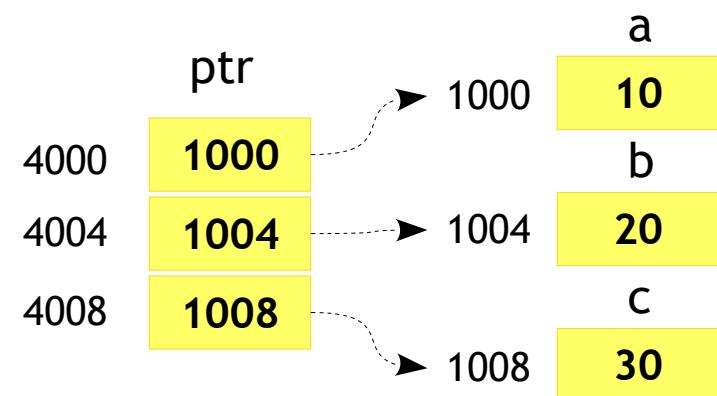
void print_array(int **p)
{
    int i;

    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>

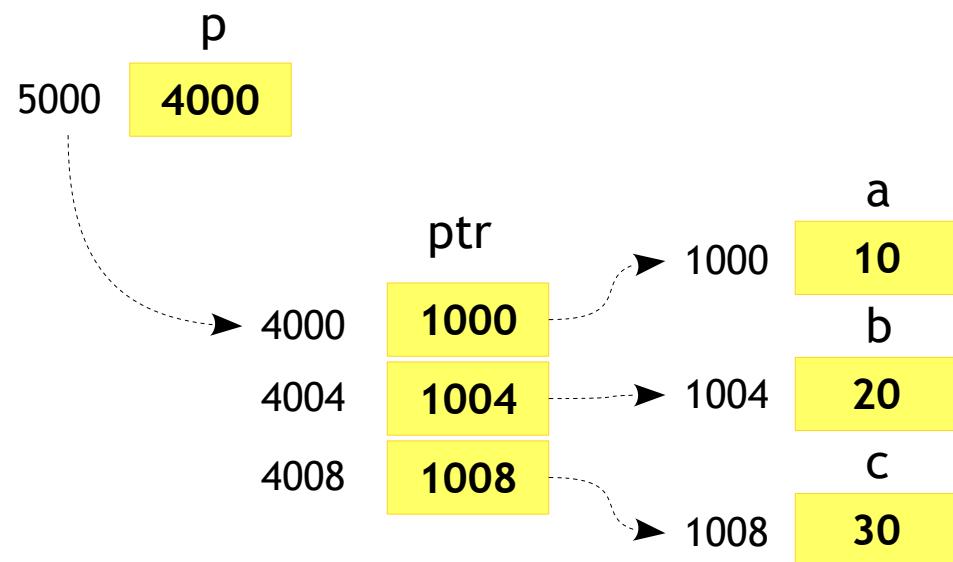
void print_array(int **p)
{
    int i;

    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

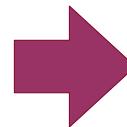
```
#include <stdio.h>

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

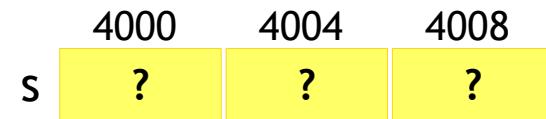
```
#include <stdio.h>

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

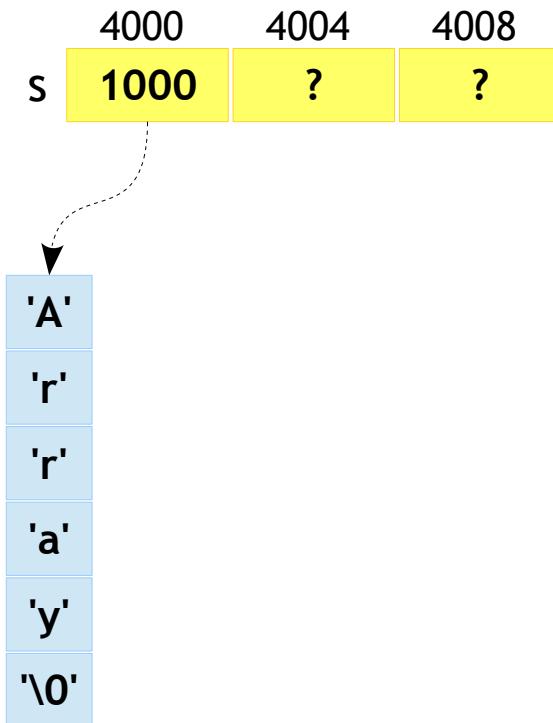
```
#include <stdio.h>

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

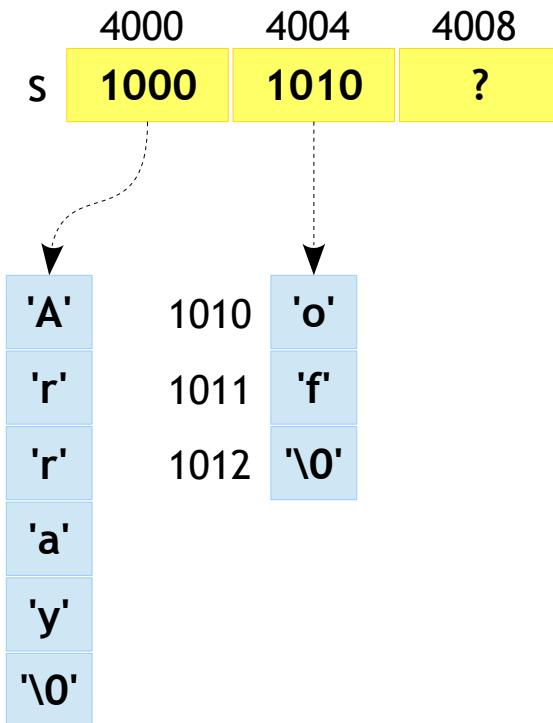
```
#include <stdio.h>

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

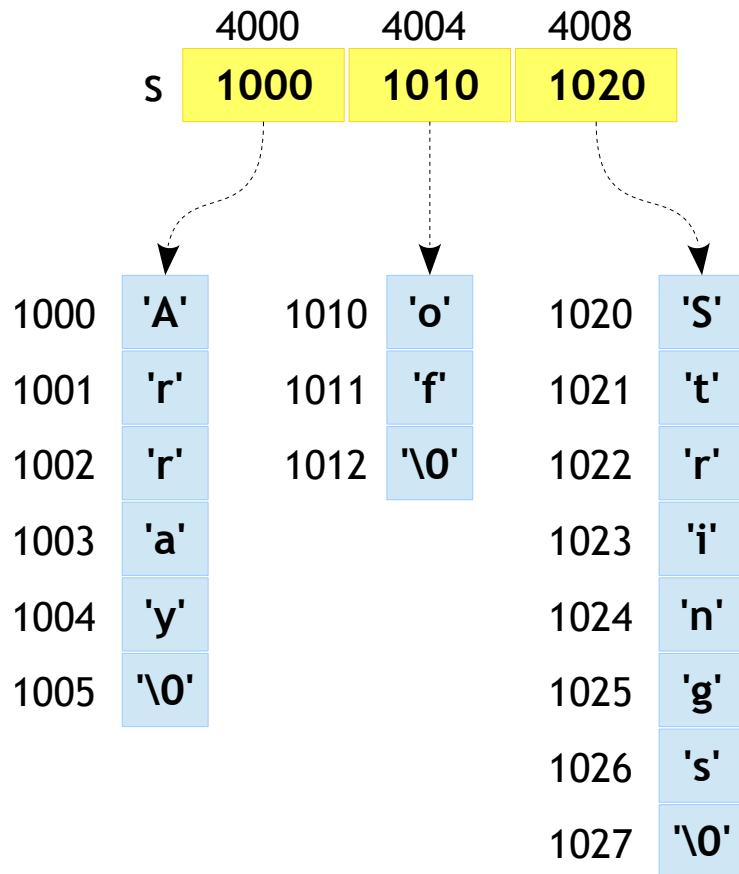
```
#include <stdio.h>

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

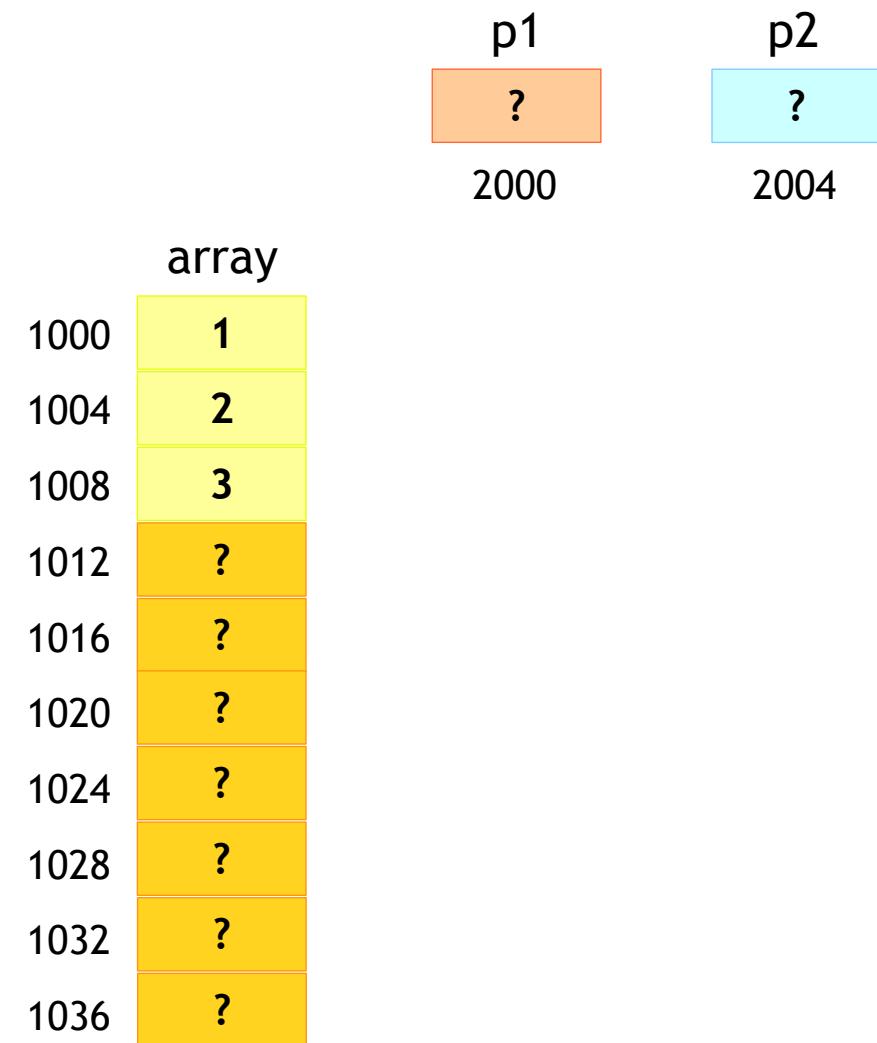
012_example.c

```
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;
}
```



Advanced C

Pointers - Pointer to an Array

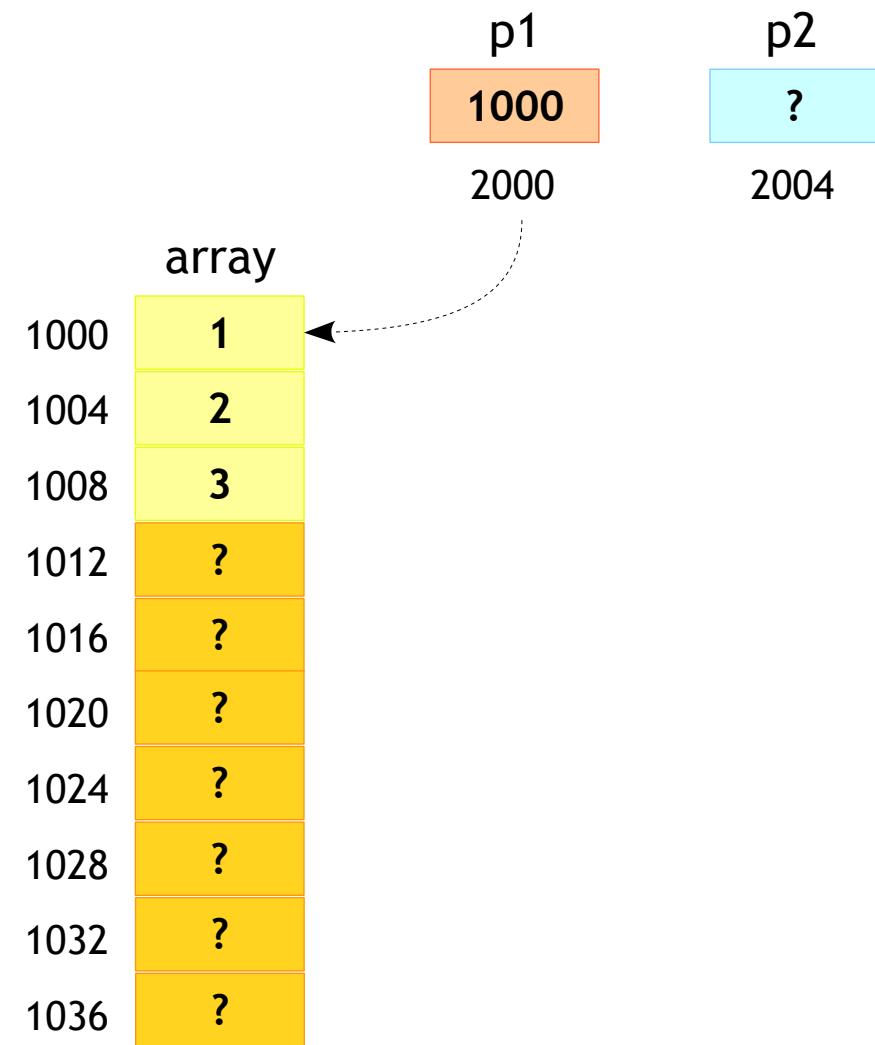
012_example.c

```
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

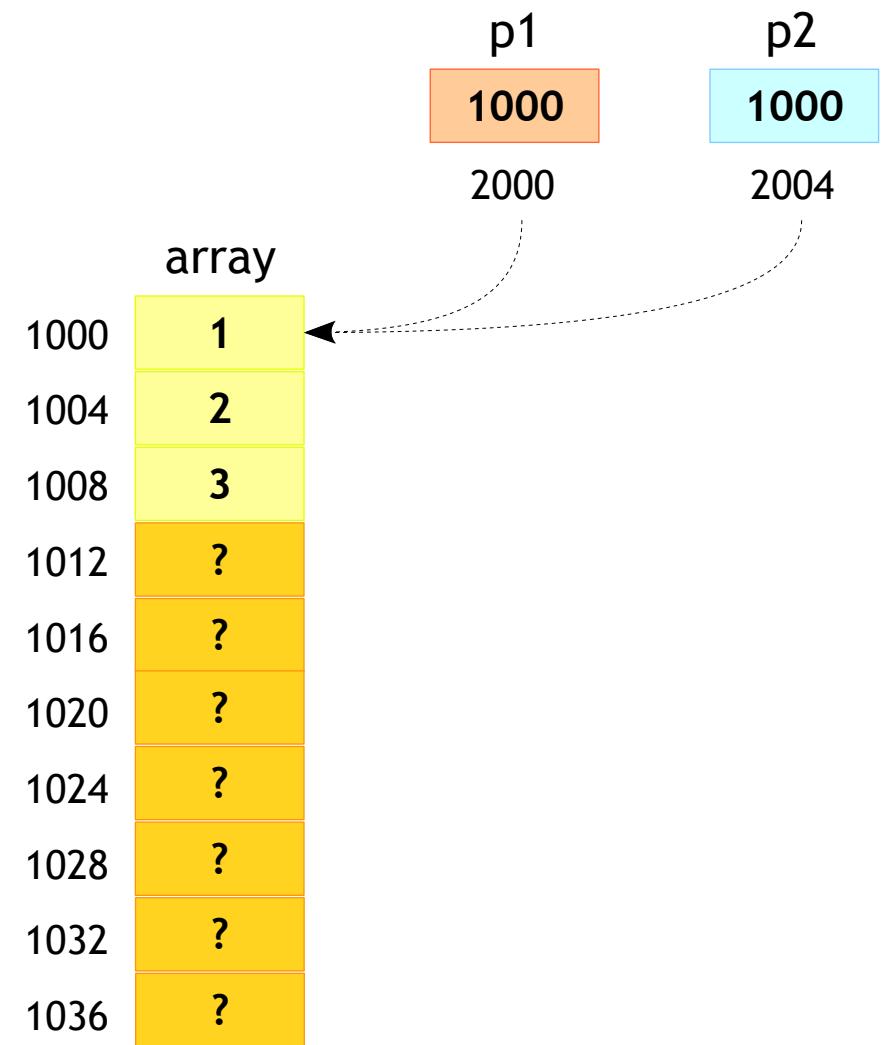
012_example.c

```
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

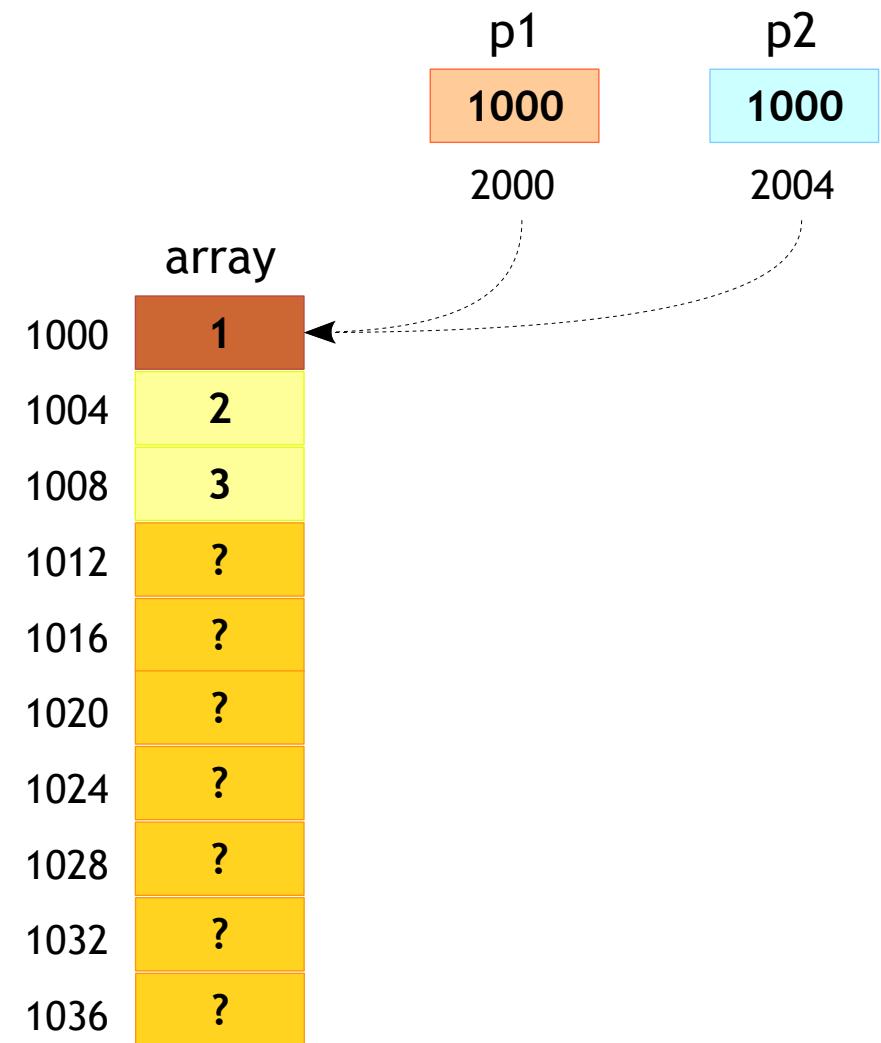
012_example.c

```
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



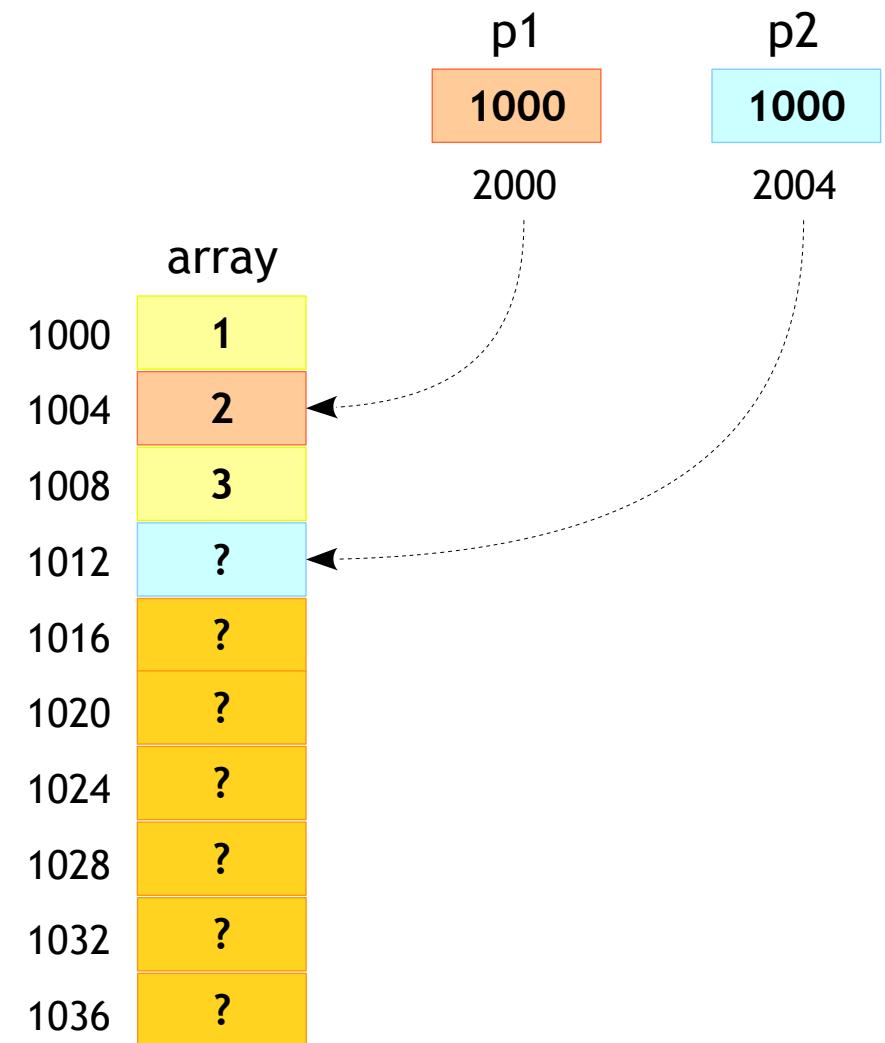
012_example.c

```
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

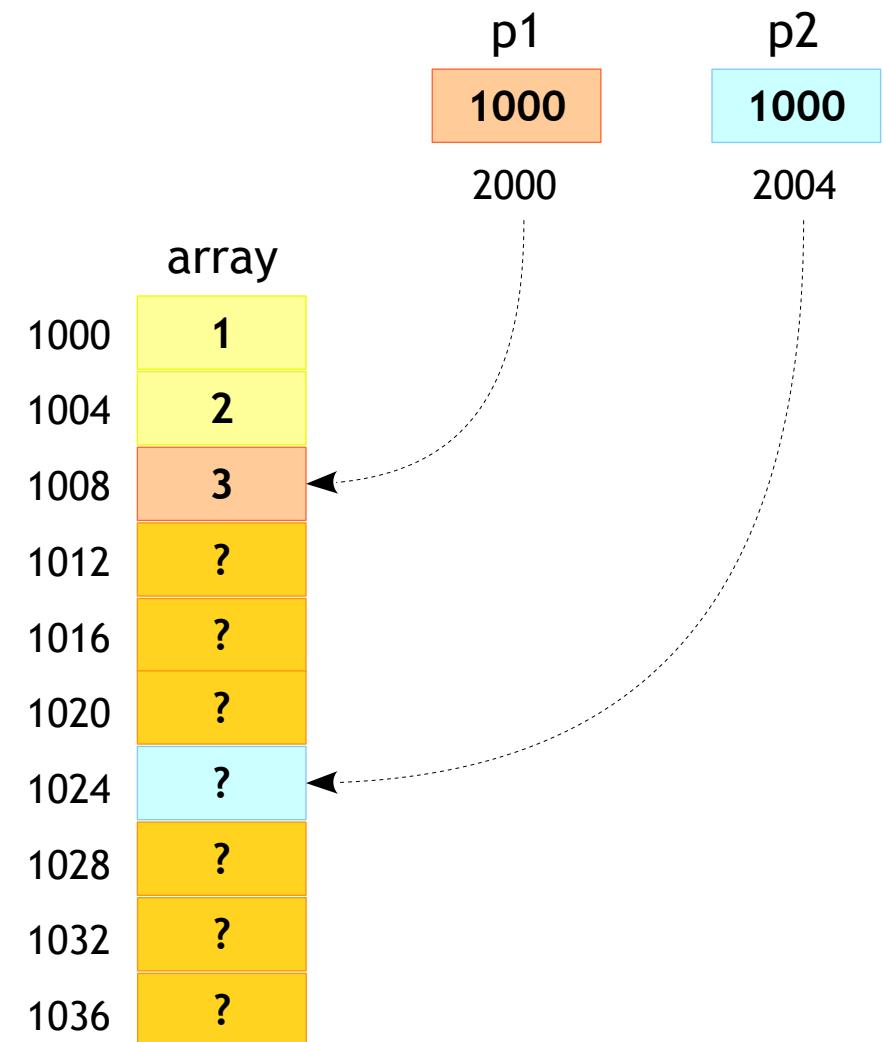
012_example.c

```
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

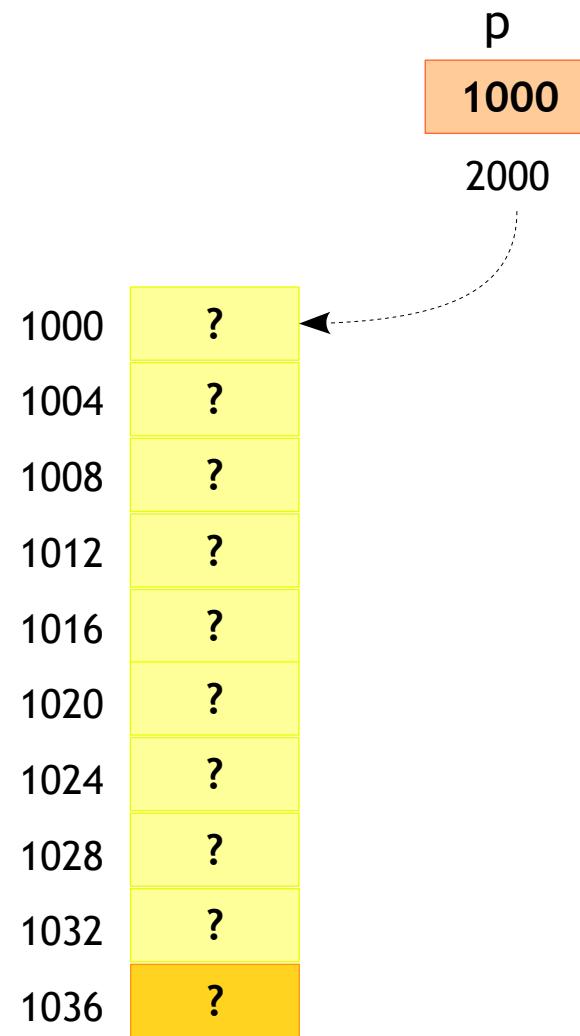
```
int main()
{
    int (*p)[3];

    → 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

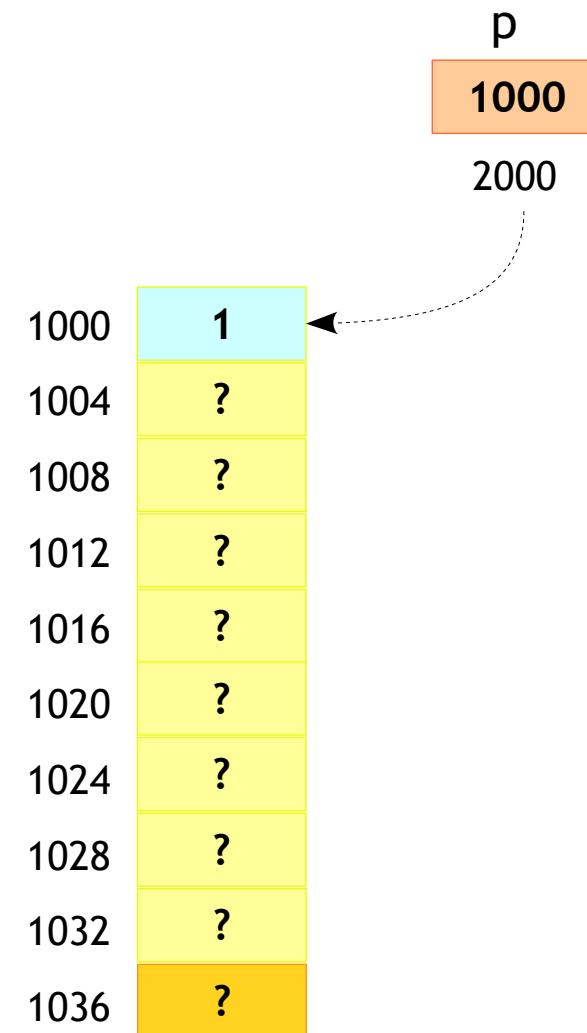
```
int main()
{
    int (*p)[3];

    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

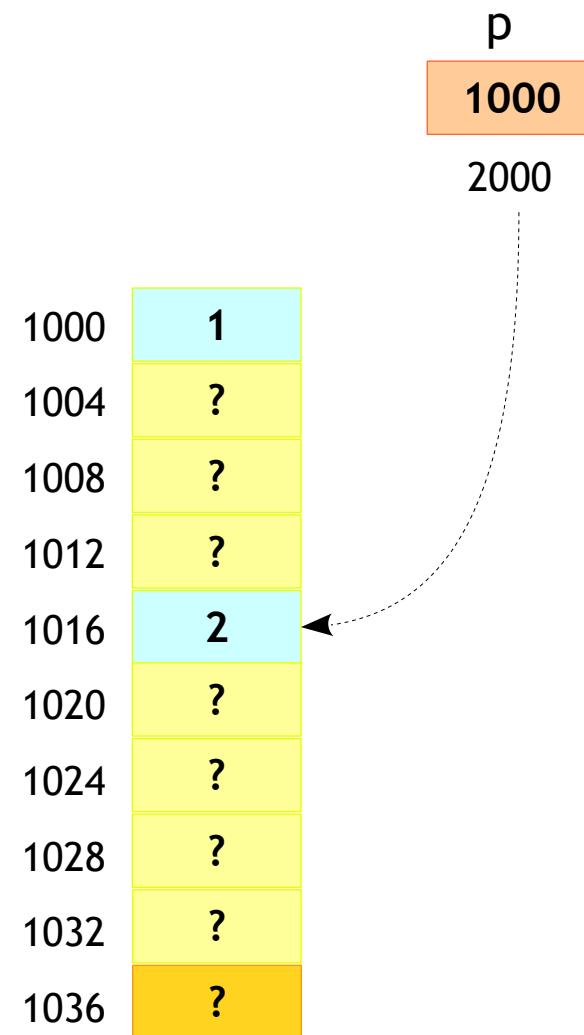
```
int main()
{
    int (*p)[3];

    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

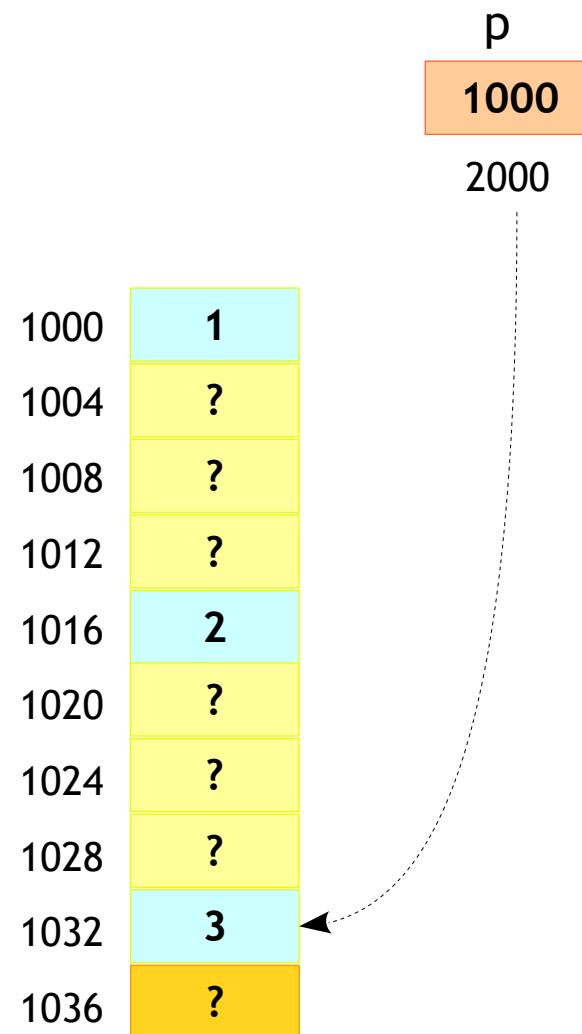
```
int main()
{
    int (*p)[3];

    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;
}
```



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

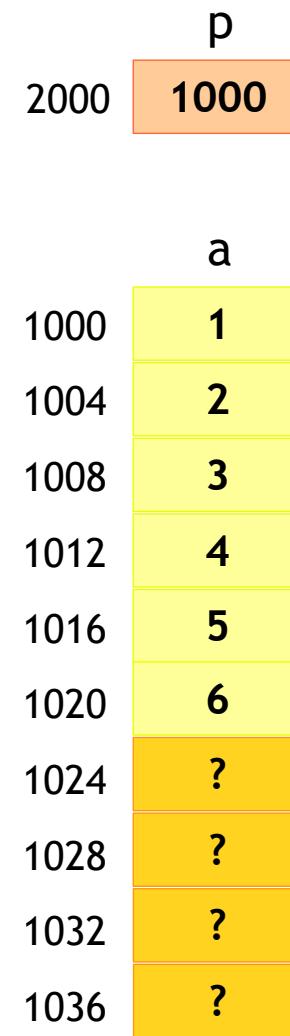
Pointers - Pointer to an 2D Array

014_example.c

```
int main()
{
    int (*p)[3];
→ int a[2][3] = {{1, 2, 3}, {4, 5, 6}};

    p = a;

    return 0;
}
```



Advanced C

Pointers - Pointer to an 2D Array

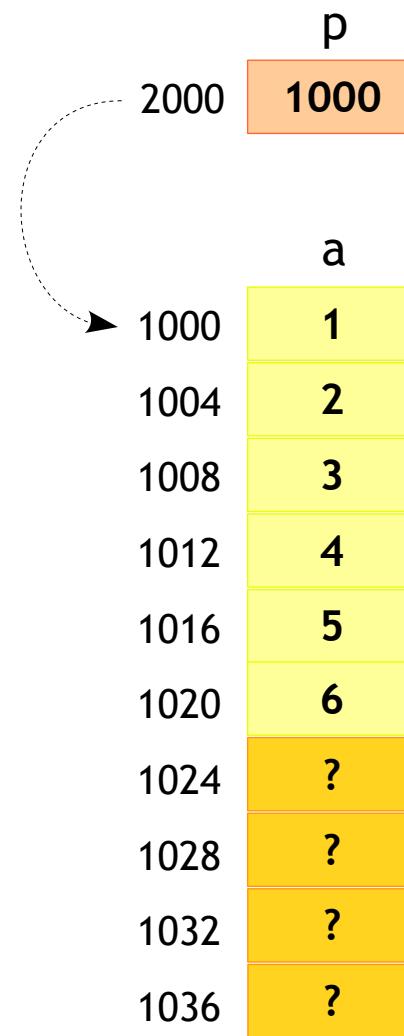


014_example.c

```
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>

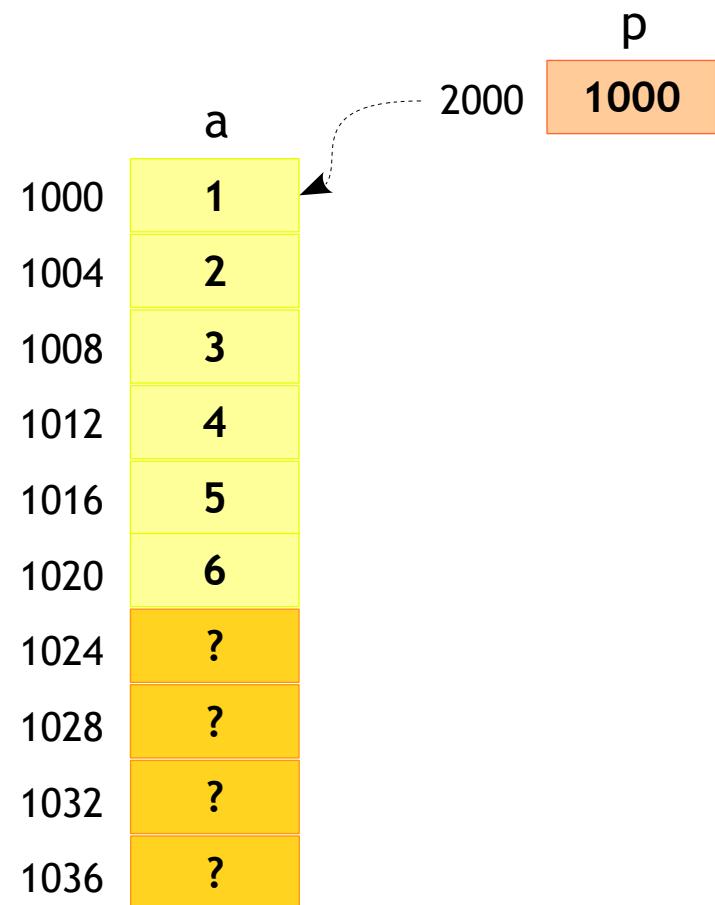
void print_array(int p[2][3])
{
    int i, j;

    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>

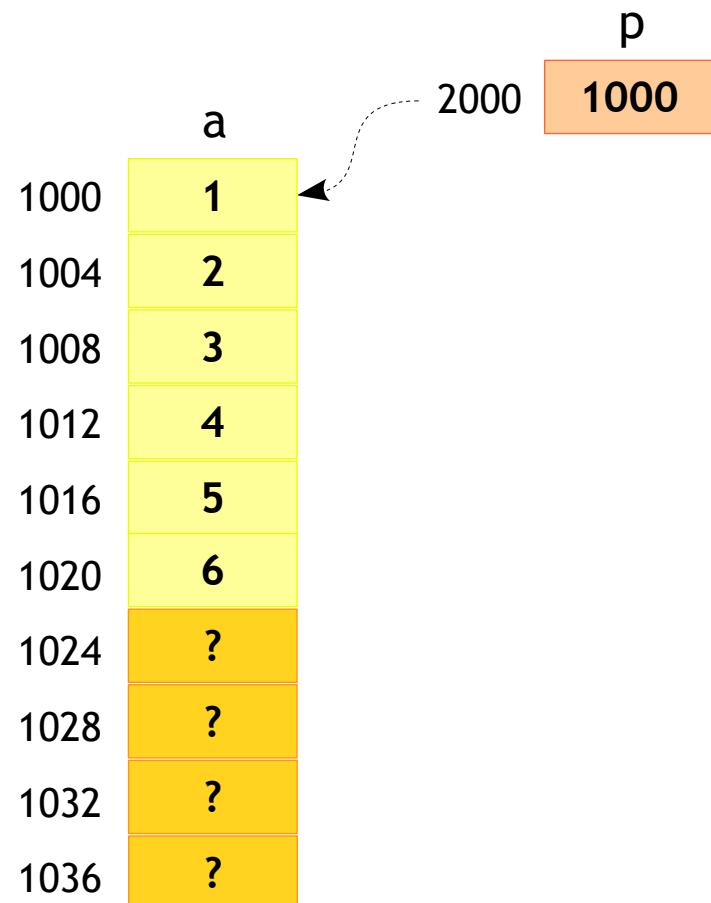
void print_array(int (*p) [3])
{
    int i, j;

    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;
}
```

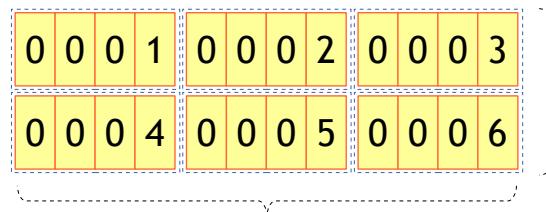
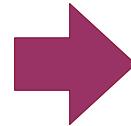
- Both Static (BS)
- Called as an rectangular array
- Total size is
$$2 * 3 * \text{sizeof(datatype)}$$
$$2 * 3 * 4 = 24 \text{ Bytes}$$
- The memory representation can be as shown in next slide

Advanced C

Pointers - 2D Array Creations - BS



a
1000
1004
1008
1012
1016
1020



Static
3 Columns
On Stack

Static
2 Rows
On Stack

Advanced C

Pointers - 2D Array Creations - FSSD

019_example.c

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

int main()
{
    int *a[2];

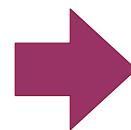
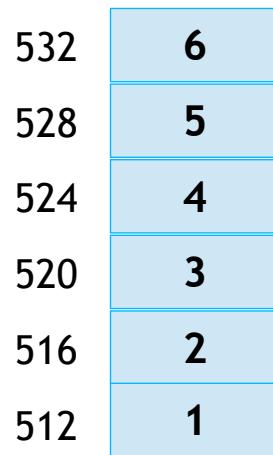
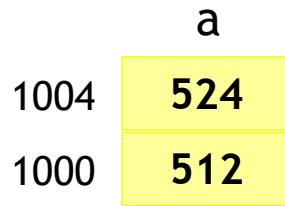
    for ( i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

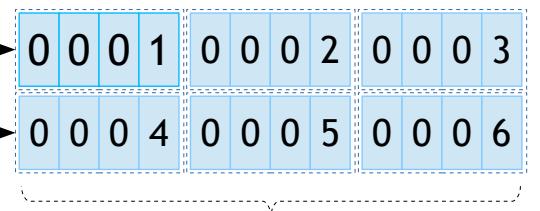
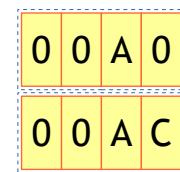
- First Static and Second Dynamic (FSSD)
- Mix of Rectangular & Ragged
- Total size is
$$2 * \text{sizeof(datatype *)} + \\ 2 * 3 * \text{sizeof(datatype)}$$
$$2 * 4 + 2 * 3 * 4 = 32 \text{ Bytes}$$
- The memory representation can be as shown in next slide

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];

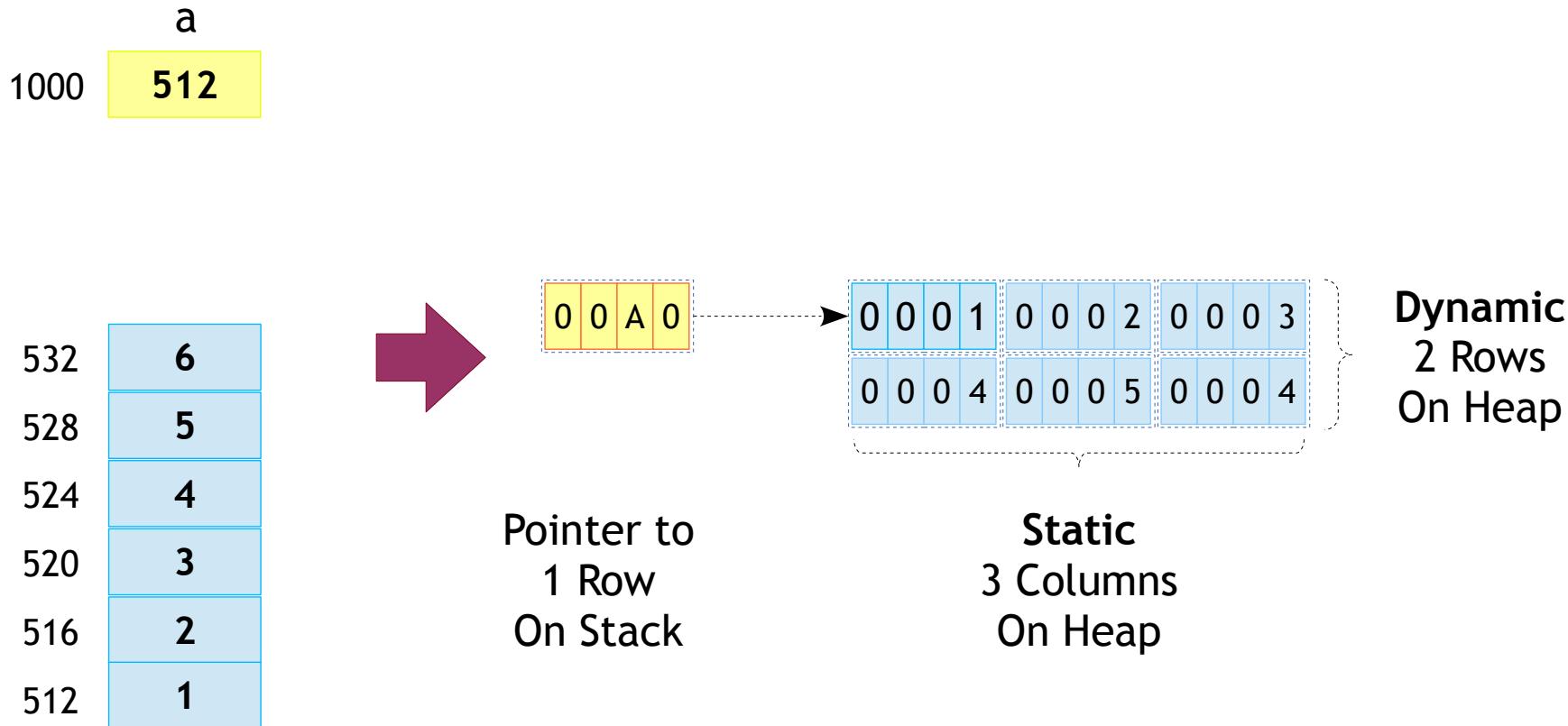
    a = malloc(2 * sizeof(int [3]));

    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



Advanced C

Pointers - 2D Array Creations - BD

021_example.c

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

int main()
{
    int **a;
    int i;

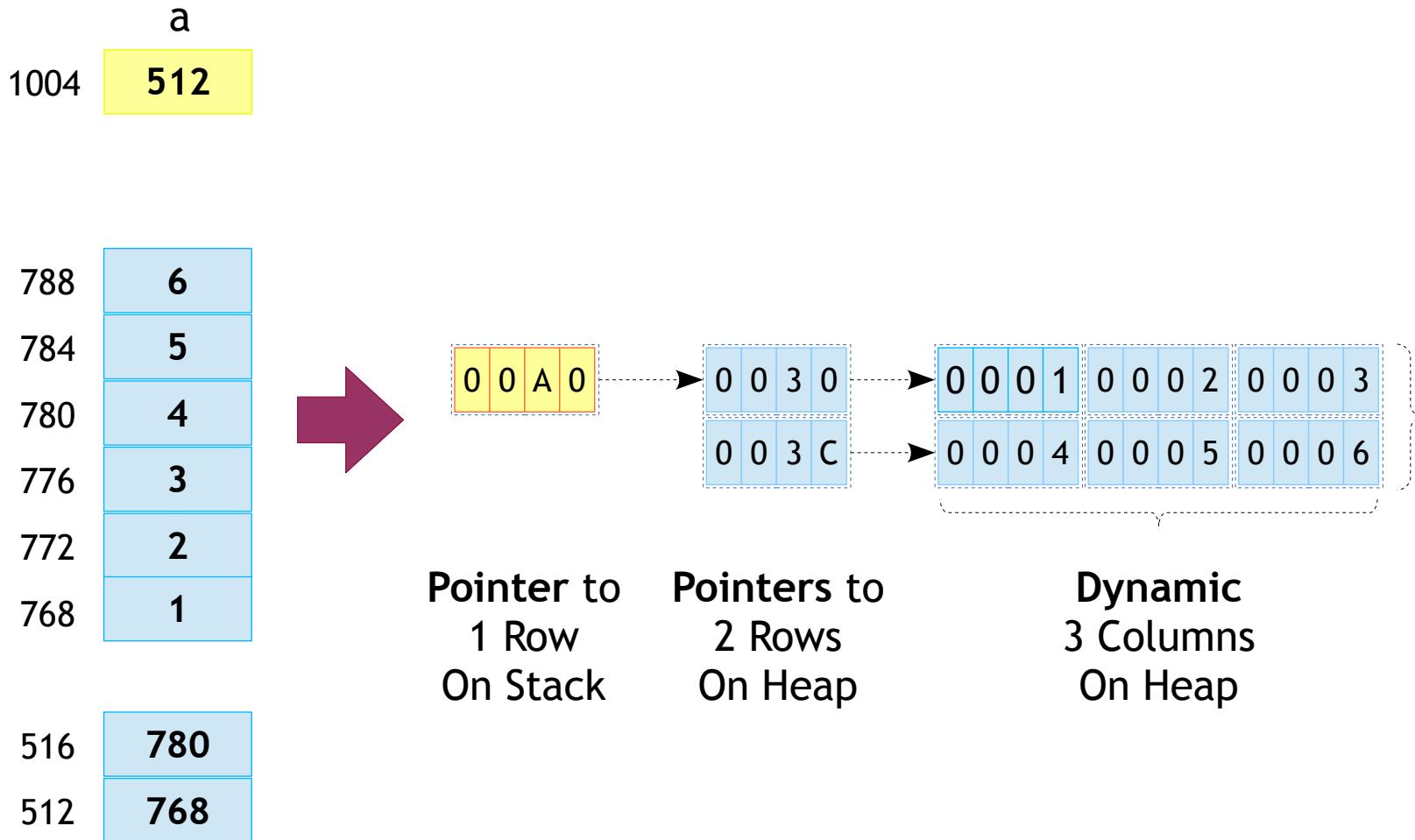
    a = malloc(2 * sizeof(int *));
    for (i = 0; i < 2; i++)
    {
        a[i] = malloc(3 * sizeof(int));
    }

    return 0;
}
```

- Both Dynamic (BD)
- Total size is
 $\text{sizeof(datatype } \star\star) +$
 $2 * \text{sizeof(datatype } \star) +$
 $2 * 3 * \text{sizeof(datatype)}$
 $4 + 2 * 4 + 2 * 3 * 4 = 36$
Bytes
- The memory representation can be as shown in next slide

Advanced C

Pointers - 2D Array Creations - BD



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
```

4th argument

3rd argument

2nd argument

1st 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)
{
    int i;

    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;
}
```

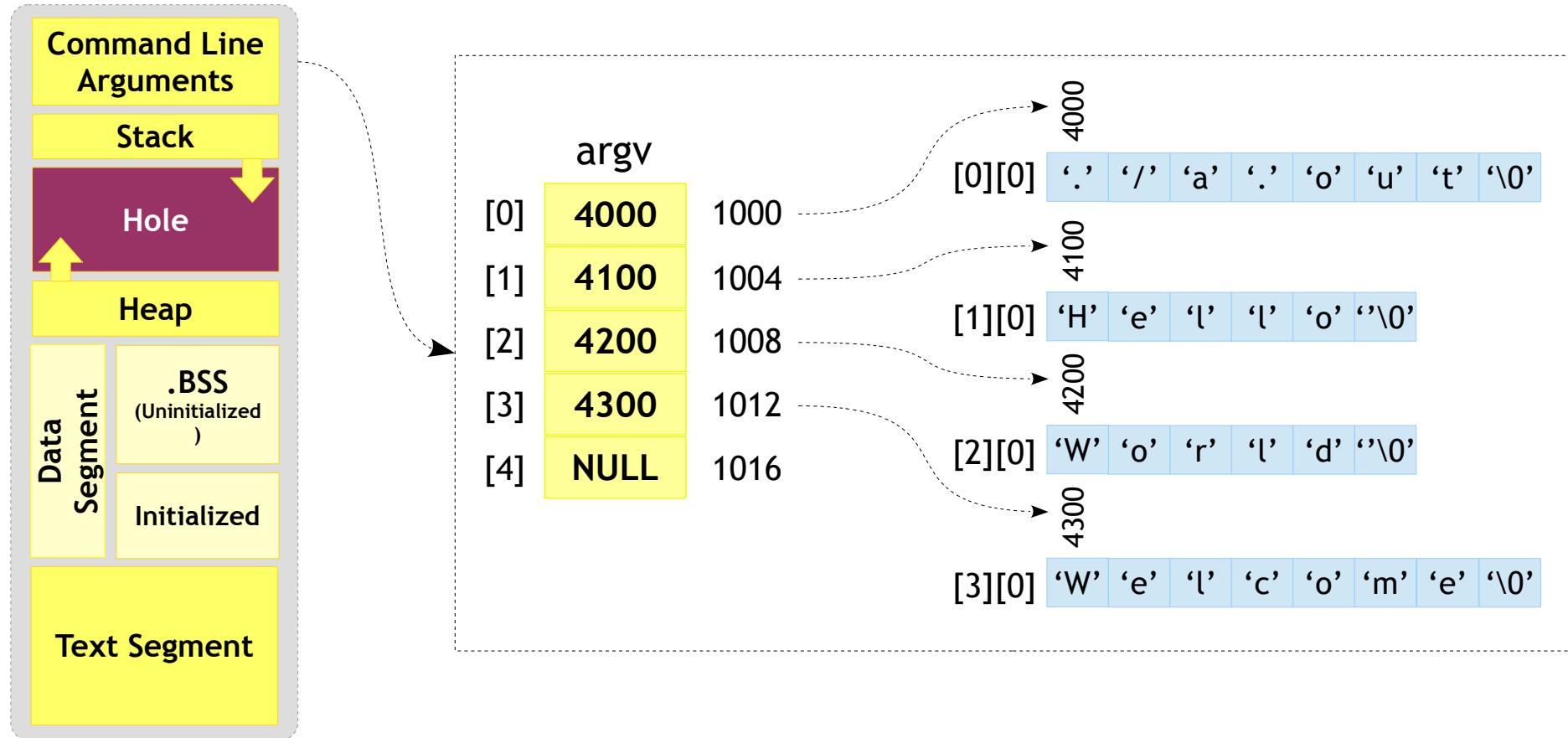
Advanced C

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;
}
```

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

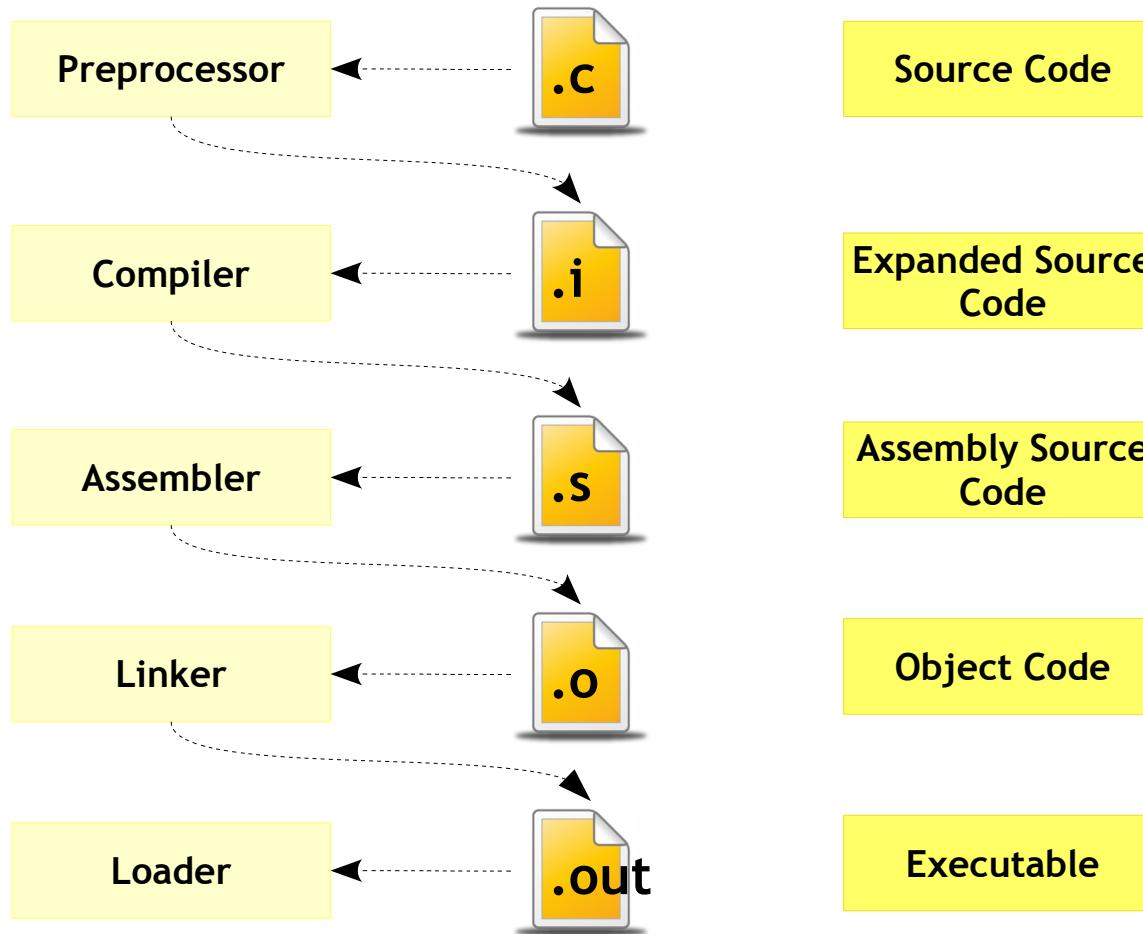
Advanced C

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
#ifndef      #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);
```

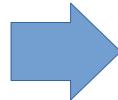
001_file1.c

```
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 001_file1.c 002_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 "003_file2.h"

int main()
{
    puts(test());

    return 0;
}
```

Compile as

```
user@user:~] gcc -E 001_file1.c 002_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 001_file1.c 002_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 Nth 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.
 - **Portability:** A program may need to use different code depending on the machine or operating system it is to run on
 - **Testing:** You may want to be able to compile the same source file into two different programs, like one for debug (**Test**) and other as final (**Production**)
 - **Reference Code:** 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, **012_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



024_example.c

```
#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

025_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

026_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

027_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:
 - Structure (derived)
 - Unions (derived)
 - Typedef (storage class)
 - Enum (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 its memory layout?

Advanced C

UDTs - Structures - Memory Layout

002_example.c

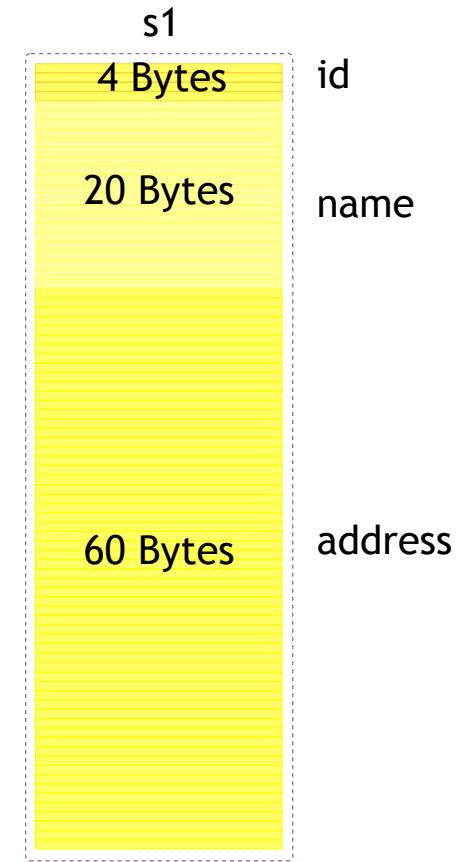
```
#include <stdio.h>

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

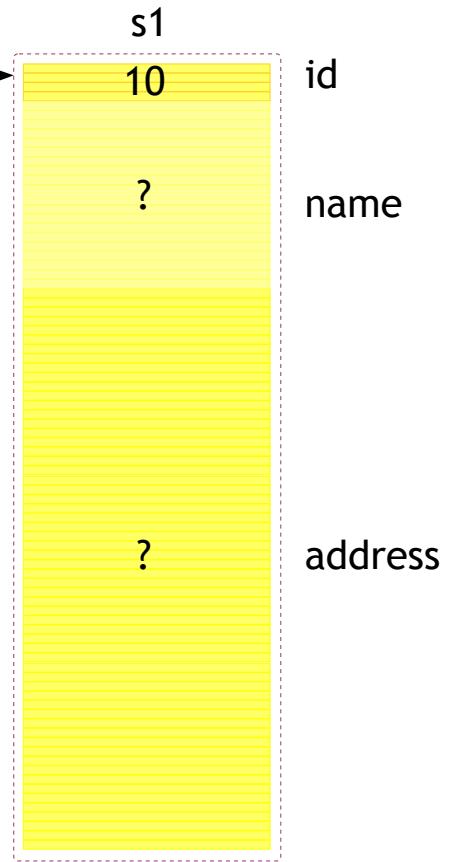
```
#include <stdio.h>

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

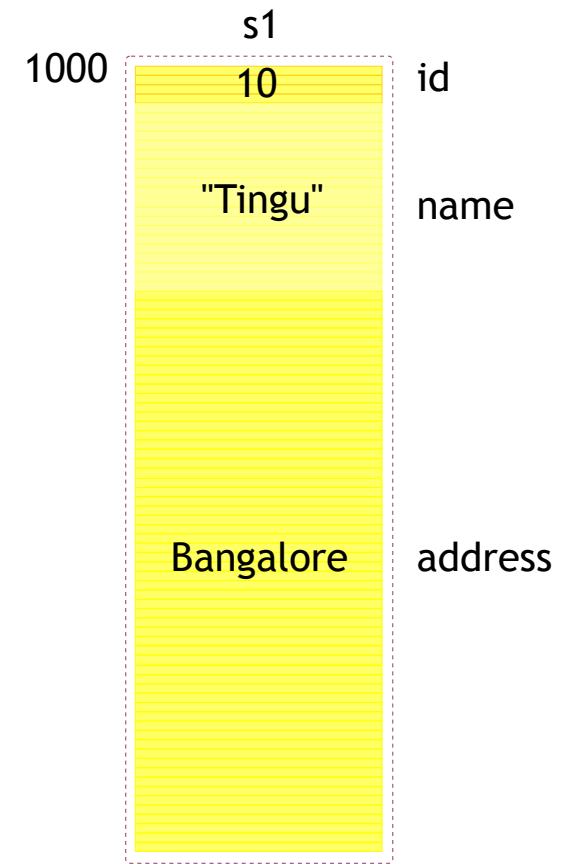
```
#include <stdio.h>

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

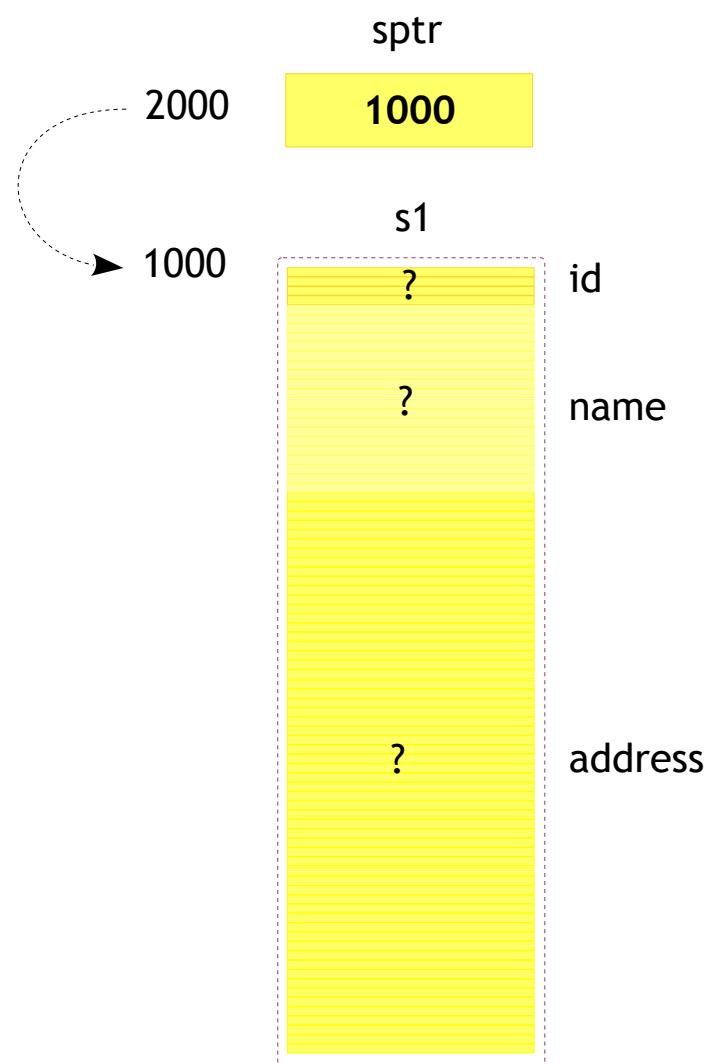
```
#include <stdio.h>

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>

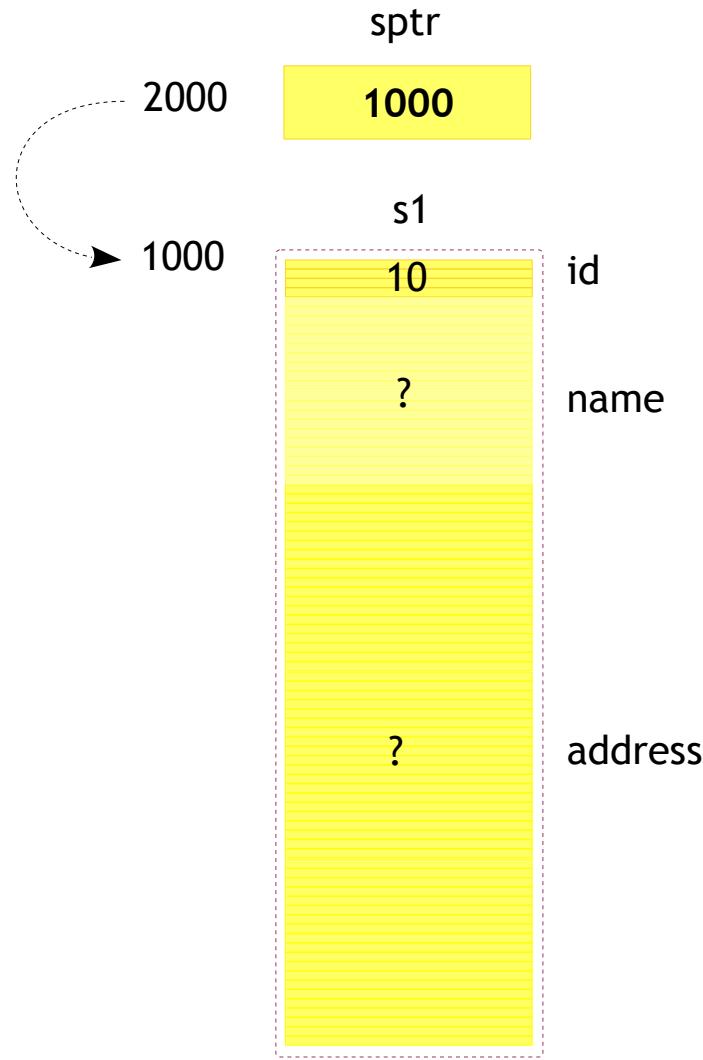
struct Student
{
    int id;
    char name[20];
    char address[60];
};

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>

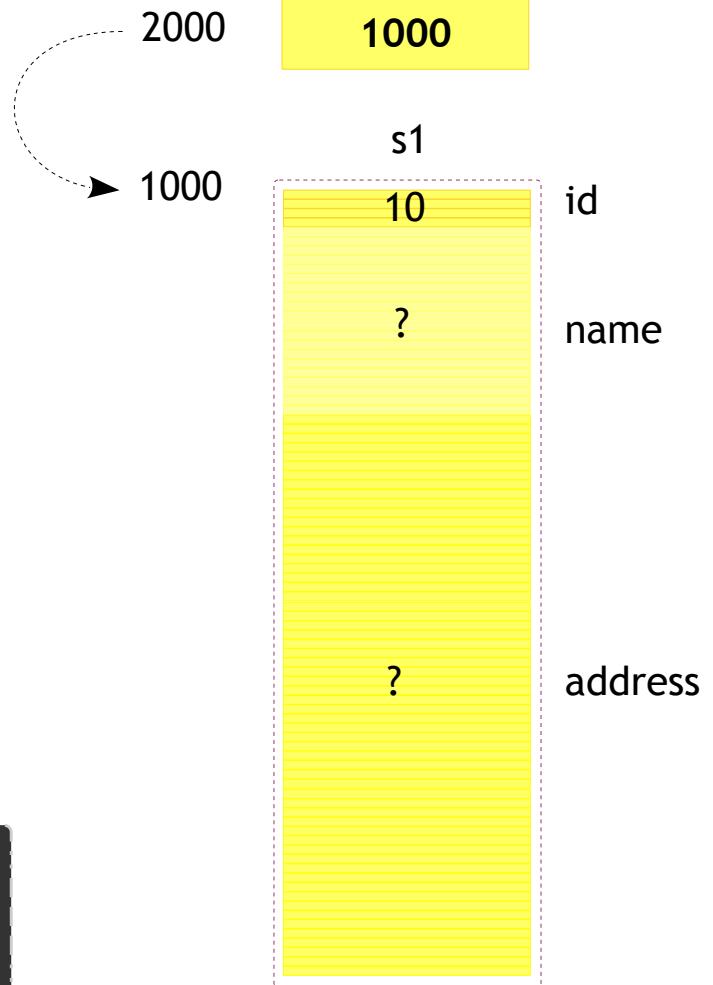
struct Student
{
    int id;
    char name[20];
    char address[60];
};

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 - Array

013_example.c

```
struct Student
{
    int id;
    char name[20];
    char address[60];
};

int main()
{
    struct Student s[5];
    int i;

    for (i = 0; i < 5; i++)
    {
        scanf("%d", &s[i].id);
    }

    for (i = 0; i < 5; i++)
    {
        printf("s[%d].id is %d\n", i, s[i].id);
    }

    return 0;
}
```

Advanced C

UDTs - Structures - Nesting

014_example.c

```
struct College
{
    struct Student
    {
        int id;
        char name[20];
        char address[60];
    } student;
    struct
    {
        int id;
        char name[20];
        char address[60];
    } faculty;
};

int main()
{
    struct College member;

    member.student.id = 10;
    member.faculty.id = 20;

    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

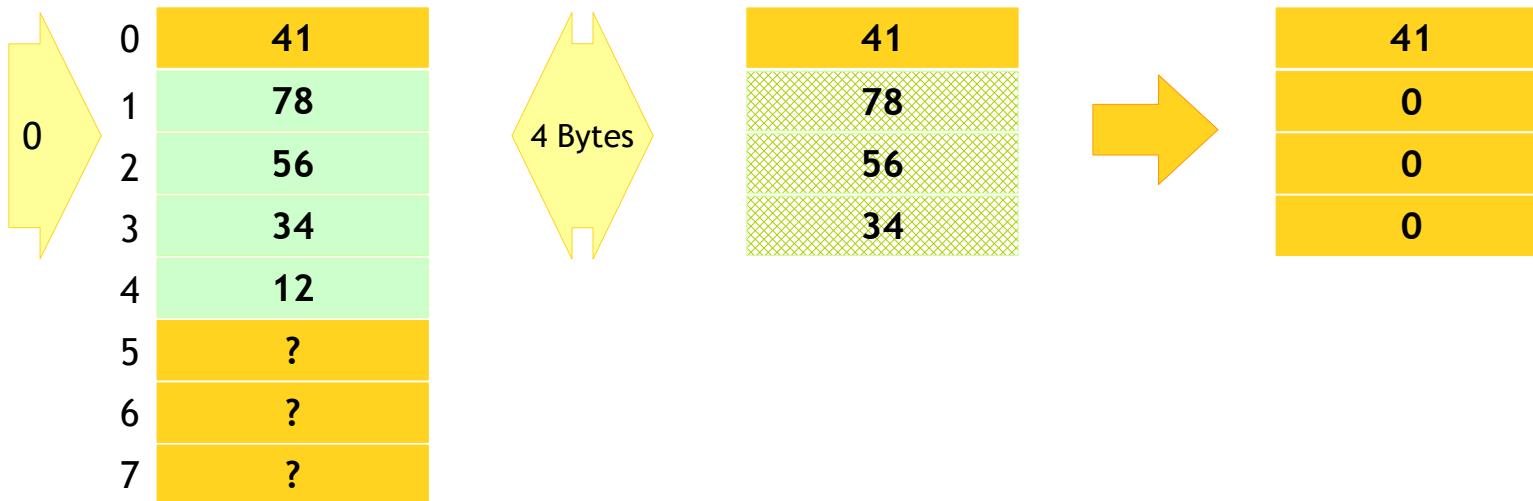
Advanced C

Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching a character by the CPU will be like shown below



Advanced C

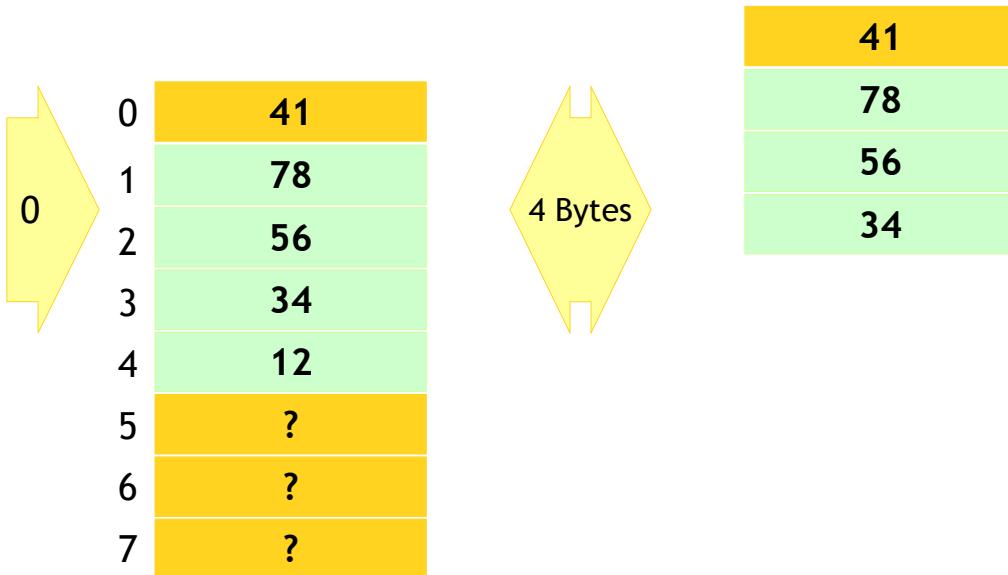
Data Alignment



Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching integer by the CPU will be like shown below



Advanced C

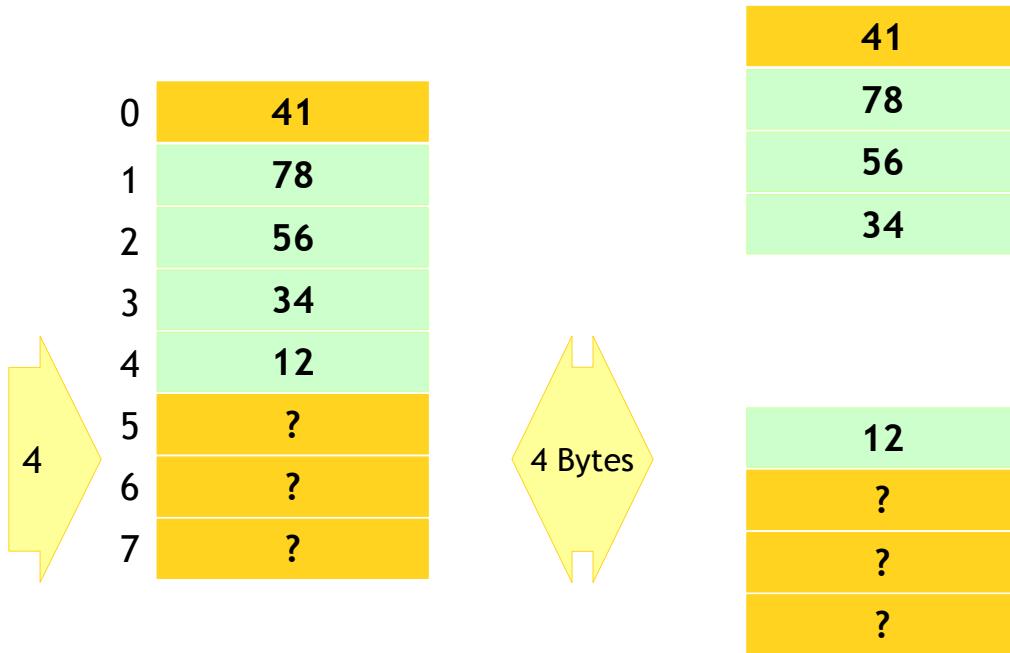
Data Alignment



Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



Advanced C

Data Alignment



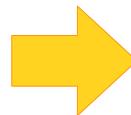
Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below

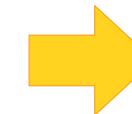
0	41
1	78
2	56
3	34
4	12
5	?
6	?
7	?

41
78
56
34
12
?
?
?



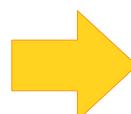
Shift 1 Byte Up

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

015_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

016_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

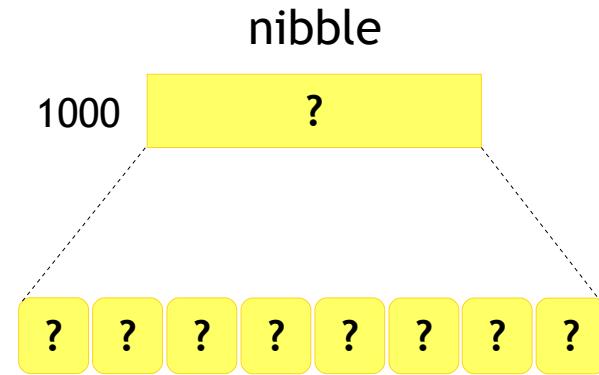
017_example.c

```
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



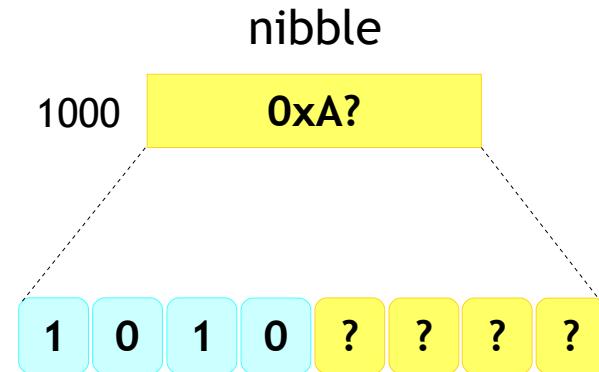
017_example.c

```
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



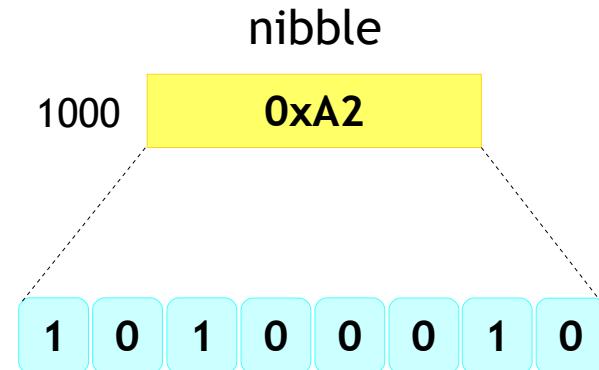
017_example.c

```
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

018_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



019_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

019_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

020_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



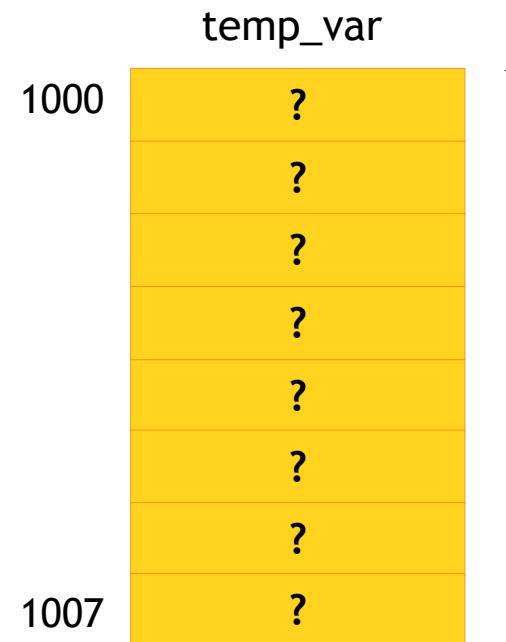
021_example.c

```
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



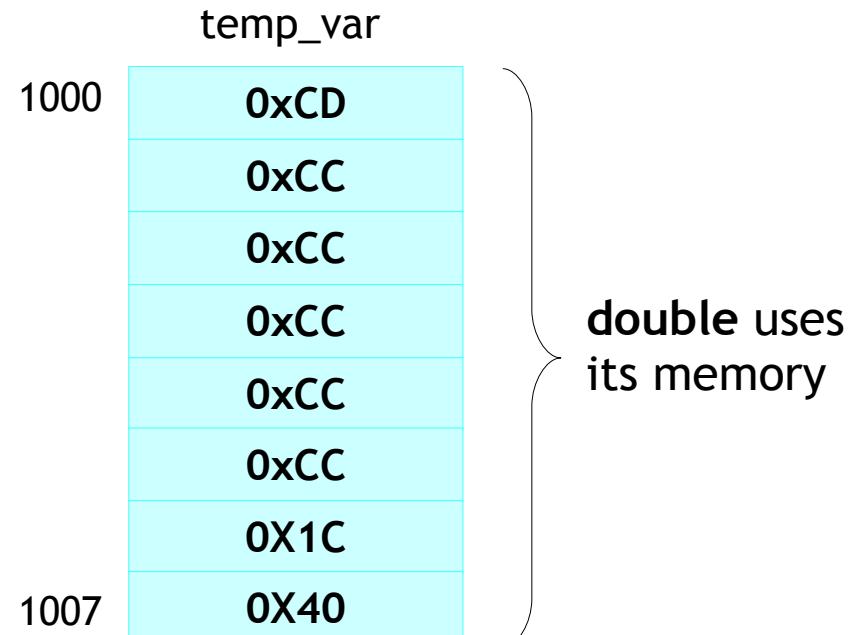
021_example.c

```
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



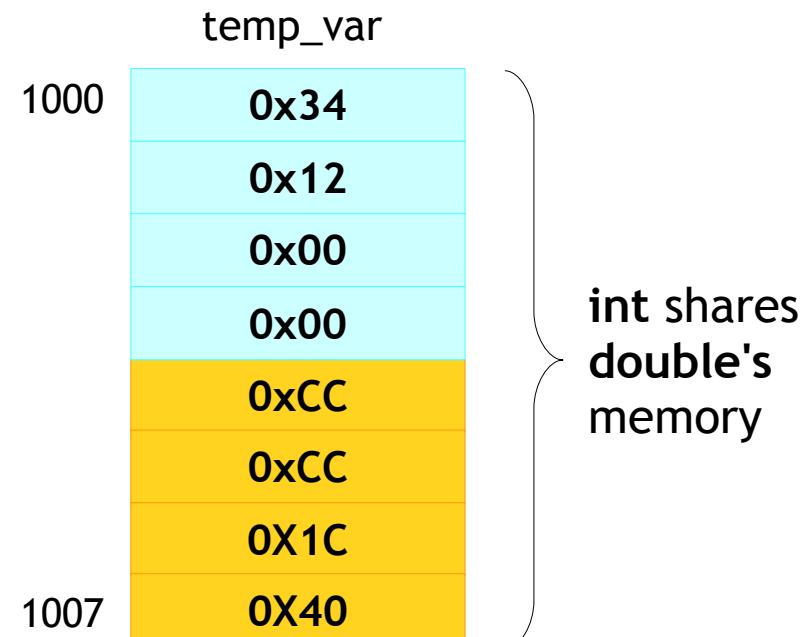
021_example.c

```
union 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



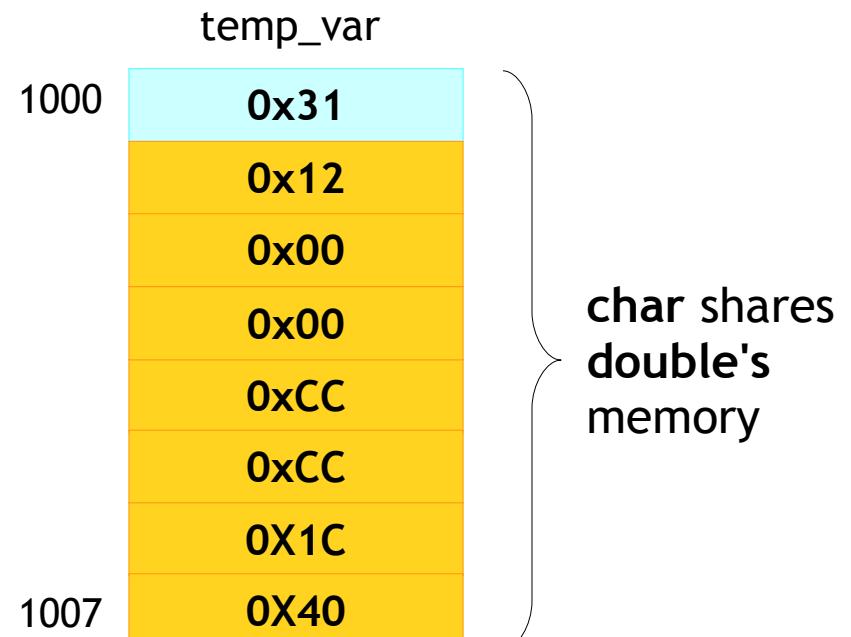
021_example.c

```
union 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



022_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



023_example.c

```
union Endian
{
    unsigned int value;
    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

025_example.c

```
typedef unsigned int uint;

int main()
{
    uint number;

    return 0;
}
```

027_example.c

```
typedef int array_of_100[100];

int main()
{
    array_of_100 array;

    printf("%zu\n", sizeof(array));

    return 0;
}
```

026_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

028_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;
}
```

029_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

030_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



031_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



032_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



033_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



034_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



035_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



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 *path, const char *mode);  
int fclose(FILE *stream);
```

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 “abcdabcbcdabc”.
 - WAP program to find the occurrences of character 'c' using **fseek()**



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, num3, num4;
    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(&num3, sizeof(num3), 1, fptr);
    fread(&num4, sizeof(num4), 1, fptr);

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

    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

Miscellaneous



Advanced C

Miscellaneous - Volatile

- A datatype qualifier
- Most commonly used Embedded Applications
- A keyword instructing the compiler not apply any optimizations on objects qualified with it!
- Why??
 - You know! the compiler sometimes act extra smart on the objects not qualified with volatile
 - Takes implicit assumption the the objects

Advanced C

Miscellaneous - Volatile

001_example.c

```
#include <stdio.h>

int main()
{
    long int wait;
    unsigned char bit = 0;

    while (1)
    {
        bit = !bit;
        printf("The bit is now: %d\r", bit);
        fflush(stdout);
        for (wait = 0xFFFFFFF; wait--; );
    }

    return 0;
}
```

Compile like

```
user@user:~] gcc 001_example.c
```

- Typical embedded bit toggle code
- The output would toggle between 0 and 1 (Depends on the system configuration, tune the delay as required)
- Note the toggle frequency and

Compile like

```
user@user:~] gcc -O3 001_example.c
```

- Now try the same code

Advanced C

Miscellaneous - Volatile

001_example.c

```
#include <stdio.h>

int main()
{
    volatile long int wait;
    unsigned char bit = 0;

    while (1)
    {
        bit = !bit;
        printf("The bit is now: %d\r", bit);
        fflush(stdout);
        for (wait = 0xFFFFFFFF; wait--; );
    }

    return 0;
}
```

Compile like

```
user@user:~] gcc 001_example.c
```

- or

Compile like

```
user@user:~] gcc -O3 001_example.c
```

- Should solve the issue!

- What happens in the previous code is that the compiler see the for loop as an unnecessary code just delaying the system operation
- Hence it removes that statement in the final output
- Adding volatile to the wait restricts the compiler from optimizing the code

Advanced C

Miscellaneous - Volatile

002_example.c

```
#include <stdio.h>

int main()
{
    unsigned int i;
    int num;

    for (i = 0; i < 0xFFFFFFFF; i++)
    {
        num = 5;
    }

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

    return 0;
}
```

Compile like

```
user@user:~] gcc 002_example.c
```

- Might take some time to see the output on screen!!

Compile like

```
user@user:~] gcc -O3 002_example.c
```

- Immediate output!!
- Compiler sees the same assignment operation is unnecessarily happening in the loop
- Optimizes the loop, by removing the for loop

Advanced C

Miscellaneous - Volatile

002_example.c

```
#include <stdio.h>

int main()
{
    volatile unsigned int i;
    int num;

    for (i = 0; i < 0xFFFFFFFF; i++)
    {
        num = 5;
    }

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

    return 0;
}
```

Compile like

```
user@user:~] gcc 002_example.c
```

- or

Compile like

```
user@user:~] gcc -O3 002_example.c
```

- Should solve the issue!
- Both should behave the same way!

Advanced C

Miscellaneous - Volatile

003_example.c

```
#include <stdio.h>

int main()
{
    int get_out;
    int num = 0;

    scanf("%d", &get_out);

    while (get_out)
    {
        num++;
    }

    return 0;
}
```

Compile like

```
user@user:~] gcc 003_example.c
```

- Enter 1 and should not come out of the loop
- Terminate and run again with input 0, you should not enter the loop

Compile like

```
user@user:~] gcc -O3 003_example.c
```

- Enter 1 and should not enter the loop!! which shouldn't be the case

Advanced C

Miscellaneous - Volatile

003_example.c

```
#include <stdio.h>

int main()
{
    int get_out;
    volatile int num = 0;

    scanf("%d", &get_out);

    while (get_out)
    {
        num++;
    }

    return 0;
}
```

Compile like

```
user@user:~] gcc 003_example.c
```

- or

Compile like

```
user@user:~] gcc -O3 003_example.c
```

- Should solve the issue!
- Both should behave the same way!

Advanced C

Miscellaneous - Volatile

004_example.c

```
#include <stdio.h>

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

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

    return 0;
}
```

Compile like

```
user@user:~] gcc 004_example.c
```

- Use the precedence table to obtain the result and verify with output

Run

```
user@user:~] ./a.out
```

Advanced C

Miscellaneous - Volatile

004_example.c

```
#include <stdio.h>

int main()
{
    int num1;
    volatile int num2 = 1;

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

    return 0;
}
```

Compile like

```
user@user:~] gcc 004_example.c
```

- and

Run

```
user@user:~] ./a.out
```

- Hmmhh, is it ok now!!