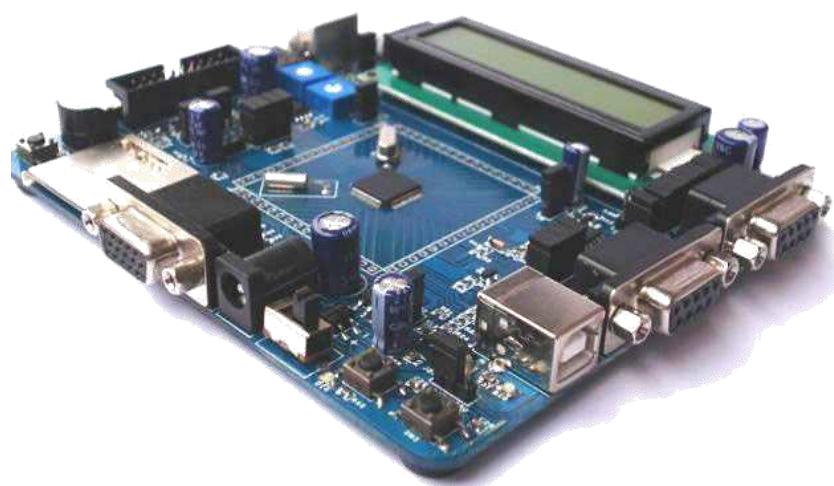




Embedded C Programming



Agenda

1. Hello C Programming
2. Control Flow
3. Preprocessors
4. Functions and Modular Programming
5. Special Data Types
6. Pointers, Arrays, and Strings
7. Data Structures
8. Algorithms
9. Code Portability

Agenda

10. Code Optimization
11. Standard C Library
12. Building Process
13. Guideline for Writing Good Code

1. Hello C Programming

Outline

- The Course consists of the following topics:
 - Types of Programming Languages
 - C Language Standards
 - Comments in C
 - Numeral Systems
 - Keywords and Identifiers
 - Data Types
 - Variables, Operators, and Casting
 - Statements and Blocks
 - Input/Output in C

Types of Programming Languages

- Low Level Language
 - Machine understandable language
 - Internal Machine Code dependent
 - Fast to Run, but slow to write and understand

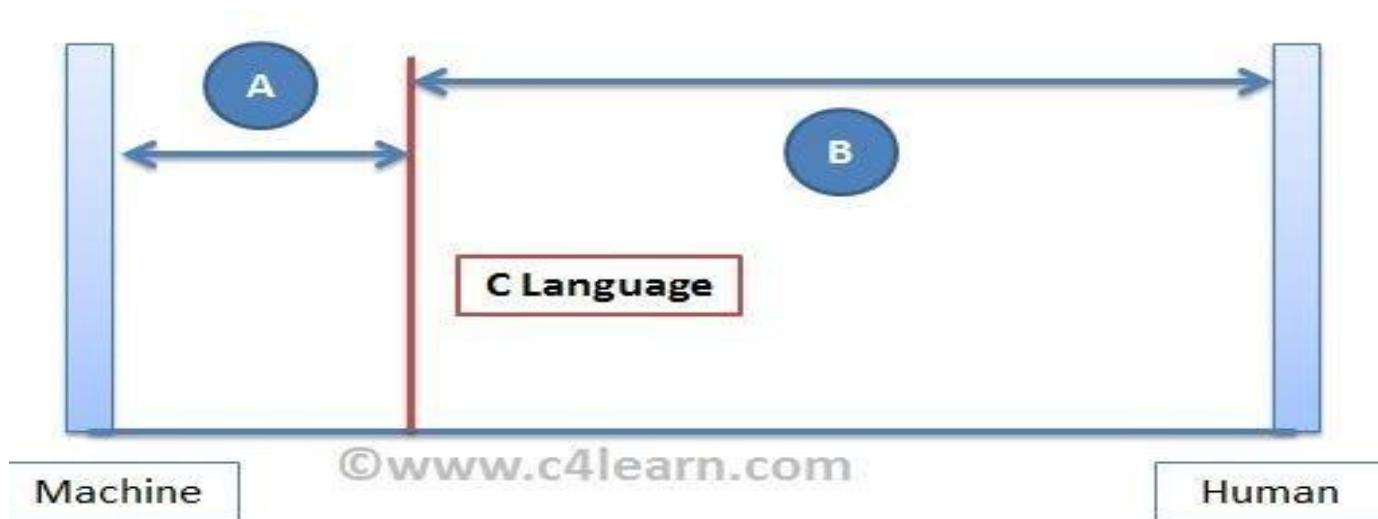
LOW-LEVEL LANGUAGES

00110101	01111010		
10110000	10100100		
11101111	11100101		
01010100	10001001	FIELDA	FIELDB
10001001	01011010	FIELDC	TALLY
	ADD	CPN	MAX
		GTR	LOOPA
		MVN	PARTNO
		BUN	LABELA

8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008BD0419 83FA0376 078BD98B
C84AEBF1 5BC3

Types of Programming Languages

- Middle Level Language



- More Close to Machine
- Far from Human
- We have to Write More Code to meet user requirement
- Easy to create Machine Level Code

Types of Programming Languages

- Why C is Middle Level Language
 - C Programming supports Inline Assembly language programs.
 - Using inline assembly feature in C, we can directly access system registers.
 - C programming is used to access memory directly using pointers.
 - C programming also supports high level language features.
 - It is more user friendly as compare to previous languages so C is middle level language.

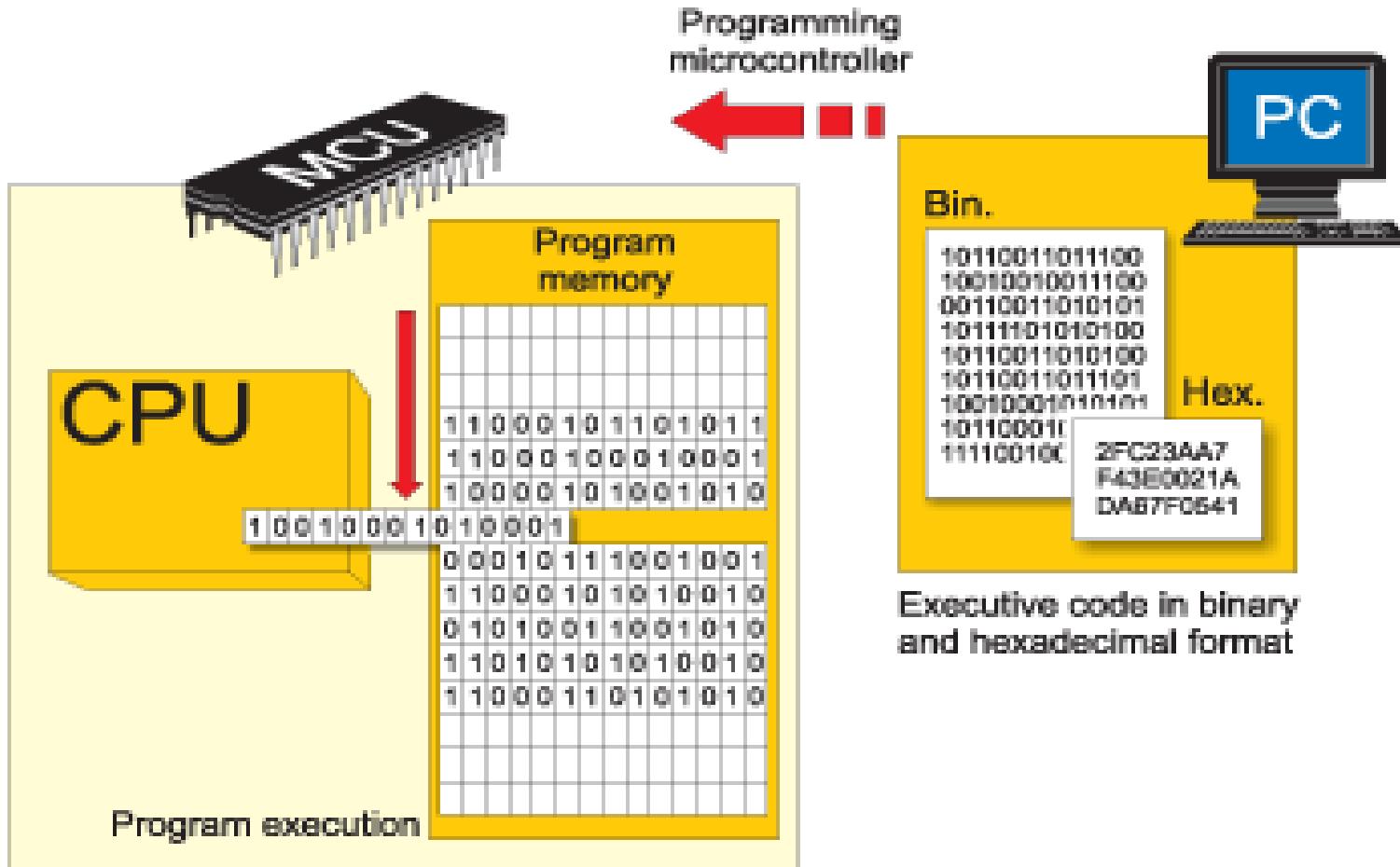
Types of Programming Languages

- High Level Language

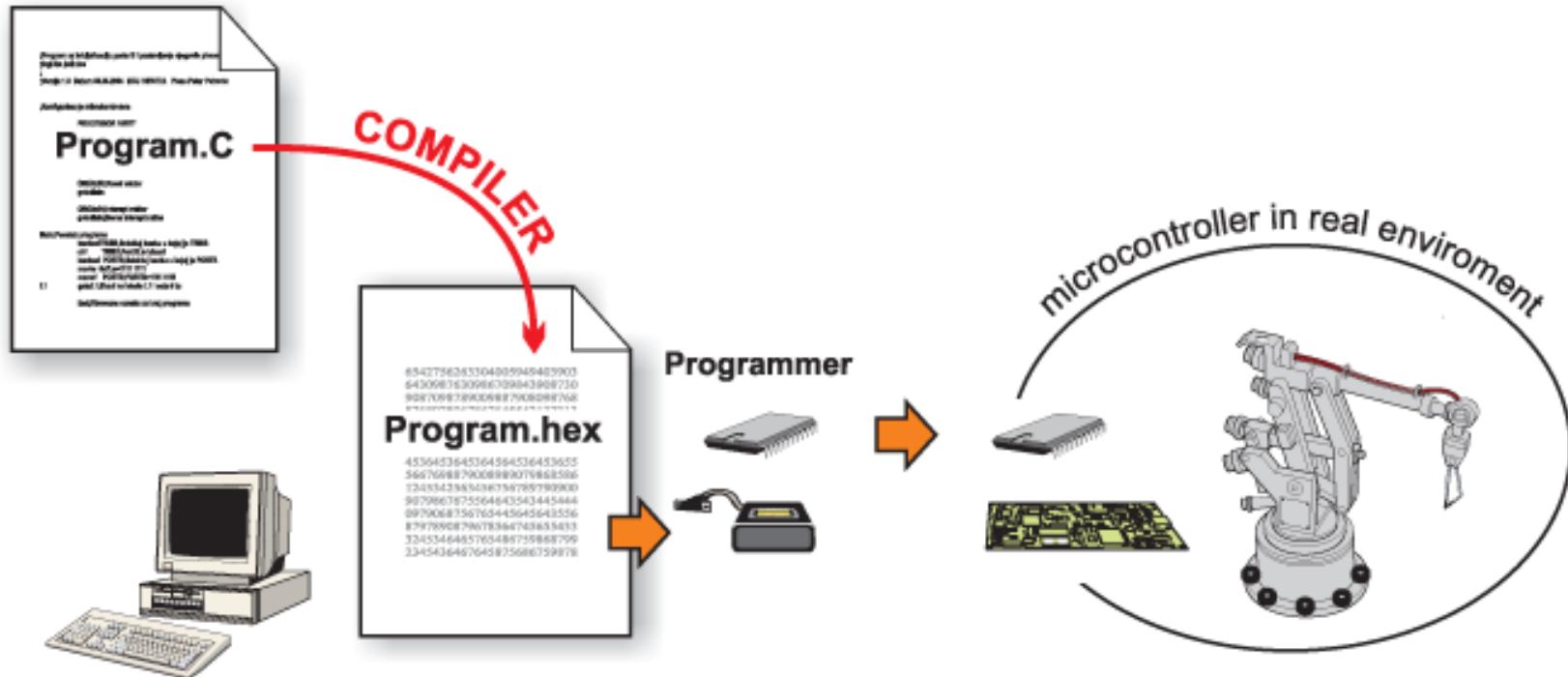


Types of Programming Languages

- Embedded systems



Types of Programming Languages



Types of Programming Languages

```
void main() {  
    TRISB = 0;                      // All port B pins are configured as  
    PORTB = 0b01010101;              // outputs  
}  
// Logic state on port B pins
```

Program written in C

ADDRESS	OPCODE	ASM
\$0000	\$2804	GOTO _main
\$0004	\$	_main:
:Test.c,1 ::	void main() {	
:Test.c,3 ::	TRISB = 0;	// All port B pins
\$0004	\$1303	BCF STATUS, RP1
\$0005	\$1683	BSF STATUS, RP0
\$0006	\$0186	CLRF TRIISB, 1
:Test.c,4 ::	PORTB = 0b01010101;	// Logic state
\$0007	\$3055	MOVlw 85
\$0008	\$1283	BCF STATUS, RP0
\$0009	\$0086	MOVWF PORTB
:Test.c,5 ::	}	
\$000A	\$280A	GOTO \$

Compiled Program

```
:100000000428FF3FFF3FFF3F03138316860155304F  
:10001000831286000A28FF3FFF3FFF3FFF3F5D  
:04400E00F22FFFFF8F  
:00000001FF
```

Executable Code of the program (HEX code)



History of C

- C language standards:
 - C89/C90 standard
 - First standardized specification for C language was developed by the American National Standards Institute in 1989.
 - C89 and C90 standards refer to the same programming language.
 - C99 standard
 - Next revision was published in 1999 that introduced new features like advanced data types and other changes.

History of C

- C11 and Embedded C language
 - C11 standard
 - Adds new features to C and library like type generic macros, anonymous structures, improved Unicode support, atomic operations, multi-threading, and bounds-checked functions.
 - It also makes some portions of the existing C99 library optional, and improves compatibility with C++.
 - Embedded C
 - Includes features not available in C like fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

Comments in C

- Single Line / Multi line

Multi-line Comment	Single-line Comments
Starts with /* and ends with */	Starts with //
All Words and Statements written between /* and */ are ignored	Statements after the symbol // upto the end of line are ignored
Comment ends when */ Occures	Comment Ends whenever ENTER is Pressed and New Line Starts
e.g /* This is Multiline Comment */	e.g // Single line Comment

```
#include<stdio.h>
void main()
{
printf("Hello");
/* Multi
   Line
Comment
*/
printf("By");
}
```

```
#include<stdio.h>
void main()
{
printf("Hello"); //Single Line Comment
printf("By");
}
```

Common Number Systems

System	Base	Symbols	Human Readable ?	Used in computers?
Decimal	10	0, 1, ... 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, ... 7	No	No
Hexa-decimal	16	0, 1, ... 9, A, B, ... F	No	No

Quantities/Counting

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7

Quantities/Counting

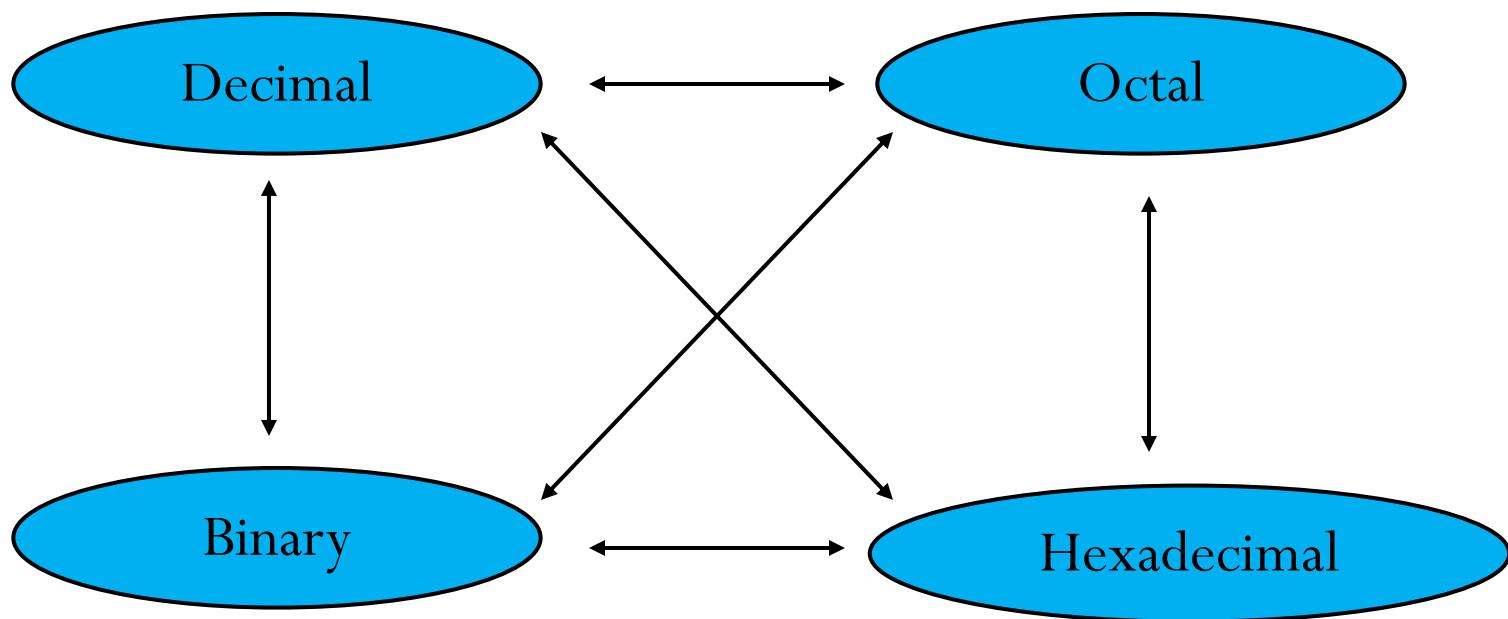
Decimal	Binary	Octal	Hexadecimal
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Quantities/Counting

Decimal	Binary	Octal	Hexadecimal
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14
21	10101	25	15
22	10110	26	16
23	10111	27	17

Conversion Among Bases

- The possibilities:



Quick Example

- $25d = 11001b = 31O = 19h$

Binary to Decimal Conversion

- Technique:
 - Multiply each bit by 2^n , where n is the “weight” of the bit
 - The weight is the position of the bit, starting from 0 on the right
 - Add the results

Octal to Decimal Conversion

- Technique :
 - Multiply each bit by 8^n , where n is the “weight” of the bit
 - The weight is the position of the bit, starting from 0 on the right
 - Add the results

Hexadecimal to Decimal Conversion

- Technique :
 - Multiply each bit by 16^n , where n is the “weight” of the bit
 - The weight is the position of the bit, starting from 0 on the right
 - Add the results

Binary to Hexadecimal Conversion

- Technique :
 - Group bits in fours, starting on right
 - Convert to hexadecimal digits

Hexadecimal to Octal

- Technique :
 - Use binary as an intermediary

Exercise – Convert ...

- Don't use a calculator!

Decimal	Binary	Octal	Hexadecimal
33			
	1110101		
		703	
			1AF

Exercise – Convert ...

- Answer

Decimal	Binary	Octal	Hexadecimal
33	100001	41	21
117	1110101	165	75
451	111000011	703	1C3
431	110101111	657	1AF

C Keywords and Identifiers

- Character set:
 - Character set are the set of alphabets, letters are valid in C language.
- Alphabets:
 - Uppercase: A B C XYZ
 - Lowercase: a b c x y z
- Digits: 0 1 2 3 4 5 6 8 9
- White space Characters: blank space, new line, horizontal tab, carriage return and form feed
- Special Characters:

,	<	>	.	_	()	;	\$:	%	[]	#	?
'	&	{	}	"	^	!	*	/		-	\	~	+	

C Keywords and Identifiers

- Keywords are the reserved words used in programming.
- Each keyword has fixed meaning and that cannot be changed by user.
- For example:

```
int money;
```

- Here, int is a keyword that indicates, 'money' is of type integer.
- C programming is case sensitive, all keywords must be written in lowercase.

C Keywords and Identifiers

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

Besides these keywords, there are some additional keywords supported by Turbo C.

Additional Keywords for Borland C

asm	far	interrupt	pascal	near	huge	cdecl
-----	-----	-----------	--------	------	------	-------

C Keywords and Identifiers

- Identifiers :
 - Identifiers are names given to C entities, such as variables, functions, structures

```
int money;  
int mango_tree;
```

C Keywords and Identifiers

- Rules for writing identifier :
 - Composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
 - The first letter of identifier should be either a letter or underscore.
 - There is no rule for the length of an identifier.
 - The first 31 letters of two identifiers in a program should be different.

What is Data Type?

- A data type is used to
 - Identify the type of a variable when the variable is declared
 - Identify the type of the return value of a function
 - Identify the type of a parameter expected by a function

C Programming Data Types

- Data types are the keywords, which are used for assigning a type to a variable.
- Fundamental Data Types:
 - Integer types
 - Floating Type
 - Character types
- Derived Data Types:
 - Arrays
 - Structures
 - Enumeration
 - Unions

C Programming Data Types

- Declaration of a variable

```
data_type variable_name;
```

- Integer data types : The size of int is either 2 bytes(In older PC's) or 4 bytes

```
int var1;
```

- Floating types

```
float var2;  
double var3;
```

```
float var3=22.442e2
```

C Programming Data Types

- Difference between float and double
 - Size of float(Single precision float data type) is 4 bytes and that of bytes. double(Double precision float data type) is 8 bytes
 - Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.
- Note: Precision describes the number of significant decimal places that a floating values carries.

C Programming Data Types

- Character types
 - The size of char is 1 byte.
 - The character data type consists of ASCII characters
 - Each Character is given a specific value

```
char var4='h';
```

```
For, 'a', value =97
For, 'b', value=98
For, 'A', value=65
For, '&', value=33
For, '2', value=49
```

C Programming Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

C Programming Data Types

- Constant qualifiers :
 - Constant qualifiers can be declared with keyword const.
 - An object declared by const cannot be modified.

```
const int p=20;
```

- Volatile qualifiers:
 - A variable should be declared volatile whenever its value can be changed by some external sources outside program.
 - Keyword volatile is used to indicate volatile variable.

Rules for Constructing Identifiers in C

- Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character
- First character must be a letter or underscore
- Usually only the first 32 characters are significant
- There can be no embedded blanks
- Keywords cannot be used as identifiers
- Identifiers are case sensitive
- Identifiers refer to the names of data types, constants, variables, and functions

Examples For Identifier Construction

- Use allowed Characters
 - Valid Names Examples:
 - num
 - Num
 - Num1
 - _NUM
 - NUM_temp2
- Blanks are not allowed
 - Invalid Names Examples:
 - number 1
 - num 1

Examples For Identifier Construction

- No special symbols other than underscore
 - Valid Identifier Examples:
 - num_1
 - number_of_values
 - status_flag
 - Invalid Identifier Example:
 - num@1

Examples For Identifier Construction

- First Character must be underscore or Alphabet

- Valid Identifier Examples:

- _num1
 - Num
 - Num_
 - _
 - __

- Invalid Identifier Examples:

- 1num
 - 1_num
 - 365_days

Examples For Identifier Construction

- Reserve words are not allowed
 - C is case sensitive.
 - Variable name should not be Reserved word.
 - However if we capitalize any Letter from Reserve word then it will become legal variable name.
 - Valid Identifier Examples:
 - Int
 - Char
 - Continue
 - CONTINUE
 - Invalid Identifier Examples:
 - int
 - char
 - continue

Examples For Identifier Construction

- Basic Note :

- Global predefined macro starts with underscore. (_) .
- We cannot use Global predefined macro as our function name.
- Example : Some of the predefined macros in C

- __TIME__
 - __DATE__
 - __FILE__
 - __LINE__

- Valid Identifier

- __NAME__
 - __SUM__

- Invalid Identifier

- __TIME__
 - __DATE__
 - __FILE__

Variables

- A variable is a named link/reference to a value stored in the system's memory or an expression that can be evaluated.

Variable Name

- Variable names can contain letters, digits and _.
- Variable names should start with letters.
- Keywords (e.g., for, while, etc.) cannot be used as variable names.
- Variable names are case sensitive.
- int x; and int X declares two different variables.

Variable Examples

- *int money\$owed;* (*incorrect: cannot contain \$*)
- *int total_count* (*correct*)
- *int score2* (*correct*)
- *int 2ndscore* (*incorrect: must start with a letter*)
- *int long* (*incorrect: cannot use keyword*)

Variable Declaration

- The general format for a declaration is

<type> <variable-name> = <initial value>;

- Examples:

- *char x; /* uninitialized */*
- *char x='A'; /* initialized to 'A' */*
- *char x='A', y='B'; /* multiple variables initialized */*

Constants

- Constants are literal/fixed values assigned to variables or used directly in expressions.
- Examples:
 - *int i=3;* *Decimal value*
 - *int i=0xA;* *Hexadecimal value*
 - *int i=012;* *Octal value*
 - *unsigned long x= 3UL;* *unsigned long value*
 - *float pi=3.141F ;* *float value*
 - *char x = 'A';* *x = the ascii of A (65)*
 - *string x = "HelloWorld";* *string value*

No string in C !!

C Operators

- Arithmetic Operations

Operator	Meaning	Example
+	Addition Operator	$10 + 20 = 30$
-	Subtraction Operator	$20 - 10 = 10$
*	Multiplication Operator	$20 * 10 = 200$
/	Division Operator	$20 / 10 = 2$
%	Remainder (Modulo Operator)	$20 \% 6 = 2$

C Operators

- Increment and decrement operators

```
Let a=5 and b=10  
a++; //a becomes 6  
a--; //a becomes 5  
++a; //a becomes 6  
--a; //a becomes 5
```

- Prefix (like: `++var`), `++ var` will increment the value of var then return it but
- Postfix (like: `var++`), operator will return the value of operand first and then only increment it.

C Operators

- Difference between postfix and prefix ++, -- operators
- What is the output?

```
#include <stdio.h>
int main(){
    int c=2,d=2;
    printf("%d\n",c++);
    printf("%d",++c);
    return 0;
}
```

C Operators

- Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	$C /= A$ is equivalent to $C = C / A$

C Operators

- Assignment Operators

Operator	Description	Example
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<code><<=</code>	Left shift AND assignment operator	<code>C <<= 2</code> is same as <code>C = C << 2</code>
<code>>>=</code>	Right shift AND assignment operator	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	bitwise exclusive OR and assignment operator	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	bitwise inclusive OR and assignment operator	<code>C = 2</code> is same as <code>C = C 2</code>

C Operators

- Relational Operators

Operator	Description	Example
<code>==</code>	Checks if the values of two operands are equal or not, if yes then condition becomes true.	$(A == B)$ is not true.
<code>!=</code>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	$(A != B)$ is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$(A > B)$ is not true.
<code><</code>	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$(A < B)$ is true.
<code>>=</code>	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$(A >= B)$ is not true.
<code><=</code>	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$(A <= B)$ is true.

C Operators

- Logical Operators

Operator	Description	Example
<code>&&</code>	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	$(A \&\& B)$ is false.
<code> </code>	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	$(A \mid\mid B)$ is true.
<code>!</code>	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	$!(A \&\& B)$ is true.

C Operators

- Conditional Operator
 - Conditional operator takes three operands and consists of two symbols ? And :
 - Conditional operators are used for decision making

```
c = ( c > 0 ) ? 10 : -10;
```

C Operators

- Conditional Operators

```
#include <stdio.h>
int main(){
    char feb;
    int days;
    printf("Enter 1 if the year is leap year otherwise enter 0: ");
    scanf("%c",&feb);
    days=(feb=='l')?29:28;
    /*If test condition (feb=='l') is true, days will be equal to 29. */
    /*If test condition (feb=='l') is false, days will be equal to 28. */
    printf("Number of days in February = %d",days);
    return 0;
}
```

Output

```
Enter 1 if the year is leap year otherwise enter n: 1
Number of days in February = 29
```

C Operators

- Bitwise Operators:
 - A bitwise operator works on each bit of data. Bitwise operators are used in bit level programming, Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator.	(A B) will give 61 which is 0011 1101
^	Binary XOR Operator.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator.	(~A) will give -61 which is 1100 0011
<<	Binary Left Shift Operator	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator.	A >> 2 will give 15 which is 0000 1111

C Operators Example

```
#include <stdio.h>
main()
{
    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13;           /* 13 = 0000 1101 */
    int c = 0;
    c = a & b;      /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c );
    c = a | b;      /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c );
    c = a ^ b;      /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c );
    c = ~a;         /* -61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c );
    c = a << 2;    /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c );
    c = a >> 2;   /* 15 = 0000 1111 */
    printf("Line 6 - Value of c is %d\n", c );
}
```

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

C Operators

- Other Operators:
 - Comma operators
 - used to link related expressions together
 - sizeof operator
 - It is a unary operator which is used in finding the size required for a certain data type

```
int a, c=5, d;
```

C Operators

```
#include <stdio.h>

main()
{
    int a = 4;
    short b;
    double c;
    int* ptr;

    /* example of sizeof operator */
    printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
    printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
    printf("Line 3 - Size of variable c= %d\n", sizeof(c) );
}
```

- Result

```
value of a is 4
*ptr is 4.
Value of b is 30
Value of b is 20
```

C Operators

- Other operators such as
 - & (reference operator),
 - * (dereference operator) and
 - ->(member selection) operator

Statements and Blocks

- A simple statement ends with semicolon:

Z = x + y;

- Block contains one or more statements inside curly braces:

```
{  
    float temp2 = x*y;  
    z += temp2;  
}
```

Blocks

- Block can substitute for simple statement
- Compiled as a single unit
- Variables can be declared inside

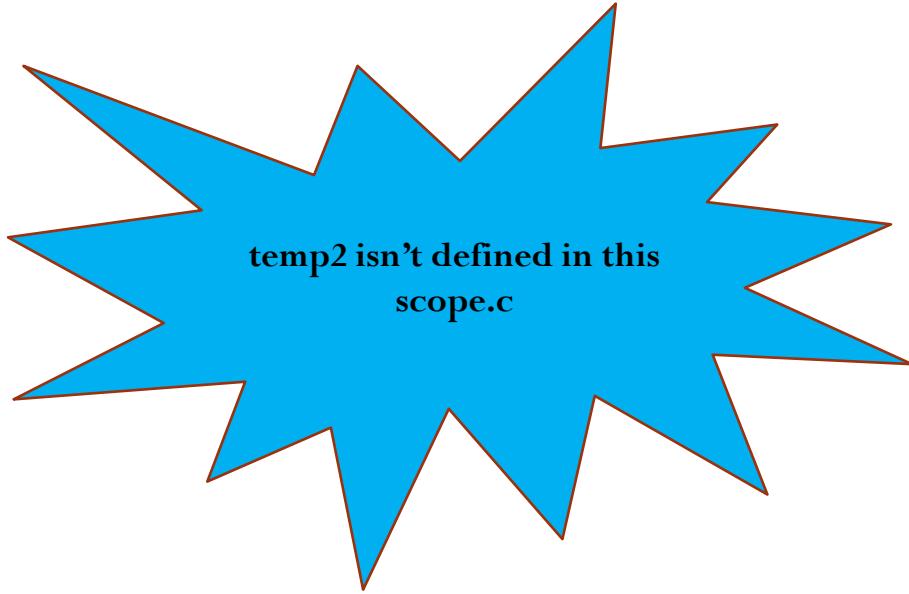
Nested Blocks

- Variables scope in nested blocks:
 - Variables are only defined within the block containing the declaration of it and all the nested blocks inside this block.

```
{  
    int x, y, z =10;  
    int temp = x+y;  
    {  
        float temp2 = x*y;  
        z += temp2;  
    }  
}
```

Nested Block

```
{  
    int x,y,z =10;  
    int temp = x+y;  
    {  
        float temp2 = x*y;  
        z += temp2;  
    }  
    z = temp2;  
}
```



temp2 isn't defined in this
scope.c

C Input/Output

- printf() and scanf()

```
#include <stdio.h>          //This is needed to run printf() function.  
int main()  
{  
    printf("C Programming"); //displays the content inside quotation  
    return 0;  
}
```

Output

```
C Programming
```

C Input/Output

- I/O of integers

```
#include<stdio.h>
int main()
{
    int c;
    printf("Enter a number\n");
    scanf("%d",&c);
    printf("Number=%d",c);
    return 0;
}
```

Output

```
Enter a number
4
Number=4
```

C Input/Output

- I/O of floats

```
#include <stdio.h>
int main(){
    float a;
    printf("Enter value: ");
    scanf("%f",&a);
    printf("Value=%f",a);      //%f is used for floats instead of %d
    return 0;
}
```

Output

```
Enter value: 23.45
Value=23.450000
```

C Input/Output

- I/O of characters and ASCII code

```
#include <stdio.h>
int main(){
    char var1;
    printf("Enter character: ");
    scanf("%c",&var1);
    printf("You entered %c.",var1);
    return 0;
}
```

Output

```
Enter character: g
You entered g.
```

C Input/Output

- I/O of ASCII code

```
#include <stdio.h>
int main(){
    char var1;
    printf("Enter character: ");
    scanf("%c",&var1);
    printf("You entered %c.\n",var1);
/* \n prints the next line(performs work of enter). */
    printf("ASCII value of %d",var1);
    return 0;
}
```

Output

```
Enter character:
g
103
```

C Input/Output

- I/O of ASCII code

```
#include <stdio.h>
int main(){
    int var1=69;
    printf("Character of ASCII value 69: %c",var1);
    return 0;
}
```

Output

```
Character of ASCII value 69: E
```

C Input/Output

- Validation in output for integers and floats

```
#include<stdio.h>
int main(){
    printf("Case 1:%6d\n",9876);
    /* Prints the number right justified within 6 columns */
    printf("Case 2:%3d\n",9876);
    /* Prints the number to be right justified to 3 columns but, there are 4 digits
       printf("Case 3:%.2f\n",987.6543);
    /* Prints the number rounded to two decimal places */
    printf("Case 4:%.f\n",987.6543);
    /* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
    printf("Case 5:%e\n",987.6543);
    /* Prints the number in exponential notation(scientific notation) */
    return 0;
}
```

Output

```
Case 1: 9876
Case 2:9876
Case 3:987.65
Case 4:988
Case 5:9.876543e+002
```

C Input/Output

- Validation in output for integers and floats

```
#include <stdio.h>
int main(){
    int a,b;
    float c,d;
    printf("Enter two intgers: ");
/*Two integers can be taken from user at once as below*/
    scanf("%d%d",&a,&b);
    printf("Enter intger and floating point numbers: ");
/*Integer and floating point number can be taken at once from user as below*/
    scanf("%d%f",&a,&c);
    return 0;
}
```

Lab #1

- Implement a program that takes 2 input numbers from the user, and prints the following:
 - First Number
 - Second Number
 - Addition Result
 - Subtraction Result
 - Multiplication Result
 - Division Result
 - Remainder Result
- Use proper displaying statement for each displayed value

2. Control Flow

Outline

- The Course consists of the following topics:
 - Conditional Statements
 - Conditional Expressions
 - IF Statement
 - SWITCH Statement
 - Loop Statements
 - FOR Statement
 - WHILE Statement
 - DO/WHILE Statement
 - Break Statement
 - Continue Statement
 - Debugging
 - Common Mistakes

The Conditional Expression

- C provides syntactic sugar to express the conditional expression using the ternary operator '?:'

```
sign=x>0?1:-1;  
if (x>0)  
    sign=1  
else  
    sign=-1
```

```
isodd=x%2==1?1:0;  
if (x%2==1)  
    isodd=1  
else  
    isodd=0
```

The “if” Statement

- if ($x \% 2 == 0$) $y += x / 2;$
 - Evaluate condition
 - If true, evaluate inner statement $y += x / 2;$
 - Otherwise, do nothing

The “else” Keyword

- Optional
- Execute statement if condition is false $y += (x+1)/2;$
- Either inner statement may be block

```
if (x%2 == 0)
    y += x / 2;
else
    y += (x + 1) / 2;
```

The “else if” Keyword

- Conditions evaluated in order until one is met; inner statement then executed
- If multiple conditions true, only first executed

```
if (x%2 == 0)
    y += x / 2;
else if (x%4 == 1)
    y+= 2* ((x+3)/4);
else
    y += (x + 1) / 2;
```

Nesting “if” Statements

- To which if statement does the else keyword belong?

```
if (x%4 == 0)
    if (x%2 == 0)
        y = 2;
else
    y = 1;
```

Nesting “if” Statements

- To associate else with outer if statement: use braces

```
if (x%4 == 0) {  
    if (x%2 == 0)  
        y = 2;  
    }  
else  
    y = 1;
```

The “switch” Statement

- Alternative conditional statement
- Integer (or character) variable as input
- Considers cases for value of variable

```
switch (ch) {  
    case 'Y': /* ch == 'Y' */  
        /* do something */  
        break;  
    case 'N': /* ch == 'N' */  
        /* do something else */  
        break;  
    default: /* otherwise */  
        /* do a third thing */  
        break;  
}
```

The “switch” Statement

- Compares variable to each case in order
- When match found, starts executing inner code until break; reached
- Execution “falls through” if break; not included

The “switch” Statement

```
switch (ch) {  
    case 'Y':  
    case 'y':  
        /* do something if  
           ch == 'Y' or  
           ch == 'y' */  
    break;  
}
```

```
switch (ch) {  
    case 'Y':  
        /* do something if  
           ch == 'Y' */  
    case 'N':  
        /* do something if  
           ch == 'Y' or  
           ch == 'N' */  
    break;  
}
```

Lab #1

- Implement a program that takes the grade of a student ranging from 0 to 100; and prints the grade as Excellent, Very Good, Good, Pass, Fail
- Use “if” statements

Lab #2

- Implement a program that takes the grade of a student ranging from 0 to 100; and prints the grade as Excellent, Very Good, Good, Pass, Fail
- Use “switch” statements

“for” Statement

- Best practice is to use it for known and precise number of iterations; e.g. “counting” loops
- Inside parentheses, three expressions, separated by semicolons:
 - Initialization: $i=1$
 - Condition: $i \leq n$
 - Increment: $i++$
- Expressions can be empty (condition assumed to be “true”)

“for” Statement

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

“for” Statement

- Compound expressions separated by commas

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

- Comma: operator with lowest precedence, evaluated left-to-right

“while” Statement

- Best practice is to Use it for unknown number of iterations.
 - while (/ * condition * /) {
 - / * loop body * /}
- Simplest loop structure – evaluate body as long as condition is true
- Condition evaluated first, so body may never be executed

“while” Statement

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }
    return 0;
}
```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

“do-while” Statement

- Differs from while loop – condition evaluated after each iteration
- Body executed at least once
- Note : Semicolon after the Condition

```
char c;
do {
    /* loop body */
    puts("Keep going? (y/n) ");
    c = getchar();
    /* other processing */
} while (c == 'y' && /* other conditions */ );
```

Loop Condition

- Remember that you have to guarantee that the loop condition should (at some certain time) be false, so that the loop doesn't get in infinite loop unless you want an infinite loop in your application.

“break” Keyword

- Sometimes want to terminate a loop early
- `break;` exits innermost loop or switch statement to exit early

```
char c;
do {
    /* loop body */
    puts("Keep going? (y/n) ");
    c = getchar();
    if (c != 'y')
        break;
    /* other processing */
} while /* other conditions */;
```

“continue” Keyword

- Used to skip an iteration; skips rest of innermost loop body, jumping to loop condition

```
int gcd(int a, int b) {  
    int i, ret = 1, minval = min(a,b);  
    for (i = 2; i <= minval; i++) {  
        if (a % i) /* i not divisor of a */  
            continue;  
        if (b % i == 0) /* i is divisor of both a and b */  
            ret = i;  
    }  
    return ret;  
}
```

Debugging

- A good technique for "debugging" code is to think of yourself in place of the computer.
- Go through all the loops in the program and ask "what is in each variable?"
- Each time you go through a loop ask "is the condition met" and "should I continue"
- The factorial program shows how to do this.

Debugging

```
int main()
{
    int number= 4;
    int answer;
    int count;

    answer= 1;
    count= number;
    while (count >= 0) {
        answer= answer* count;
        count--;
    }
    printf ("%d! = %d\n",
            number, answer);
return 0;
}
```

number= 4
answer= 1
count= 4
enter while loop
answer= $1 \times 4 = 4$
count=3
enter while loop
answer= $4 \times 3 = 12$
count=2
enter while loop
answer= $12 \times 2 = 24$
count= 1
enter while loop
answer= $24 \times 1 = 24$
count= 0
enter while loop
answer= $24 \times 0 = 0$
AHA - I see!!!

Other Techniques for Debugging

- Check missing brackets and commas.
- Check that you have a semicolon at the end of every line which needs one.
- Put in some printf()s – if you know what your program is DOING you will know what it is DOING WRONG.
- Try to explain to someone else what the program is meant to do.
- Take a break, get a cup of coffee and come back to it fresh.
(Debugging is FRUSTRATING).

Lab #3

- Run the following program and see if the result is as expected and if not – debug until it works properly

```
#include <stdio.h>

/* Print the sum of the integers from 1 to 1000 */
int
main(int argc, char **argv)
{
    int i;
    int sum;

    sum = 0;
    for(i = 0; i -= 1000; i++) {
        sum += i;
    }
    printf("%d\n", sum);
    return 0;
}
```

Common Mistakes

- Most of the following mistakes will not often rise compilation error, but will introduce a bug, meaning that the program will not provide the expected behavior. These bugs are very hard to be detected by beginner C students, however, they're very common.

Common Mistakes

```
int x = 6;  
if(x == 6)  
    printf("True");  
else  
    printf("False");
```

Common Mistakes

```
int x = 0;  
if(x == 0)  
    printf("True");  
else  
    printf("False");
```

Common Mistakes

```
int x = 0;  
while (x < 10);  
{  
    printf("%d\n", x);  
    x++;  
}
```

Common Mistakes

```
int x = 0;  
for (x=0,x<10,x++)  
{  
    //loop statements  
    //this mistake will rise compilation error  
}
```

Common Mistakes

int x = 0;

while (x < 10)

printf(“%d\n”,x);

x++;

Common Mistakes

Char x = ‘C Programming’;

Common Mistakes

```
char x [] = “C Programming”;  
x = “Java Programming”;  
//this error will cause compilation error.
```

Common Mistakes

```
int x;
```

```
scanf("%d",x);
```

Common Mistakes

```
char name[25];  
scanf("%s", &name);
```

Common Mistakes

```
char name[25];  
printf("%s", &name);
```

Common Mistakes

```
int x = 6;
```

```
printf("%d", &x);
```

Lab #4

- Implement a program that takes an input value from the user, and prints the summation result of the odd the numbers, from 0 to the given input value
- Use “for” statement

Lab #5

- Implement a program that takes an input value from the user, and prints the factorial value
- Use “while” statement

3. Preprocessor

Outline

- The Course consists of the following topics:
 - Preprocessors
 - Macros
 - File Inclusion
 - Preprocessor Constants
 - Commonly Used Macros

Preprocessors

- Pre-processor commands start with “#” which may optionally be surrounded by spaces and tabs
- The preprocessor allows us to :
 - Include files
 - Define, test and compare constants
 - Write macros
 - Debug

Definitions

- It is a better way to write the literal constants, as it increase the code readability and maintainability.
 - `#define Phi 3.14`
 - `#define Num_Of_Emp 10`
 - `#define Emp_Salary 100`
 - `#define Total_Salary Num_Of_Emp* Emp_Salary`
- These definitions are simply handled as a text replacement operation done by the preprocessor before the compilation of the file.

Macros

- Text replacement for expressions
- `#define MAX(a,b) (a)>(b)?(a):(b)`
- No semicolon at the end.
- No space after the macro name and before left parentheses.
- If more than one line is needed, mark each line end with a \.
- Dummy text replacement.
- Can be used in Wrapping functions.
- Reduce context-switching time but increases code size, but functions, on the other hand, waste some time in context-switch, but reduces code size.
- Both (macros and functions) provide single point of code change.

Macros

- Parameterized Macros
 - Macro that is able to insert given objects into its expansion. This gives the macro some of the power of a function.

```
#define add(x, y) ((x)+(y))
```

- Example:

```
main()
{
    int i=5, j=7, result=0;
    result= add(i, j);
    printf("result = %d", result);
}
```

Macros

- Using Macros to Define Used Constants to improve readability and maintainability.

```
#define MAX_STUDENTS_COUNT 20  
#define PI 3.14F  
#define INTER_PLATFORM_MESSAGE_1 0xAA
```

- Example:

```
main()  
{  
    int Students_List [ MAX_STUDENTS_COUNT];  
}
```

Macros

- Writing Multiline Macro

```
#define MULT(x, y) \  
int i = 0; \  
int j = 0; \  
i=x; \  
j=y; \  
x=i*j
```

Macros

- Use Macros to replace tiny functions or a bulk of code that are called or used so many times like inside a loop to save calling time and improve performance.
- Always surround all symbols inside your macro with parenthesis to avoid any future mistakes due to Using this macro.

Macros

- Example:

```
#define MULT(x,y) x*y
```

```
int i=2,j=3;
```

```
result= MULT(i+5, j);
```

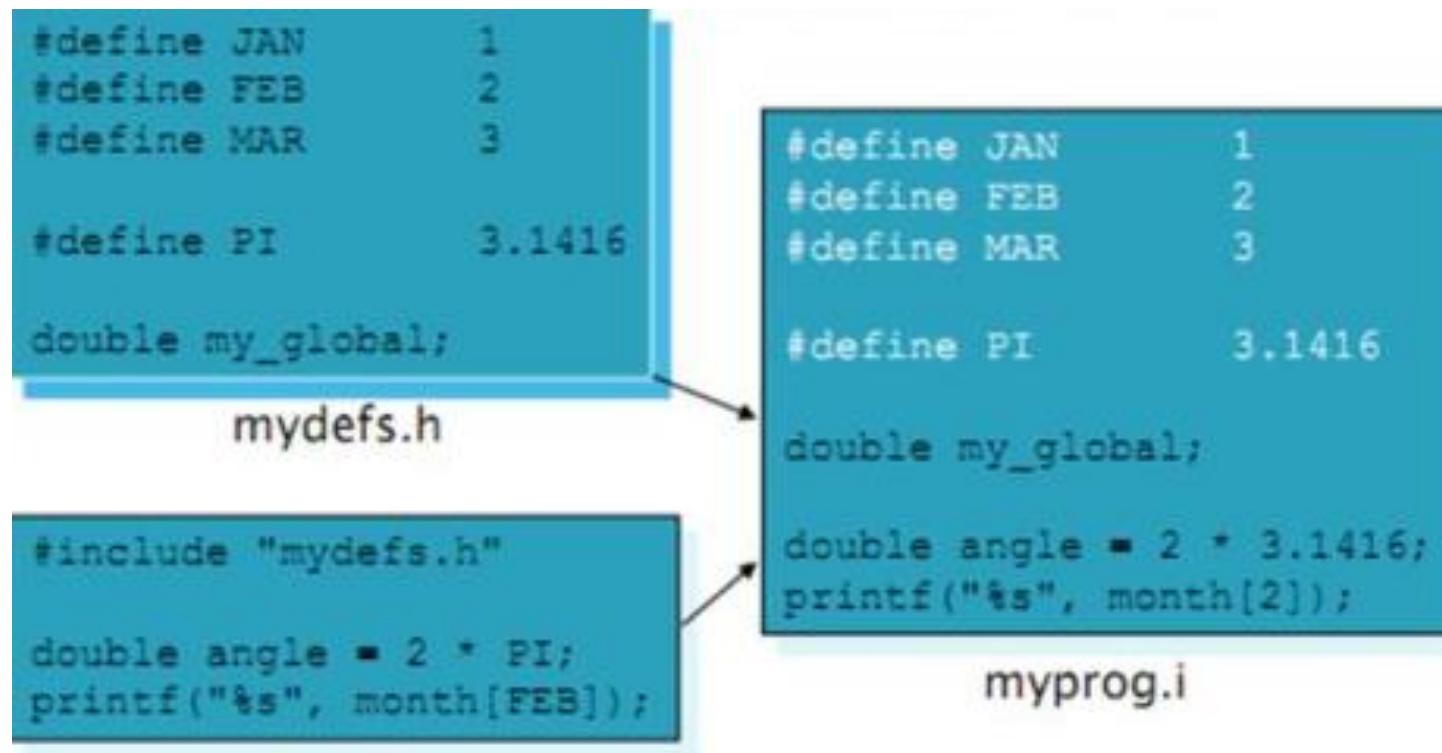
- You expect result= 21
- Actual result result= 17

Lab #1

- Implement a program that takes 2 input numbers i.e. x and y; and do the following:
 - Check value of given x is less than a predefined value
 - Check value of given y is less than a predefined value
 - Print the result of $x + y$
 - Print the result of $x * y$
 - Print the result of $(x + y) * x$
- Use Macros to hold the predefined values of x and y
- Use Macro to hold addition formula
- Use Macro to hold multiplication formula

File Inclusion

- The #include directive causes the pre-processor to "edit in" the entire contents of another file



File Inclusion

- Pathnames:
 - Full pathnames may be used, although this is not recommended

```
#include "C:\cct\course\cprog\misc\slideprog\header.h"
```

Preprocessor Constants

- Preprocessor Constants :
 - Constants may be created, tested and removed

```
#if !defined(SUN)
#define SUN 0
#endif
```

if "SUN" is not defined, then begin
 define "SUN" as zero
end

```
#if SUN == MON
#undef SUN
#endif
```

if "SUN" and "MON" are equal, then begin
 remove definition of "SUN"
end

```
#if TUE
```

if "TUE" is defined with a non zero value

```
#if WED > 0 || SUN < 3
```

if "WED" is greater than zero or "SUN" is less than 3

```
#if SUN > SAT && SUN > MON
```

if "SUN" is greater than "SAT" and "SUN" is greater than "MON"

Preprocessor Constants

- Debugging
 - Several extra features make the pre-processor an indispensable debugging tool

```
#define GOT_HERE printf("reached %i in %s\n", \
                      __LINE__, __FILE__)  
  
#define SHOW(E, FMT) printf(#E " = " FMT "\n", E)
```

```
printf("reached %i in %s\n", 17, "mysource.c");  
  
GOT_HERE;  
SHOW(i, "%x");  
SHOW(f/29.5, "%lf");  
  
printf("i = %x\n", i);  
  
printf("f/29.5 = %lf\n", f/29.5);
```

The #if statement

- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on a certain condition.
- The #if only deals with definition as it is evaluated before the compilation of the program

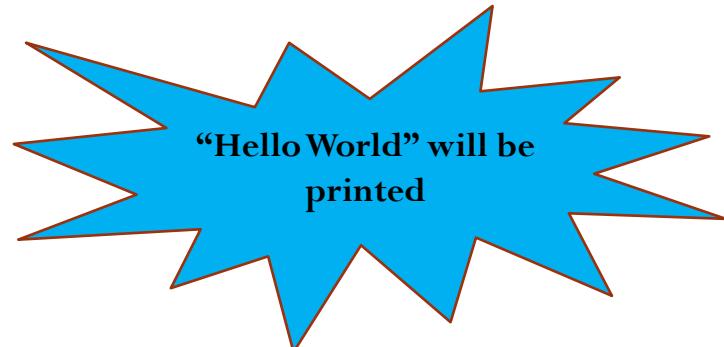
```
#define log 1
int main()
{
    int x =1;
    printf("%d\n",x);
    # if (log == 1)
    printf("Hello world!\n");
    # else
    printf("Hello Earth\n");
    #endif
    return 0;
}
```



The #ifdef statement

- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on the existence of a certain definition.

```
#define log
int main()
{
    int x =1;
    printf ("%d\n",x);
    # ifdef log
    printf ("Hello world!\n");
    # else
    printf ("Hello Earth\n");
    # endif
    return 0;
}
```



The #ifndef statement

- It is a preprocessor keyword.
- Normally it is used to add or delete some parts of code based on the inexistence of a certain definition.

```
#define log
int main()
{
    int x =1;
    printf("%d\n", x);
    # ifndef log
    printf("Hello world!\n");
    # else
    printf("Hello Earth\n");
    #endif
    return 0;
}
```



Lab #2

- Create DebugHeader.h file, with a predefined macro i.e. IS_DEBUG_MODE having a value defining whether the software runs in debug mode or not
- In the main file, include DebugHeader.h
- In the main function, use the preprocessor keyword #if to print whether the software running in debug mode or not

Lab #3

- Create DebugHeader.h file, with a predefined macro “DEBUG_MODE_ACTIVE”
- If DEBUG_MODE_ACTIVE is defined, this indicated that debug mode is running
- In the main file, include DebugHeader.h
- In the main function, use the preprocessor keyword #ifdef to print whether the software running in debug mode or not

Commonly Used Macros

- Set Bit in a byte :
 - $X = X | (1 << n)$
- Clear Bit in a byte :
 - $X = X \& (- (1 << n))$
- Toggle Bit in a byte :
 - $X = X ^ (1 << n)$

Commonly Used Macros

- Return max out of two numbers

```
#define MAX(A,B) ((A)> (B) ? (A) :(B))
```

- Return min out of two numbers

```
#define MIN(X,Y) ((X) < (Y) ? (X) :(Y))
```

4. Functions and Modular Programming

Outline

- The Course consists of the following topics:
 - Functions
 - Variable Scope and Lifetime
 - Static and Extern
 - Inline Functions
 - Callback
 - Modular Programming

Functions

- The function is one of the most basic things to understand in C programming.
- A function is a sub-unit of a program which performs a specific task.
- Functions take arguments (variables) and may return an argument.

Functions

```
#include <stdio.h>
int maximum (int, int); /* Prototype - see later in lecture */

int main(int argc, char*argv[])
{
    int i= 4;
    int j= 5;
    int k;
    k= maximum (i,j); /* Call maximum function */
    printf ("%d is the largest from %d and %d\n",k,i,j);
    printf ("%d is the largest from %d and %d\n",maximum(3,5), 3, 5);
    return 0;
}

int maximum (int a, int b)
/* Return the largest integer */
{
    if (a > b)
        return a; /* Return means "I am the result of the function"*/
    return b;      /* exit the function with this result */
}
```

Prototype the function

Call the function

Function header

The Function!

Functions Can Access Other Functions

- Once you have written a function, it can be accessed from other functions. We can therefore build more complex functions from simpler functions.

```
int max_of_three (int, int, int); /* Prototype*/
```

```
/* Main and rest of code is in here */
```

```
int max_of_three (int i1, int i2, int i3)  
/* returns the maximum of three integers */  
{  
    return (maximum (maximum(i1, i2), i3));  
}
```

Void Functions

- A function doesn't have to take or return arguments.
- We prototype such a function using void.

```
void print_hello (void);
```

Prototype (at top of file)

```
void print_hello (void)
```

Function takes and returns
void (no arguments)

```
/* this function prints hello */
```

```
{
```

Another prototype

```
void odd_or_even (int);
```

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

```
}
```

Void Functions

```
void odd_or_even (int num)
/* this function prints odd or even appropriately */
{
    if ((num % 2) == 0) {
        printf("Even\n");
        return;
    }
    printf("Odd\n");
}
```



Function which takes one
int arguments and returns none

Notes About Functions

- A function can take any number of arguments mixed in any way.
- A function can return at most one argument.
- When we return from a function, the values of the argument HAVE NOT CHANGED.
- We can declare variables within a function just like we can within main() - these variables will be deleted when we return from the function

What are these prototype things?

- A prototype tells your C program what to expect from a function what arguments it takes (if any) and what it returns (if any)
- #include finds the prototypes for library functions (e.g. printf)
- A function **MUST** return the variable type we say that it does in the prototype.

Lab #1

- Implement a program, that holds 2 functions for setting and clearing flags of a variable; in addition to a function that prints the variable's value
 - Create an 8-bit variable
 - Create void SetFlag(uint8 bit_number);
 - Create void ClearFlag(uint8 bit_number);
 - Create void PrintFlag(void);
- Create test cases to test the program behavior

Lab #2

- What will be the output of the program?

```
#include<stdio.h>
int i;
int fun();

int main()
{
    while(i)
    {
        fun();
        main();
    }
    printf("Hello\n");
    return 0;
}
int fun()
{
    printf("Hi");
}
```

Variable Scope and Life Time

- Variable Scope:
 - The region in which a variable is valid
 - Variables declared outside of a function have global scope
- Variable Redefinitions:
 - You could re-declare any variable in your outer scope, but you couldn't declare 2 variables with the same name in the same scope.

Variable Scope and Life Time

```
int nmax = 20;

/* The main() function */
int main(int argc, char ** argv) /* entry point */
{
    int a = 0, b = 1, c, n;
    printf("%3d: %d\n", 1, a);
    printf("%3d: %d\n", 2, b);
    for (n = 3; n <= nmax; n++) {
        c = a + b; a = b; b = c;
        printf("%3d: %d\n", n, c);
    }
    return 0; /* success */
}
```

Variable Scope and Life Time

```
int nmax = 20;
int main ( int argc , char argv)
{
    int a=0, b=1, c, n, nmax =25;
    printf ( "%3d: %d\n" ,1 ,a );
    printf ( "%3d: %d\n" ,2 ,b );
    {
        int nmax = 10;
        printf ( "%3d: %d\n" ,1 ,nmax );
    }
    for (n =3; n<= nmax; n++)
    {
        c=a+b; a=b; b=c;
        printf ( "%3d: %d\n" ,n, c );
    }
}
return 0;
}
```

Static Keyword

- “static” keyword has two meanings, depending on where the static variable is declared
- Outside a function, static variables/functions only visible within that file, not globally (cannot be extern’ed)
- Inside a function, static variables:
 - Still local to that function
 - Initialized only during program initialization
 - Do not get reinitialized with each function call .

Extern Keyword

- When a variable is defined, the compiler allocates memory for that variable and possibly also initializes its contents to some value.
- When a variable is declared, the compiler requires that the variable was defined elsewhere.
- The declaration informs the compiler that a variable by that name and type exists, but the compiler need not allocate memory for it since it is allocated elsewhere.
- The “extern” keyword means "declare without defining"

Inline Functions

- Inline function is a function upon which the programmer has requested that the compiler insert the complete body of the function in every place that the function is called, rather than generating code to call the function in the one place it is defined.
- Compilers are not obligated to respect this request; due to some constraints according to each compiler
- Example:

```
inline int max(int a, int b)
{
    return (a> b) ? a :b;
}
```

Inline Functions Vs. Macros

- Use of inline functions provides several benefits over macros
- Macro invocations do not perform type checking, or even check that arguments are well-formed, whereas function calls usually do.
- In C, a macro cannot use the return keyword with the same meaning as a function would do (it would make the function that asked the expansion terminate, rather than the macro). In other words, a macro cannot return anything which is not the result of the last expression invoked inside it.

Inline Functions Vs. Macros

- Since C macros use more textual substitution, this may result in unintended side-effects and inefficiency due to re-evaluation of arguments and order of operations.
- Compiler errors within macros are often difficult to understand, because they refer to the expanded code, rather than the code the programmer typed.
- Debugging information for inline code is usually more helpful than that of macro-expanded code .

Call Back

- The Call back function is just a normal calling using a pointer to function.
- It is always used to make a needed call from a lower layer to upper layer like calling a function from the scheduler component by timer component to provide it with the needed configurations.

Call Back Example

```
#include <stdio.h>
#include <stdlib.h>
typedef void (* t_call_back) (int, int);
void test_call_back(int a, int b)
{
    printf("In call back function, a:%d \t b:%d \n", a,b);
}
void call_callback_func(call_back back)
{
    int a = 5;
    int b = 7;
    back(a, b);
}
```

Call Back Example

```
void main(void)
{
    int ret = SUCCESS;
    t_call_back back;
    back = test_call_back;
    call_callback_func(back);
    return ret;
}
```

Modular Programming

Modular Programming

- Introduction
- Header Files
- Large Scale projects
- Data Sharing – extern keyword
- Data Hiding – static keyword

Introduction

- Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

Introduction

- Modules:
 - Conceptually, modules represent a separation of concerns, and improve maintainability by enforcing logical boundaries between components.
 - Modules are typically incorporated into the program through interfaces. A module interface expresses the elements that are provided and required by the module.
 - The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface.

Header Files

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

Header Files

file1.h

```
int add(int,int);
```

file2.h

```
#include "file1.h"
```

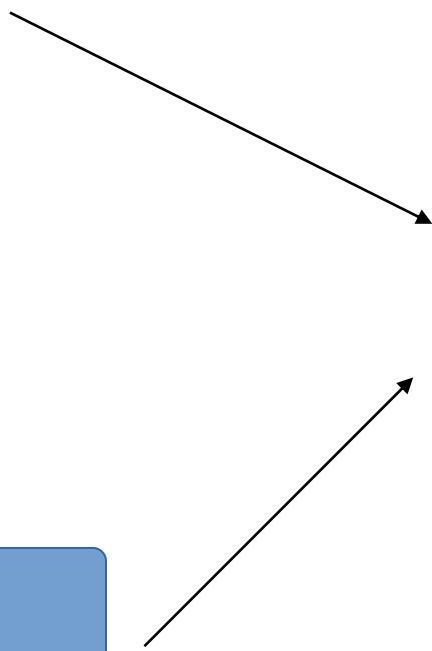
```
int sub(int,int);
```

main.c

```
#include "file1.h"  
#include "file2.h"
```

```
int main() {  
/* code */  
return 0;  
}
```

file1.h included twice!!



Large Scale Project

- Large projects may potentially involve many hundreds of source files (modules).
- Global variables and functions in one module may be accessed in other modules.
- Global variables and functions may be specifically hidden inside a module.
- Maintaining consistency between files can be a problem.

Large Scale Project

- Data Sharing

```
extern float step;

void print_table(double, float);

int main(void)
{
    step = 0.15F;

    print_table(0.0, 5.5F);

    return 0;
}

#include <stdio.h>

float step;

void print_table(double start, float stop)
{
    printf("Celsius\tFarenheit\n");
    for(;start < stop; start += step)
        printf("%.1lf\t%.1lf\n", start,
               start * 1.8 + 32);
}
```

Large Scale Project

- Data Hiding: When static is placed before a global variable, or function, the item is locked into the module

```
static int entries[S_SIZE];
static int current;

void push(int value)
{
    entries[current++] = value
}

int pop(void)
{
    return entries[--current];
}

static void print(void)
{
}
```

```
void      push(int value);
int      pop(void);
void      print(void);
extern int entries[];

int main(void)
{
    push(10); push(15);
    printf("%i\n", pop());
    entries[3] = 77; X
    print(); X
    return 0;
}
```

Large Scale Project

```
extern float step;

void print_table(double, float);

int main(void)
{
    step = 0.15F;

    print_table(0.0, 5.5F);

    return 0;
}

#include <stdio.h>
float step;

void print_table(double start, float stop)
{
    printf("Celsius\tFarenheit\n");
    for(;start < stop; start += step)
        printf("%.1lf\t%.1lf\n", start,
                           start * 1.8 + 32);
}
```

Large Scale Project

- Recommendations to follow:
 - Use Header Files
 - Maintain consistency between modules by using header files.
 - Place Module definitions and configurations.
 - Place an extern declaration of a variable in Module.c in a Module.h file.
 - Place a prototype of a non static (i.e. Sharable) function in a Module.h file.
 - Place the user defined data type declaration like structures, unions and enums declaration in a Module.h

Header Files

- Header files should not be nested.
- It is common to put the following into each .h file to prevent accidental double-inclusion.
 - `#ifndef EXAMPLE_H`
 - `#define EXAMPLE_H`
 - `/* body of example.h file */`
 - `#endif /* EXAMPLE_H */`
- This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

Lab #3

- Design a program composed of the following:
- Math Library:
 - MathLib.c and MathLib.h
 - unsigned char Math_Add(unsigned char param_1, unsigned char param_2);
- Error Log:
 - ErrorLog.c and ErrorLog.h
 - void ErrLog_Log(void);
- Main:
 - Main.c
 - Takes 2 8-bit variables from the user
 - Call Math_Add()
 - Detect if the value returned is a valid addition value or not
 - If the value is invalid, then report the error through calling ErrLog_Log()
- Hint: Error can be caused due to addition overflow

5. Special Data Types

Outline

- The Course consists of the following topics:
 - Structures
 - Unions
 - Bitfields
 - Enumerations
 - Typedef

Structure Data Type

- Collections of related variables under one name.
- Structures may contain variables of many different data types in contrast to arrays that contain only elements of the same data type.
- Structure is not a variable declaration, but a type declaration.
- The variables in a structure are called members and may have any data type, including int or char or arrays or other structures.
- Each structure definition must end with a semicolon.

Structures – Structure Declaration

- Example of employee structure data type declaration:

```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age; char gender;  
    double hourly_salary;  
};
```

- Structure variables(objects or instants) are declared like variables of other types: *struct employee emp1,emp2;*

Structures – Structure Members

- Structure members can be initialized at declaration.

```
struct employee person = {"Ahmed","Rafik",25,"m",100.5};
```

- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.

```
struct fruit
{
    char *name;
    char *color;
}fruit1;
fruit1.name = "apple";
```

```
struct vegetable
{
    char *name;
    char *color;
}veg1;
Veg1.name = "tomatoes"
```

Structures – Array of Structures

```
#include<stdio.h>
#include<string.h> //for strcpy function
struct student {
    int id;
    char name[30];
};

void main() {
    struct student record[2];
    // 1st student's record
    record[0].id=1;
    strcpy(record[0].name, "Khaled");
    // 2nd student's record
    record[1].id=2;
    strcpy(record[1].name, "Ayman");
    printf("%d %s \n %d %s", record[0].id,
    record[0].name, record[1].id,
    record[1].name);
}
```

Structures – Pointers to Structures

```
struct employee emp3, *emp_ptr;  
emp_ptr = &emp3;  
(*emp_ptr).age = 22;  
(*emp_ptr).hourly_salary = 70;
```

Structures – Pointers to Structures

- The type of the variable `emp_ptr` is pointer to a employee structure.
- In C, there is a special operator „->“ which is used as a shorthand when working with pointers to structures. It is officially called the structure pointer operator or arrow.
- the last two lines of the previous example could also have been written as:

```
struct employee emp3, *emp_ptr;
```

```
emp_ptr = &emp3;
```

```
emp_ptr -> age = 22;
```

```
emp_ptr -> hourly_salary = 70;
```

Structures – Self-Referential Structures

- A structure cannot contain an instance of itself.
- Example:

```
struct employee {  
    char firstName[ 20 ];  
    char lastName[ 20 ];  
    int age;  
    char gender;  
    double hourlySalary;  
    struct employee emp; // error  
    struct employee *emp_ptr; // pointer  
};
```

Structures – Structure Members

- To access a given member the dot notation is used.
- The “dot” is officially called the member access operator.

emp1.age = 25;

- Structure member can be treated the same as any other variable of that type. For example the following code:

emp2.age = emp1.age + 2;

Lab #1

- Implement a code that provides Set Calendar, and Show Calendar:
 - Create structure type with the elements day, month, and year
 - Create void SetCalendar(unsigned char day, unsigned char month, unsigned int year);
 - Create void GetDay(void);
 - Create void GetMonth(void);
 - Create void GetYear(void);
- Create test code to test

Typedef

- It is used to define your own data types.
- Example:
 - *typedef unsigned char uint8;*
 - *typedef signed char sint8;*
 - *typedef unsigned short uint16;*
 - *typedef signed short sint16;*
 - *typedef unsigned long uint32;*
 - *typedef signed long sint32;*
 - *typedef int* ptoi;*

Typedef

```
struct  
employee{ char  
name[20]; int age;  
};  
struct employee  
e1,*e2;
```

```
struct employee  
{ char name[20]; int  
age;  
};  
typedef struct employee EMP; EMP  
e1,*e2;
```

```
typedef struct{ char  
name[20]; int age;  
}EMP;
```

```
EMP e1,*e2;
```

Typedef

- Boolean in C Language

- `#define False 0`

- `#define True 1`

- `typedef unsigned char bool;`

```
typedef enum {  
    False=0,  
    True=1  
}bool;
```

- Which method typedef or #define is preferred in declaring data types and why?

```
#define p_newtype struct newtype *p_newtype p1, p2;
```

```
typedef struct newtype * newtype_ptr;newtype_ptr p3,p4;
```

Exercise

- What will be the output of the program?

```
#include<stdio.h>

int main()
{
    typedef int LONG;
    LONG a=4;
    LONG b=68;
    float c=0;
    c=b;
    b+=a;
    printf("%d, ", b);
    printf("%f\n", c);
    return 0;
}
```

Lab #2

- Repeat Lab #1, with the use of compiler independent types instead of unsigned char and unsigned int

Enumerations

- An enumeration, introduced by the keyword enum, is a set of integer enumeration constants represented by identifiers.
- Values in an enum start with 0, unless specified otherwise, and are incremented by 1 .

```
#define sun 0
#define mon 1
#define tue 2
#define wed 3
#define thu 4
#define fri 5
#define sat 6
void main(){
    int today = sun;
    if(today == mon){
        /* code*/
    }
}
```

```
enum day { sun, mon, tue, wed, thu, fri, sat };
void main(){
    enum day today = sun;
    if(today == mon) {
        /* code */
    }
}
```

Enumerations

- Since the first value in the preceding enumeration is explicitly set to 1, the remaining values are incremented automatically, resulting in the values 1 through 7.

```
enum day { sun = 1, mon, tue, wed, thu, fri, sat };
```

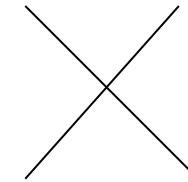
- The constants used may be specified.

```
enum pc_state {  
    shut_down = 1, sleep = 3, hibernate = 5, running = 7  
};
```

Enumerations

- The identifiers in an enumeration must be unique.

```
enum pc_state {  
    shut_down = 1, sleep = 3, sleep = 5, running = 7  
};
```



- Multiple members of an enumeration can have the same constant value.

```
enum pc_state {  
    shut_down = 1, sleep = 3, hibernate = 3, running = 7  
};
```

Exercise

- What will be the output of the program?

```
#include<stdio.h>

int main()
{
    enum color{red, green, blue};
    typedef enum color mycolor;
    mycolor m = red;
    printf("%d", m);
    return 0;
}
```

Lab #3

- Repeat Lab #1, with the use of enum types for each of day, month, and year

Unions

- Unions are C variables whose syntax look similar to structures, but act in a completely different manner.
- Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the largest member. (Unlike a structure where memory is reserved for all members).
- Data actually stored in a union's memory can be the data associated with any of its members.
- But only one member of a union can contain valid data at a given point in the program.
- It is the user's responsibility to keep track of which type of data has most recently been stored in the union variable.

Unions – Example

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
void main( )
{
    union Data d1;
    d1.i = 10;
    printf( "d1.i : %d\n", d1.i);
    d1.f = 220.5;
    printf( "d1.f : %f\n", d1.f);
    strcpy(d1.str,"Embedded C");
    printf( "d1.str : %s\n", d1.str);
    printf("Union size : %d bytes",sizeof(d1));
```

Results

```
data.i : 10
data.f : 220.5
data.str : Embedded C
Union size : 20 bytes
```

Unions – Example

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
void main( ){
    union Data d1;
    d1.i = 10;
    d1.f = 220.5;
    strcpy(d1.str,"Embedded C");
    printf( "d1.i : %d\n", d1.i);
    printf( "d1.f : %f\n", d1.f);
    printf( "d1.str : %s\n", d1.str);
    printf("Union size : %d
bytes",sizeof(d1));
```

Here, we can see that values of i and f members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Results

```
data.i : 1700949317
data.f :
668294057629101740
00000.000000
data.str : Embedded C
```

Exercise

- What will be the output of the program?

```
#include<stdio.h>

int main()
{
    union a
    {
        int i;
        char ch[2];
    };
    union a u;
    u.ch[0]=3;
    u.ch[1]=2;
    printf("%d, %d, %d\n", u.ch[0], u.ch[1], u.i);
    return 0;
}
```

Bit Fields

- C language enables you to specify the number of bits in which an unsigned or int member of a structure or union is stored.
- To define bit fields, Follow unsigned or int member with a colon (:) and an integer constant representing the width of the field.

```
struct my_bitfield
{
    unsigned short_element: 1; // 1-bit in size
    unsigned long_element: 8; // 8-bits in size
};
```

Bit Fields

- Unnamed bit field used as padding in the structure and nothing may be stored in these bits.
- Unnamed bit field with a zero width is used to align the next bit field on a new storage-unit boundary.

```
struct example
{
    unsigned a : 13;
    unsigned : 19;
    unsigned b : 20;
};
```

```
struct example
{
    unsigned a : 13;
    unsigned : 0;
    unsigned b : 3;
};
```

Bit Fields – Use Case

- Used to define the peripheral register bits inside Micro-controller.

```
typedef unsigned char byte;  
typedef union {  
    byte Data;  
    struct {  
        byte BIT0 :1; /* Register Bit 0 */  
        byte BIT1 :1; /* Register Bit 1 */  
        byte BIT2 :1; /* Register Bit 2 */  
        byte BIT3 :1; /* Register Bit 3 */  
        byte BIT4 :1; /* Register Bit 4 */  
        byte BIT5 :1; /* Register Bit 5 */  
        byte BIT6 :1; /* Register Bit 6 */  
        byte BIT7 :1; /* Register Bit 7 */  
    } Bits;  
} HW_Register;
```

Lab #4

- Implement a program, that holds 2 functions for setting and clearing flags of a variable; in addition to a function that prints the variable's value
 - Create an 8-bit union variable
 - Create void SetFlag(uint8 bit_number);
 - Create void ClearFlag(uint8 bit_number);
 - Create void PrintFlag(void);
- Create test cases to test the program behavior

6. Pointers, Arrays, and Strings

Outline

- The Course consists of the following topics:
 - Pointers
 - Introduction to Pointers
 - Passing by Reference
 - Pointer to Function
 - Pointer Arithmetic
 - Constants

Outline

- The Course consists of the following topics:
 - Arrays
 - Arrays
 - Multi-dimensional Array
 - Errors
 - Pointers vs. Arrays
 - Array of Pointers
 - Read your Pointer
 - Strings
 - String
 - String Library
 - Array of Strings

Pointers

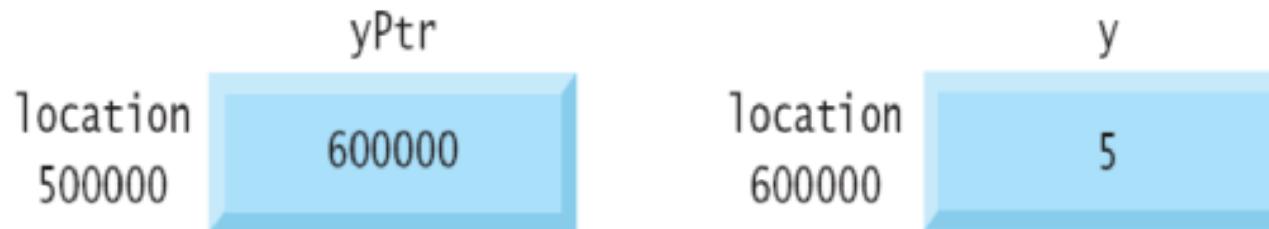
- Pointers are variables whose values are memory addresses.
- Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an address of a variable that contains a specific value.
- The & or address operator, is a unary operator that returns the address of its operand.

Pointers

- For example, assuming the definitions :

int y = 5;

*int *yPtr = &y;*



Addresses and Pointers in C

- Variable declarations :

int x, y;

- Pointer declarations use * :

*int *ptr;*

- Assignment to a pointer :

ptr = &x;

& = 'address of value'

** = 'value at address' or 'dereference'*

Addresses and Pointers in C

- To use the value pointed to by a pointer we use dereference (*)
- Examples:
 - If $\text{ptr} = \&x$ then $y = *ptr + 1$ is the same as $y = x + 1$
 - If $\text{ptr} = \&y$ then $y = *ptr + 1$ is the same as $y = y + 1$

NULL Pointers

- Pointers can be defined to point to objects of any type.
- A pointer may be initialized to NULL, 0 or an address.
- A pointer with the value NULL points to nothing.

Arguments Passing

- There are two ways to pass arguments to a function:
 - Pass-by-value
 - Pass-by-reference.
- All arguments in C are passed by value.
- Return keyword may be used to return one value from a called function to a caller but Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the overhead of making a copy of the object). For these purposes, C provides the capabilities for simulating call by reference. Using pointers and the indirection operator.

Pass-by-Value

- All arguments in C are passed by value.

Pass-by-Reference

- Return keyword may be used to return one value from a called function to a caller.
- Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value.
- For these purposes, C provides the capabilities for simulating call by reference. Using pointers and the indirection operator.

Pass-by-Reference

- When calling a function with arguments that should be modified, the addresses of the arguments are passed. This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

Pointers to Functions

- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.

Pointers to Functions

- Definition:

*Return_Value (*ptr_name)(Argument_Type);*

- Initialization:

ptr_name=func_name;

- Calling:

ptr_name(argument_name);

or

*(*ptr_name)(argument_name);*

Pointers Arithmetic

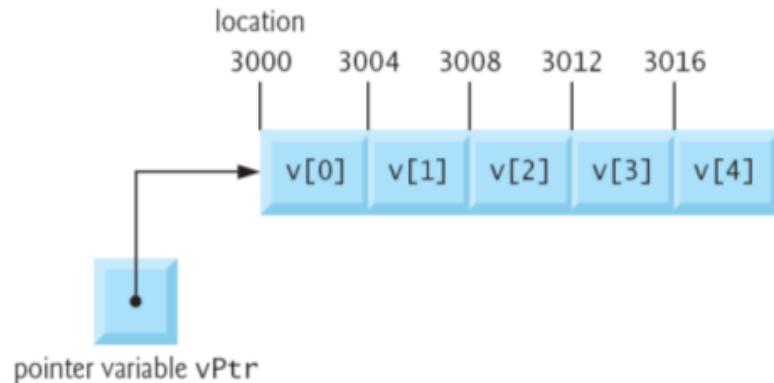
- A limited set of arithmetic operations may be performed on pointers, A pointer may be:
 - Incremented (++)
 - Decrement (--)
 - An integer may be added to a pointer (+ or +=)
 - An integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.

Pointers Arithmetic

- Assume that array int v[5] has been defined and its first element is at location 3000 in memory.
- Assume int pointer vPtr has been initialized to point to v[0] i.e., the value of vPtr is 3000.
- Variable vPtr can be initialized to point to array v with either of the statements:

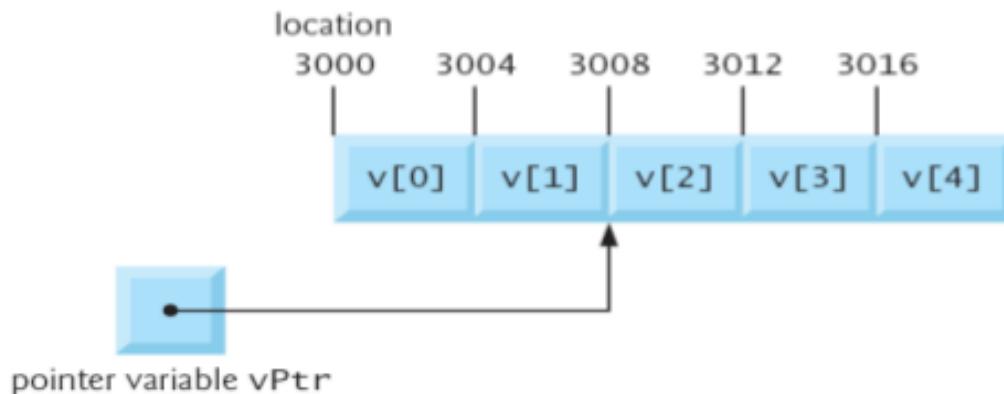
$vPtr = v;$

$vPtr = \&v[0]$



Pointers Arithmetic

- In conventional arithmetic, $3000 + 2$ yields the value 3002. This is normally not the case with pointer arithmetic.
- For pointer arithmetic, the statement : `vPtr += 2;` would produce 3008 ($3000 + 2 * 4$), assuming an integer is stored in 4 bytes of memory.
- In the array v, vPtr would now point to v[2].



Pointers Arithmetic

- Pointer variables may be subtracted from one another. For example, if vPtr points to the first array element v[0] contains the location 3000, and v2Ptr points to the third array element v[2] contains the address 3008, the statement : $x = v2Ptr - vPtr;$ would assign to x the number of array elements from vPtr to v2Ptr, in this case 2 (not 8).
- Pointer arithmetic is meaningless unless performed on an array.
- We cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent elements of an array.

Pointers Arithmetic

- Either of the statements increments the pointer to point to the next location in the array :

$++vPtr;$

$vPtr++;$

- Either of the statements decrements the pointer to point to the previous element of the array :

$--vPtr;$

$vPtr--;$

Const Variable

- The const qualifier enables you to inform the compiler that the value of a particular variable should not be modified.
- The values of constant variables not changeable through out the program.
- Example:

const int x = 10; // must be initialized int const x = 10; // the same.

Const Qualifier with Pointers

- A non constant pointer to constant data
 - Can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified by the pointer.
- Example:

```
int x = 10, y = 15;  
const int *ptr; ptr = &x;  
ptr = &y;  
y = 20;  
*ptr = 5; //error
```

Const Qualifier with Pointers

- A constant pointer to non constant data
 - A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer.
 - This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting.
 - Pointers that are declared const must be initialized when they’re defined.

Const Qualifier with Pointers

```
int x = 10,y = 15;  
int * const ptr = &x;  
*ptr = 5; // x=5  
ptr = &y; // error
```

```
void print_characters(char * const arr_ptr )  
{  
    int i = 0;  
    while(arr_ptr[i] != '\0')  
    {  
        printf("%c",arr_ptr[i]); i++;  
    }  
}
```

Exercise

- What is the size of pointers to int, char and float?

Lab #1

- Write a program to swap two variable values using pointers.

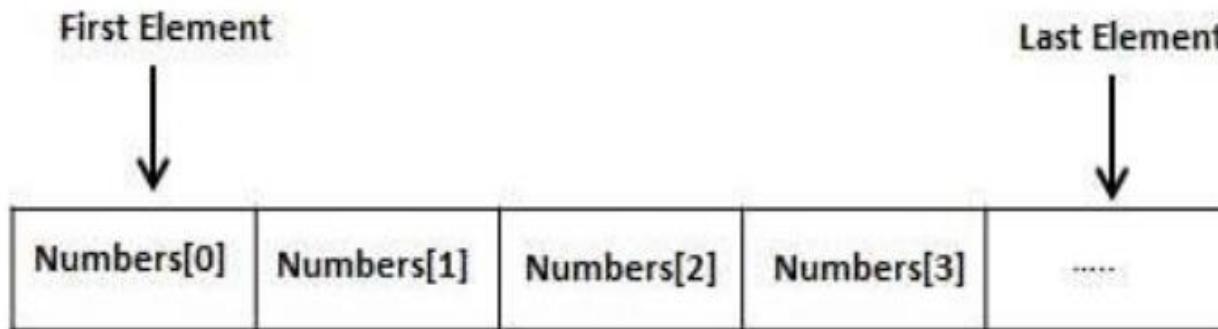
Arrays

Arrays

- C programming language provides a data structure called array, which can store a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Arrays

- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Arrays

- Important Note : array name is a constant pointer holds the first element address.

```
#include <stdio.h>

void main( void )
{
    char array[ 5 ]; /* define an array of size 5 */
    printf( "array = %p\n&array[0] = %p\n &array=%p\n"
        ,array,&array[0],&array);
}
```

Results:

```
array = 0012FF78
&array[0] = 0012FF78
&array = 0012FF78
```

Arrays

- Array Declaration:

data_type array_name [array_size];

- To declare a 5 elements array called balance of type double, use this statement:

double balance[5];

Arrays

- We can initialize array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 30.0};

- We can also assign a single element of the array:

balance[4] = 50.0;

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Arrays

- If there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list. For example:
double balance[] = {1000.0, 2.0, 3.4, 7.0, 30.0};
- It would create a five-element array

Arrays – Example

```
#include <stdio.h>

int main () {
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;
    /* initialize elements of array n */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }
    /* output each array element's value */
    for (j = 0;j < 10;j++ )
    {
        printf("Element[%d] = %d\n",j, n[j] );
    }
    return 0;
}
```

Results:

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

Arrays – Pass Single Element to Function

- Single element of an array can be passed in similar manner as passing normal variable to a function.

```
#include <stdio.h>
void display(int a)
{
    printf("%d",a);
}
int main()
{
    int c[ ]={2,3,4};
    /* Passing array element c[2] only. */
    display(c[2]);
    return 0;
}
```

Result: 4

Arrays – Passing Array to a Function

- To pass an array argument to a function, specify the name of the array without any brackets. For example, if array temp has been defined as :

int temp[5]={1,2,3,4,5};

- And it is required to pass it to start_temp_processing

start_temp_processing(temp,5)

Arrays – Passing Array to a Function

- Unlike char arrays that contain strings, other array types do not have a special terminator.

```
#include <stdio.h>
#define SIZE 6
void increment_array(int a[ ],int n) {
    int i;
    for( i = 0; i < n; i++) {
        a[i]++;
    }
}
void main(){
    int i;
    int arr[SIZE]={10,20,30,40,50,60};
    increment_array(arr,SIZE); //pass the array name only.
    for( i = 0; i < n; i++) {
        printf("arr[%d] = %d \n",i,a[i]);
    }
}
```

Results:

```
arr[0] = 11
arr[1] = 21
arr[2] = 31
arr[3] = 41
arr[4] = 51
arr[5] = 61
```

Arrays – Multi-Dimensional Array

- Multidimensional arrays:

`int b[3][5];`

- This array has 3 rows and 5 columns and can be listed as:

	0 th column	1 st column	2 nd column	3 rd column	4 th column
0 th row	<code>b[0][0]</code>	<code>b[0][1]</code>	<code>b[0][2]</code>	<code>b[0][3]</code>	<code>b[0][4]</code>
1 st row	<code>b[1][0]</code>	<code>b[1][1]</code>	<code>b[1][2]</code>	<code>b[1][3]</code>	<code>b[1][4]</code>
2 nd row	<code>b[2][0]</code>	<code>b[2][1]</code>	<code>b[2][2]</code>	<code>b[2][3]</code>	<code>b[2][4]</code>

Arrays – Init Multi-Dimensional Array

- In C, multidimensional arrays can be initialized in different number of Ways:
 - `int a[2][2] = {{1,3}, {-1,5}};`
 - `int a[2][2] = {1,3,-1,5};`
 - `int a[][2] = {{1,3}, {-1,5}}; // must specify the number of columns at least`
 - `int a[2][2] = {1}; // a[0][0]=1 and the other elements are equal to zero`

Arrays – Passing Multi-Dimensional Array

```
/* Instead of the below line, void Function(int c[ ][2]) { is also valid */
void display_array(int a[2][2]){
    int i,j;
    printf("Displaying Array Elements:\n");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j)
            printf("%d\n",a[i][j]);
}
void main(){
    int c[2][2],i,j;
    printf("Enter 4 numbers:\n");
    for(i=0;i<2;++i)
        for(j=0;j<2;++j)
            scanf("%d",&c[i][j]);
    /* passing multi-dimensional array to function */
    display_array(c);
}
```

Pointers Vs. Arrays

- Arrays and pointers closely related
- Array name like a constant pointer
- Pointers can do array subscripting operations
- Declare an array $b[5]$ and a pointer $bPtr$
$$bPtr = b;$$
- Array name actually a address of first element
OR
$$bPtr = \&b[0]$$
- Explicitly assign $bPtr$ to address of first element

Pointers Vs. Arrays

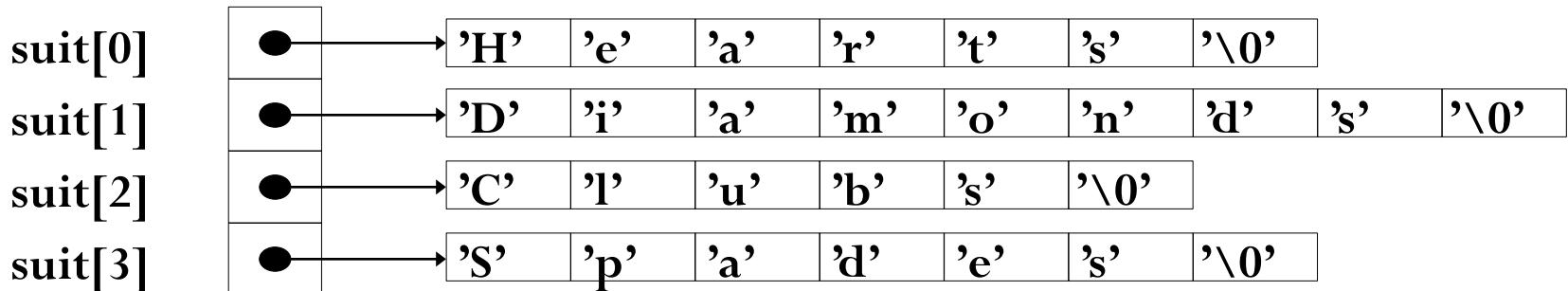
- Element $b[n]$ can be accessed by $*(b\text{Ptr} + n)$
 - n - offset (pointer/offset notation)
- Array itself can use pointer arithmetic.
 - $b[3]$ same as $*(b + 3)$
- Pointers can be subscripted (pointer/subscript notation)
 - $b\text{Ptr}[3]$ same as $b[3]$

Arrays of Pointers

- Arrays can contain pointers - array of strings

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

- String: pointer to first character
- char * - each element of suit is a pointer to a char
- Strings not actually in array - only pointers to string in array



- Suit array has a fixed size, but strings can be of any size.

Lab #2

- Implement a code that provides Set Weather Info, and Get Weather Info:
 - Create structure type with the elements temperature and humidity level
 - Create an array of 30 elements holding 30 structure elements for the weather of a month
 - Create void SetWeatherInfo (uint8 day, uint8* temp, uint8* humidity);
 - Create void GetWeatherInfo (uint8 day, uint8* temp, uint8* humidity);
- Create test code to test

Reading Pointer Declarations

- Up until now we have seen straightforward declarations:

```
long    sum;  
int*   p;
```

- Plus a few trickier ones:

```
void member_display(const struct Library_member *p);
```

- However, they can become much worse:

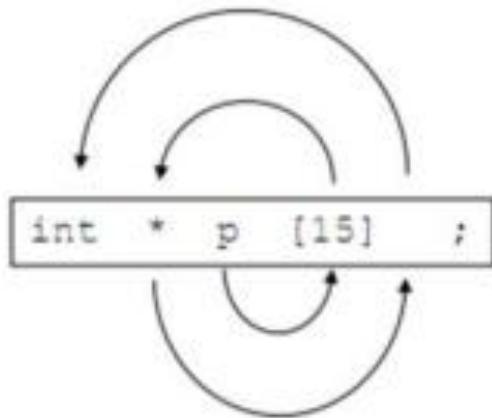
```
int    *p[15];  
float  (*pfa)[23];  
long   (*f)(char, int);  
double *(*(*n)(void))[5];
```

Reading Pointer Declarations

- SOAC
- Find the variable being declared
- Spiral Outwards Anti Clockwise
- On meeting: say:
 - * pointer to
 - [] array of
 - () function taking and returning
- Remember to read "struct S", "union U" or "enum E" all at once
- Remember to read adjacent collections of [) [) all at once

Reading Pointer Declarations

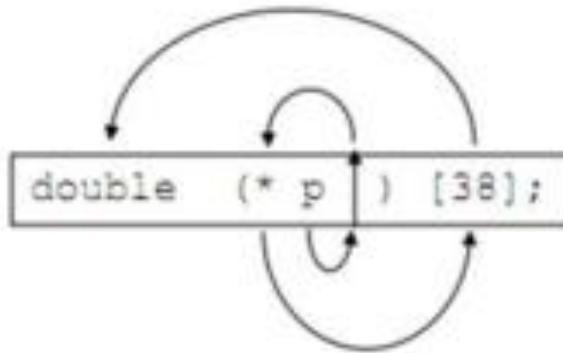
What is “int * p[15]” ?



p is an array of 15 pointers to integers

Reading Pointer Declarations

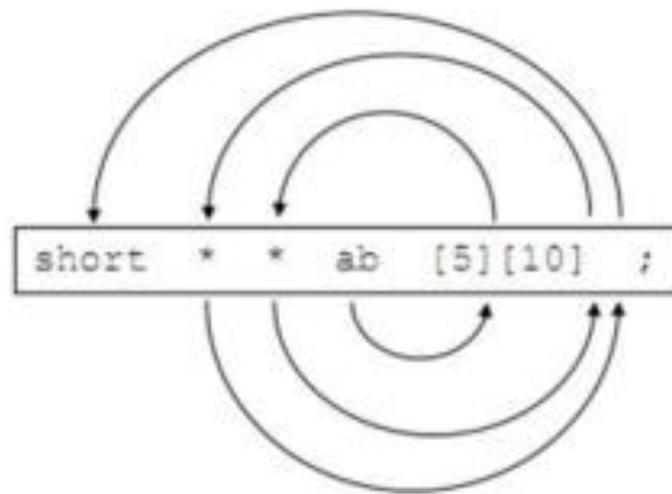
What is “double (*p)[38]” ?



p is a pointer to an array of 38 doubles

Reading Pointer Declarations

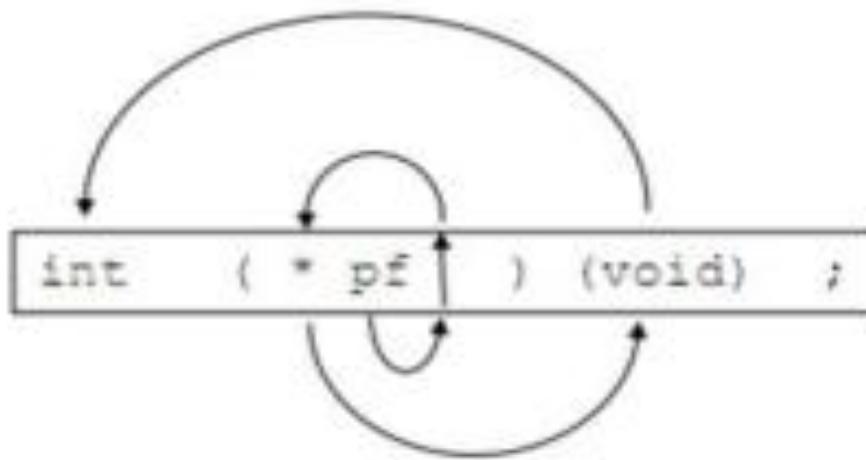
What is “short **ab[5][10]” ?



ab is an array of 5 arrays of 10 arrays of pointers
to pointers to short int

Reading Pointer Declarations

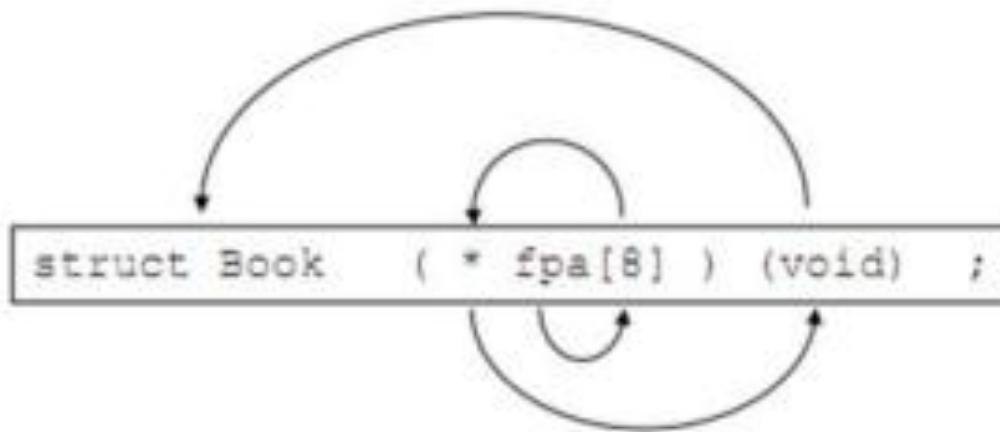
What is “int (*pf)(void)” ?



pf is a pointer to a function taking no parameters and returning an int

Reading Pointer Declarations

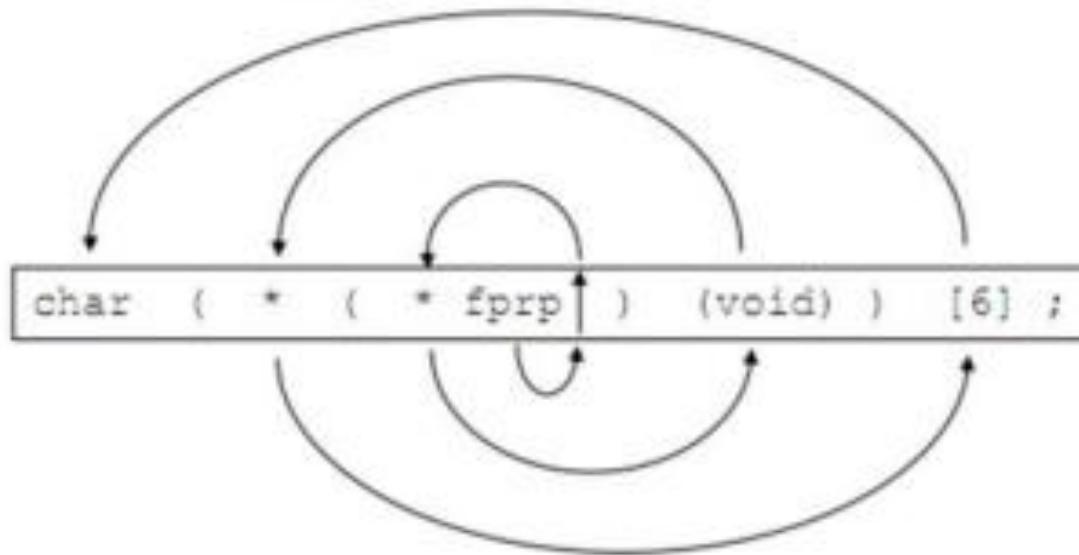
What is “struct Book (*fpa[8])(void)” ?



fpa is an array of 8 pointers to functions, taking no parameters, returning Book structures

Reading Pointer Declarations

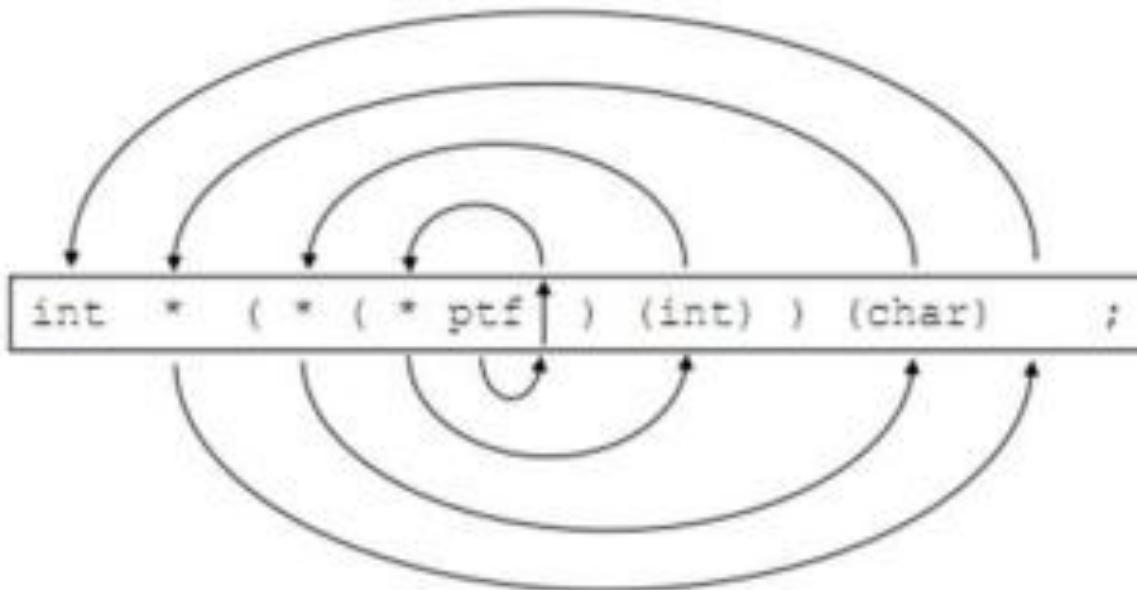
What is “char (*(*fprp)(void))[6]” ?



`fprp` is a pointer to a function taking no parameters returning a pointer to an array of 6 char

Reading Pointer Declarations

What is “int * (*(*ptf)(int))(char)” ?



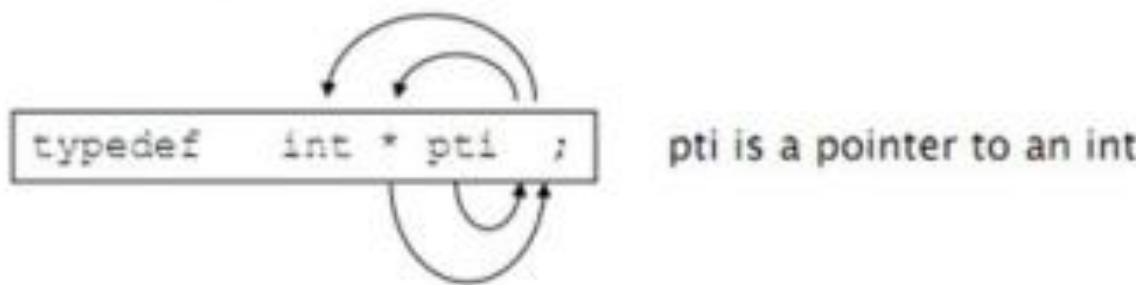
ptf is a pointer to a function, taking an integer, returning a pointer to a function, taking a char, returning a pointer to an int

Pointers Typedef

- The declaration can be broken into simpler steps by using typedef
- To tackle typedef, pretend it isn't there and read the declaration as for a variable When finished remember that a type has declared, not a variable

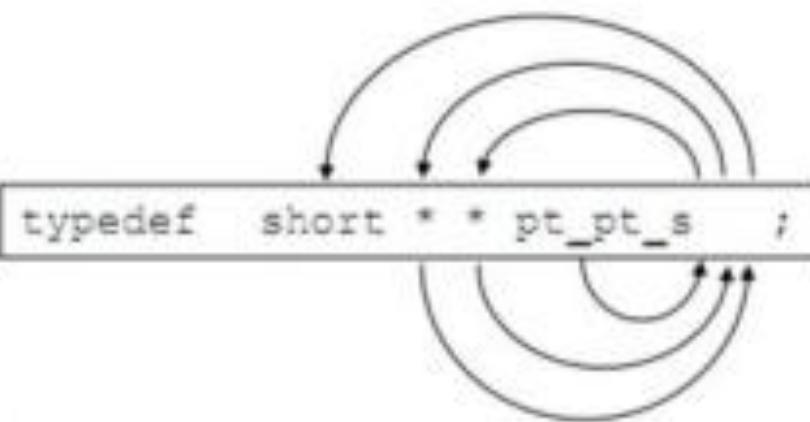
Reading Pointer Declarations

Simplify “int * p[15]”

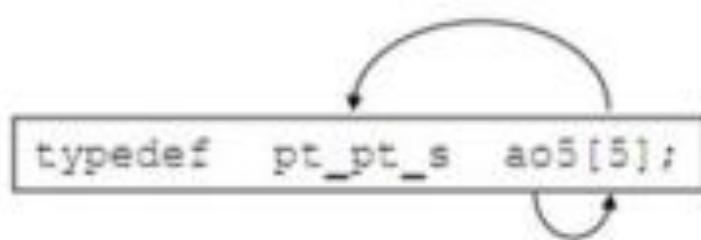


Reading Pointer Declarations

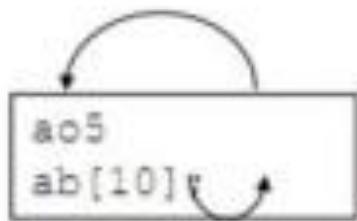
Simplify “short **ab[5][10]”



`pt_pt_s` is a pointer to a
pointer to a short



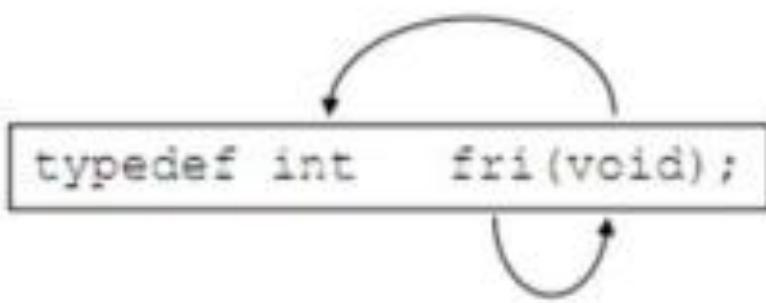
`ao5` is an array of 5 pointers
to pointers to short



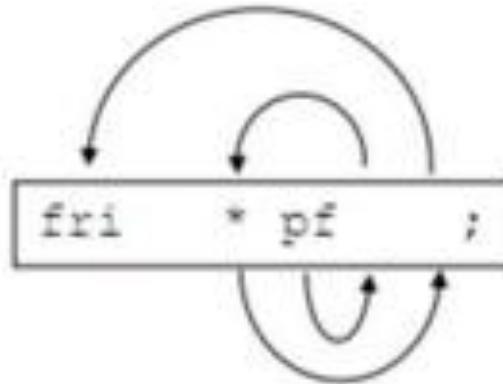
`ab` is an array of 10 arrays of
pointers to pointers to short

Reading Pointer Declarations

Simplify “int (*pf)(void)”



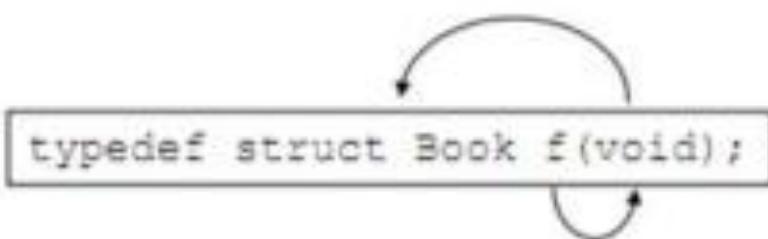
fri is a function, taking no parameters, returning an int



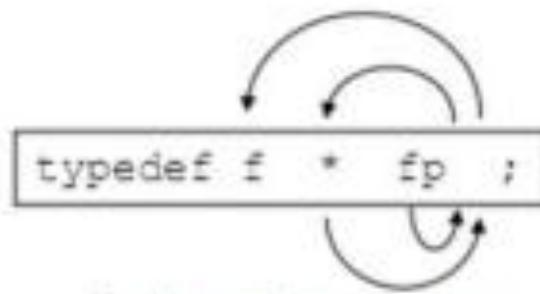
pf is a pointer to a function, taking no parameters, returning an int

Reading Pointer Declarations

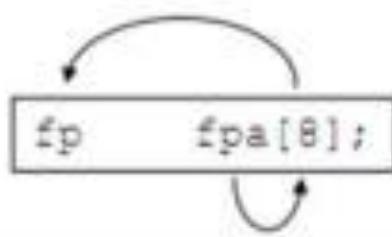
Simplify “struct Book (*fpa[8])(void)”



f is a function, taking no parameters, returning a Book structure



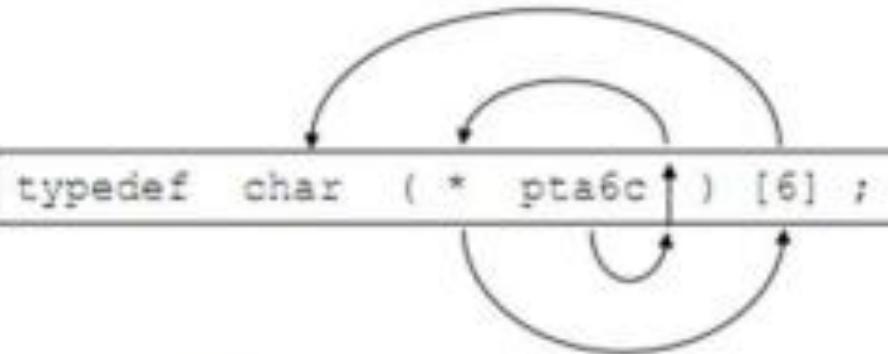
fp is a pointer to a function, taking no parameters, returning a Book structure



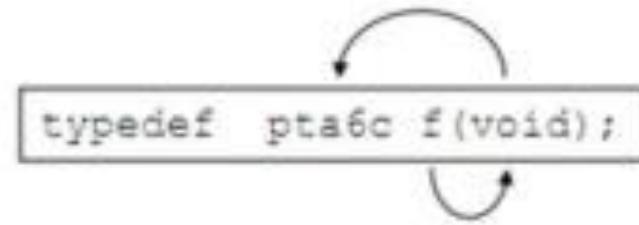
fpa is an array of 8 pointers to functions, taking no parameters, returning a Book structure

Reading Pointer Declarations

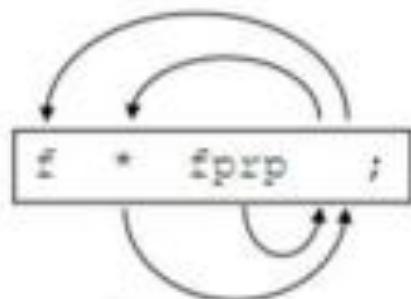
Simplify “char (*(*fprp)(void))[6]”



`pta6c` is a pointer to an array of 6 char



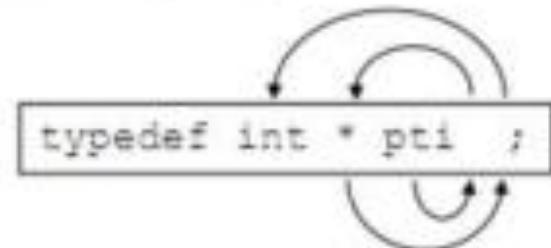
`f` is a function, taking no parameters, returning a pointer to an array of 6 char



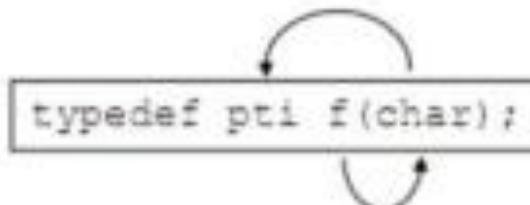
`fprp` is a pointer to a function, taking no parameters, returning a pointer to an array of 6 char

Reading Pointer Declarations

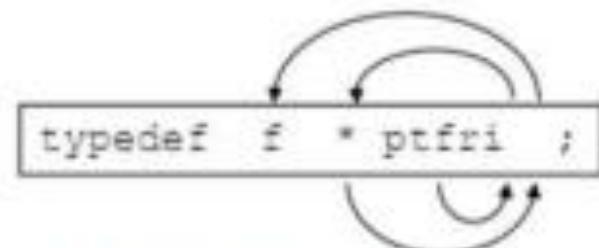
Simplify “int * (*(*ptf)(int))(char)”



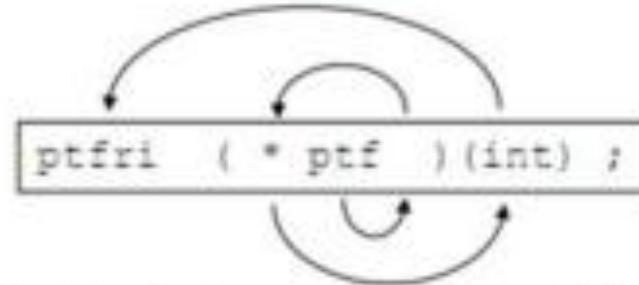
pti is a pointer to an int



f is a function, taking a char, returning



ptfrf is a pointer to



ptf is a pointer to a function, taking int, returning

Reading Pointer Declarations - Summary

- Don't even think that you cannot read any variable
- SOAC- Spiral Outwards Anti Clockwise
- To simplify complex pointer declarations , use typedef(s)

Strings in C

Strings in C

- Strings are 1D arrays of characters.
- It terminated by the null character „\0“ which is (naturally) called the end-of-string character.

Strings in C

char string1[6] = {"h","e","l","l","o"};

char string1[6] = "hello";

char string1[] = "hello";

Index	0	1	2	3	4	5
Variable	H	e	I	I	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Note: Don't forget that one character is needed to store the null character "\0", which indicates the end of the string.

Strings in C

```
void string_copy(char s1[],char s2[])
{
    int i = 0;
    while( s2[i] != „\0“ )
    {
        s1[i] = s2[i];
        i++;
    }
    s1[i] = s2[i];
}
void main()
{
    char str1[15],str2[15] = “Embedded C”;
    string_copy(str1,str2);
    printf(“str1 = %s\n”,str1);
}
```

Results:

str1 = Embedded C

String - Functions

- There are numerous functions defined in "string.h" header file for strings manipulation.

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

String – Functions

- Here are some examples of string functions in action:

char s1[30] = “Embedded Software”;

char s2[30] = “ahly club”;

char s3[30];

Function	Results
strlen(s1)	17
strlen(s2)	9
strcpy(s3,s1)	Embedded Software
strcmp(s1,s2)	Negative number
strcat(s2,”is the best”);	ahly club is the best

String – Array of Strings

- A string is an array of characters so, an array of strings is an array of arrays of characters.

```
void main() {  
    char str[2][20] = {"Embedded", "Systems"};  
    printf("str[0] = %s\n", str[0]);  
    printf("str[1] = %s\n", str[1]);  
    printf("str[1][3] = %c\n", str[1][3]);  
}
```

Results:

str[0] = Embedded

str[1] = Systems

str[1][3] = t

Lab #3

- Write a program that has a string “LabTest” as a variable; and you have to copy it in another variable
- Constraint: Use sizeof operator in your solution

7. Data Structures

Outline

- The Course consists of the following topics:
 - Introduction
 - Arrays
 - Linked Lists
 - Queue
 - Stack

Introduction

- Data structures is a way of organizing data items by considering its relationship to each other.
- Data structures affects the design of structural and functional aspects of a program

Data Structures

- Primitive DS:
 - Directly operated on by machine instructions
 - Integers, floating point numbers, characters, strings, pointers
- Non-Primitive DS:
 - Arrays
 - Lists
 - Linear Lists
 - Stack
 - Queue
 - Non-linear Lists
 - Graph
 - Trees
 - Files

Arrays

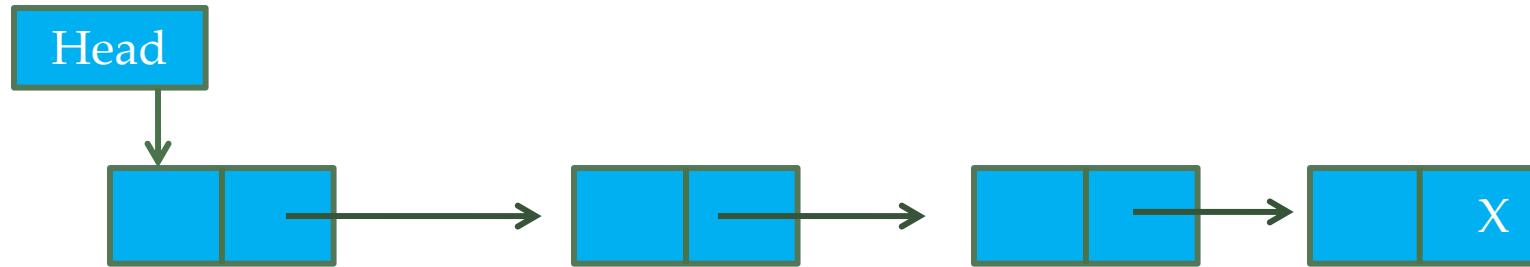
- The most frequently used data structure
- A collection of homogenous data elements described by a single name
- Can be single or multiple dimensional
- Elements are stored in successive memory locations.

Disadvantage of Arrays

- Fixed size
- Difficulty of insertion and deletion of elements

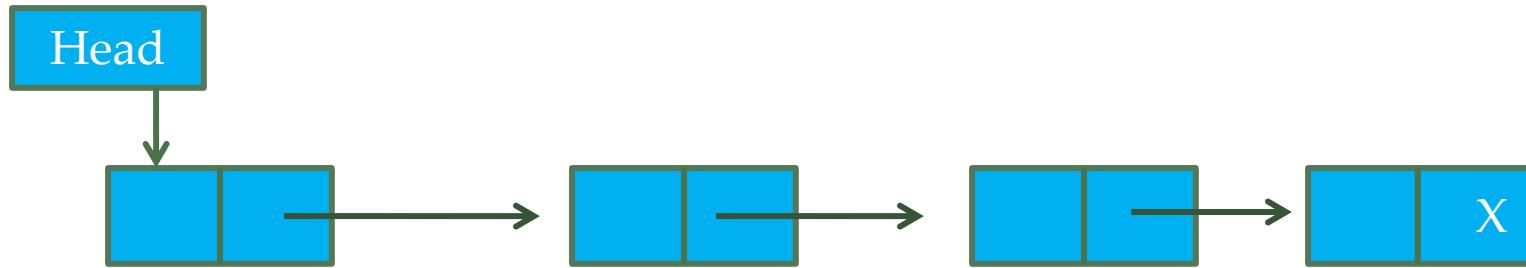
Linked Lists

- A set of varying number of elements to which insertion and deletion can be made.
- A linear list, has elements adjacent to each other.
- Each element is called a node

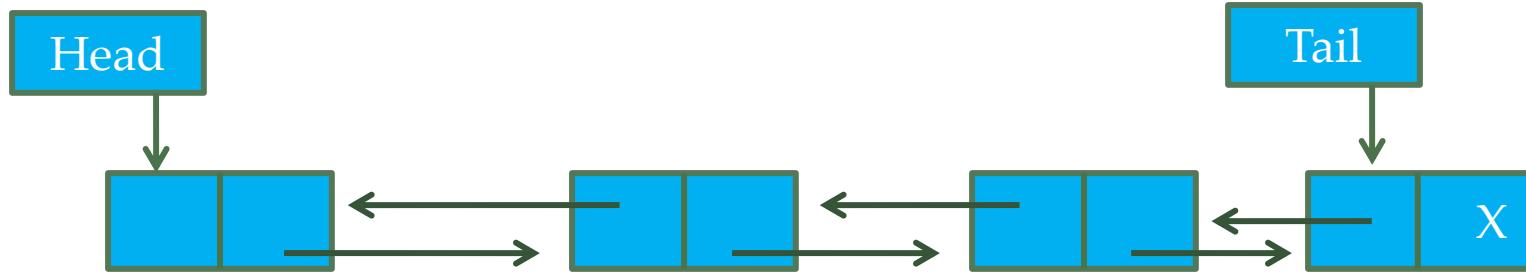


Linked Lists

- Single Linked List



- Double Linked List



Double Linked List

- Each node must contain:
 - Data saved
 - Pointer to the next node
 - Pointer to the previous node
- Example:

```
struct node
{
    struct node * pNext;
    struct node * pPrev;
    int data;
}

struct node * pHead
struct node * pTail
```

Operations on Double Linked List

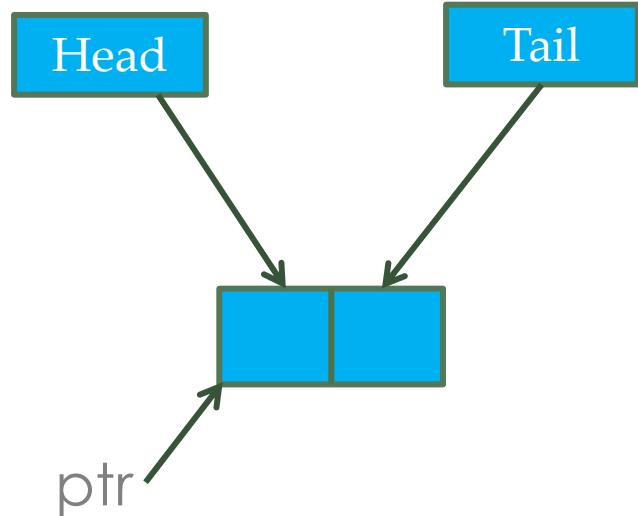
- Initially a list is empty (NULL/Empty List)
 - $pHead=pTail=NULL;$
- Addition
- Insertion
- Deletion
- Search (Linear Sequential Search)
- Free List

Insertion at LOC

- Have the following cases to test:
 - Empty List (pHead=pTail=NULL)
 - LOC=0 (same as Insertion At BEGIN)
 - LOC out of List (same as Insertion AT END)
 - LOC in the anywhere in list

Insertion at LOC

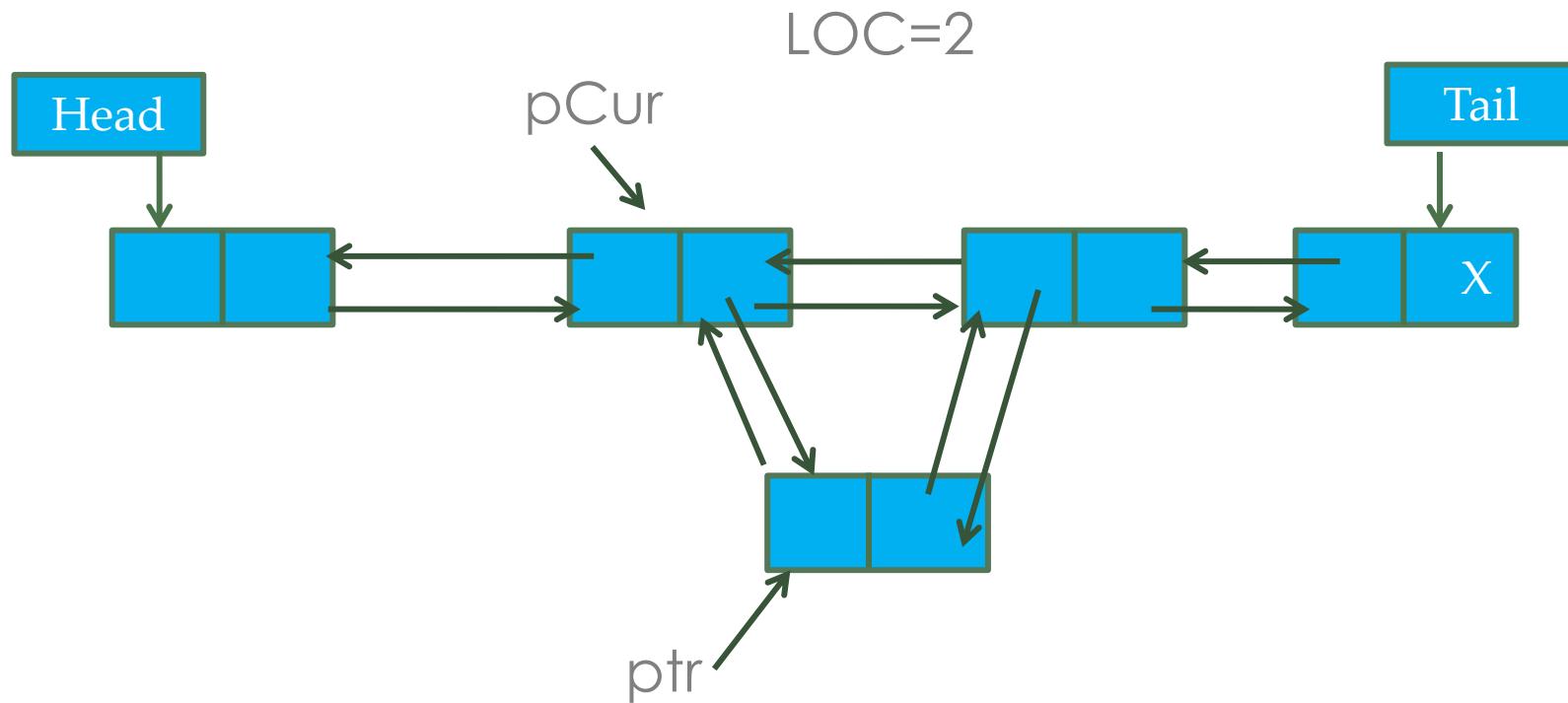
- Empty List ($pHead=pTail=NULL$)



- $pNext=pPrev=NULL$

Insertion at LOC

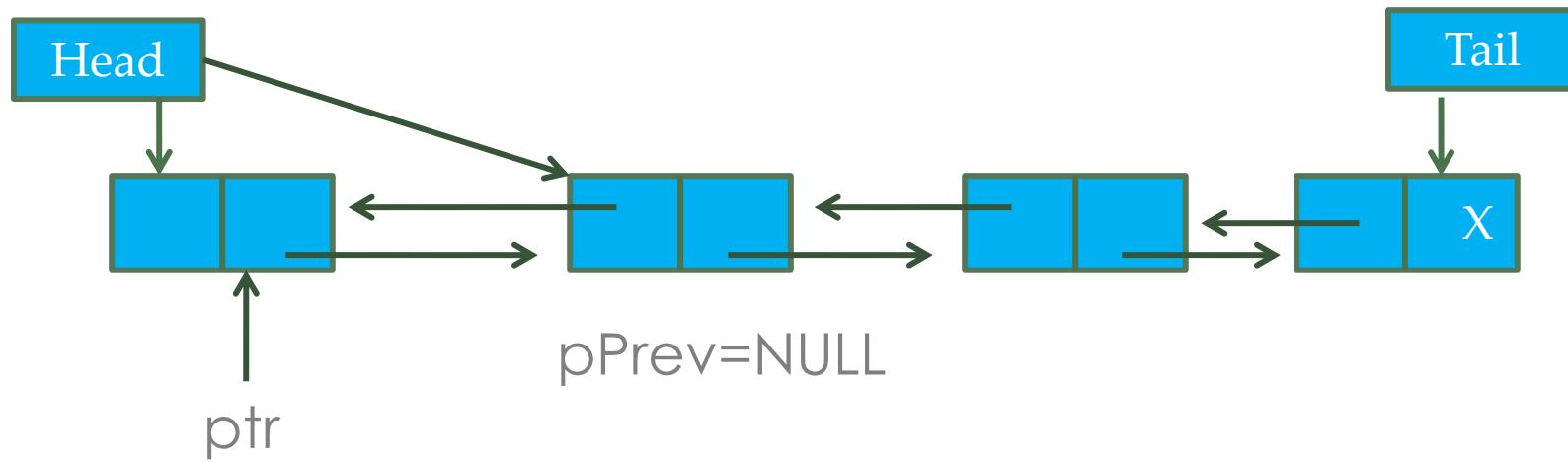
- LOC in the anywhere in list
 - Start from Head and iterate (LOC-1) number of times



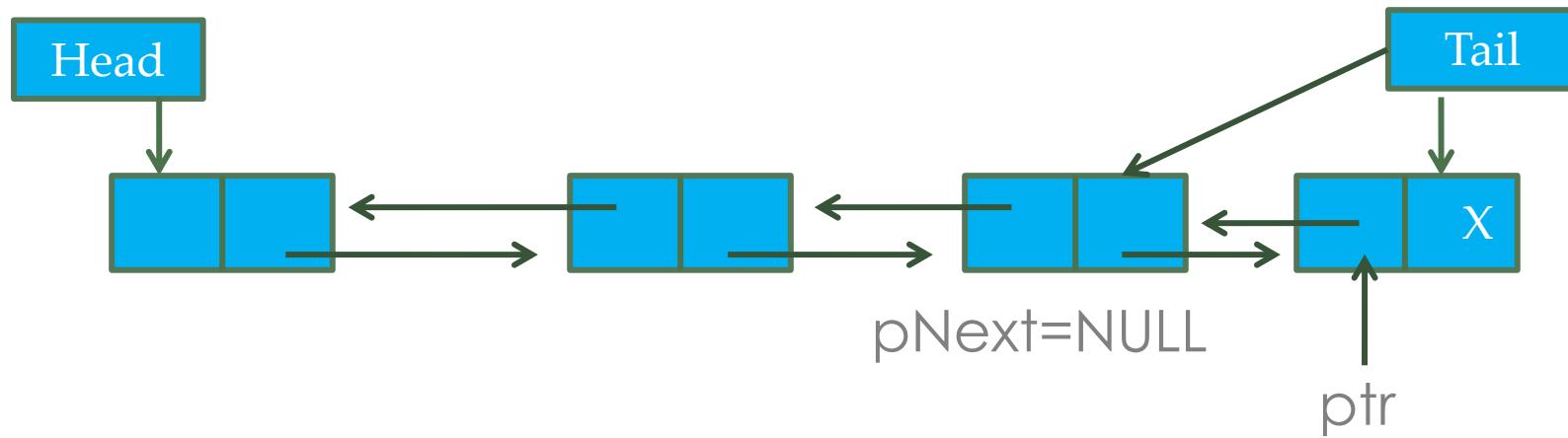
Deletion At LOC

- Have the following cases to test:
- Empty List ($pHead=pTail=NULL$) -> Noting to delete
- Only One NODE ($pHead=pTail !=NULL$) ->
 $pHead=pTail=NULL$
- $LOC=0$
- Delete Last Node
- LOC out of List -> Iterate till end and Noting to delete
- LOC in the anywhere in list

Deletion At LOC=0



Deletion At Tail



Implementation in C

```
struct node
{
    int data;
    struct node *pNext;
    struct node *pPrev;
};

struct node *pHead;
struct node *pTail;
struct node* createnode(int data);
int search(int data);
void printall(void);
int addnode(int data);
int insert(int data,int loc);
int delete (int loc);
```

Implementation in C

```
struct node* createnode(int data)
{
    struct node *ptr=NULL;
    ptr=(struct node*) malloc(sizeof(struct node));
    if(ptr)
    {
        ptr->data=data;
        ptr->pNext=ptr->pPrev=NULL;
    }
    return ptr;
}
```

Implementation in C

```
int addnode(int data)
{
    int flag=0;      struct node *ptr=createnode(data) ;
    if(ptr)
    {
        flag=1;
        if(pTail==NULL) //Empty List
            pTail=pHead=ptr;
        else //List exists so add at end
        {
            pTail->pNext=ptr;
            ptr->pPrev=pTail;
            pTail=ptr;
        }
    }
    return flag;
}
```

Implementation in C

```
int search(int data)
{
    struct node * pCur=pHead;
    int flag=0;
    while(pCur && !flag)
    {
        if(pCur->data==data)
        {
            flag=1;
        }
        else
            pCur=pCur->pNext;
    }
    return flag;
}
```

Implementation in C

```
int insert(int data, int loc)
{
    int flag=0,i=0;
    struct node *ptr,*pCur;
    ptr=createnode(data);
    if(ptr)
    {
        flag=1 ;
        if(!pHead)      //Empty List
            pHead=pTail=ptr;
        else if(loc==0)
            // as first location
        {
            ptr->pNext=pHead;
            pHead->pPrev=ptr;
            pHead=ptr;
        }
        else{ pCur=pHead;
        for(;(i<loc-1&&pCur) ;i++)
            pCur=pCur->pNext;
        if(!pCur || pCur==pTail)          // reached Tail
        {
            ptr->pPrev=pTail;          pTail->pNext=ptr;
            pTail=ptr;
        }
        else
        {
            pCur->pNext->pPrev=ptr;
            ptr->pNext=pCur->pNext;
            pCur->pNext=ptr;
            ptr->pPrev=pCur;
        }
    }
    return flag;
}
```

Queue (FIFO)

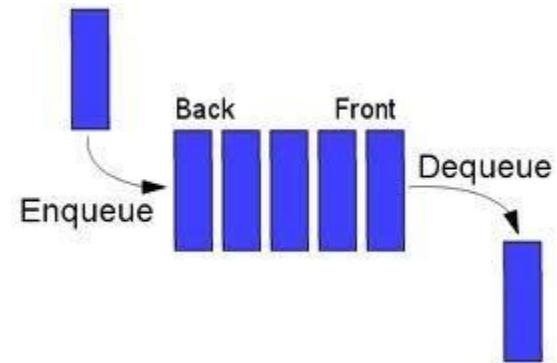
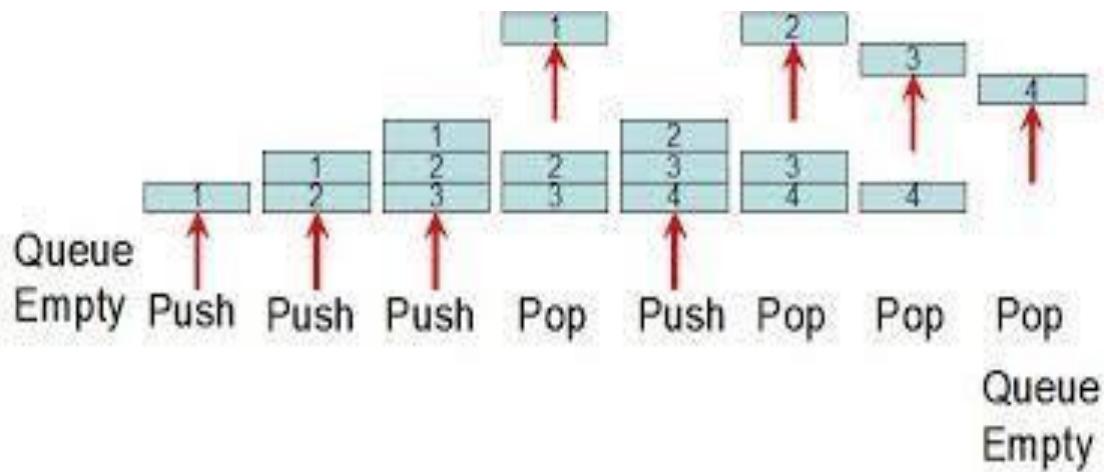
- First element inserted is first one retrieved
- Can't retrieve elements from middle of queue or end of queue
- Can't insert elements into middle or start of queue
- Examples:
 - Process/Thread Queue
 - Event Queue
 - Print Queue

Queue (FIFO)



Queue Implementation

- We need to know Front and Back of Queue
- Operations done on Queue:
 - EnQueue -> insert element at Back
 - DeQueue -> retrieve element from Front



Implementing Queue using Linked Lists

```
#define SIZE 3

struct node
{
    int data;
    struct node *pNext;
};

struct node *pHead;
struct node *pTail;
int count;

struct node* createnode(int data);

int enqueue(int data);
int dequeue(int *flag);
```

Implementing Queue using Linked Lists

```
struct node* createnode(int data)
{
    struct node *ptr=NULL;
    if(count<SIZE)
    {
        ptr=(struct node*) malloc(sizeof(struct node));
        if(ptr)
        {
            count++;
            ptr->data=data;
            ptr->pNext=NULL;
        }
    }
    return ptr;
}
```

Implementing Queue using Linked Lists

```
int enqueue(int data)
{
    int flag=0; struct node *ptr=createnode(data);
    if(ptr)
    {
        flag=1;
        if(pTail==NULL) //Empty List
            pTail=pHead=ptr;
        else //List exists so add at end
        {
            pTail->pNext=ptr;
            pTail=ptr;
        }
    }
    return flag;
}
```

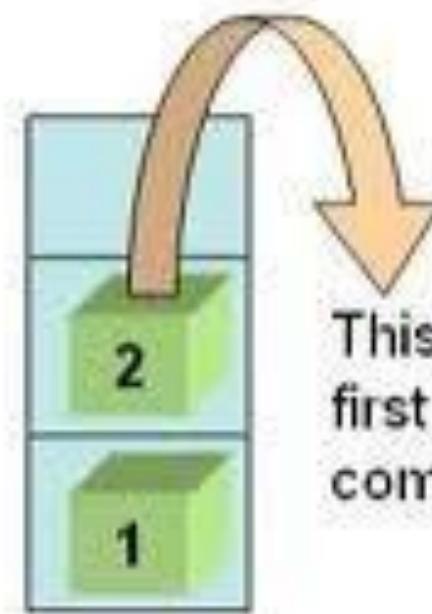
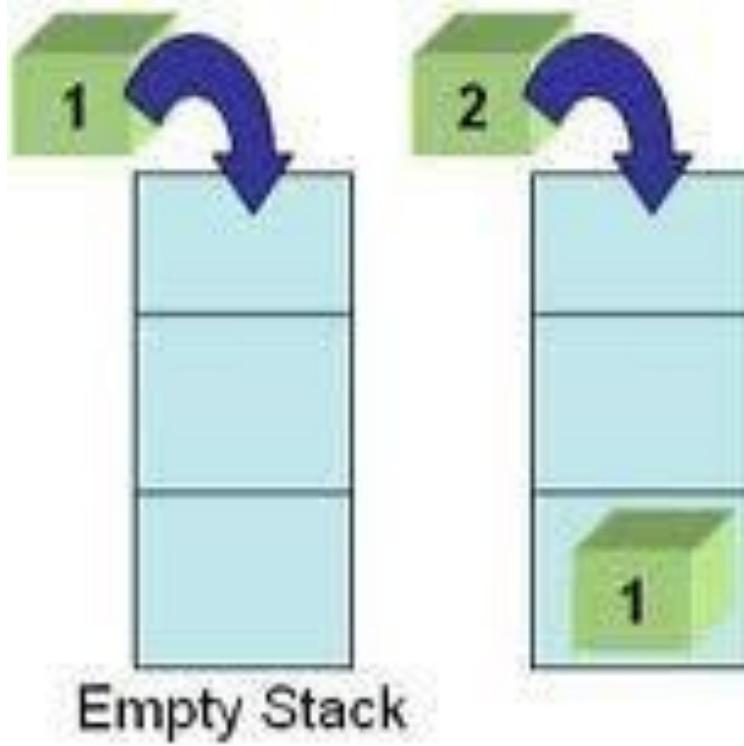
Implementing Queue using Linked Lists

```
int dequeue(int*flag)
{
    struct node *pCur=pHead;
    int data;
    *flag=0;
    if(pCur)
    {
        pHead=pHead->pNext;
        data=pCur->data;
        free(pCur); count--;
        *flag=1;
    }
    return data;
}
```

Stack (LIFO)

- Last element inserted is first one retrieved
- Can't retrieve elements from middle of stack or end of stack
- Can't insert elements into middle or end of stack
- Examples:
 - A pile of dishes
 - Call Stack
 - Undo Stack

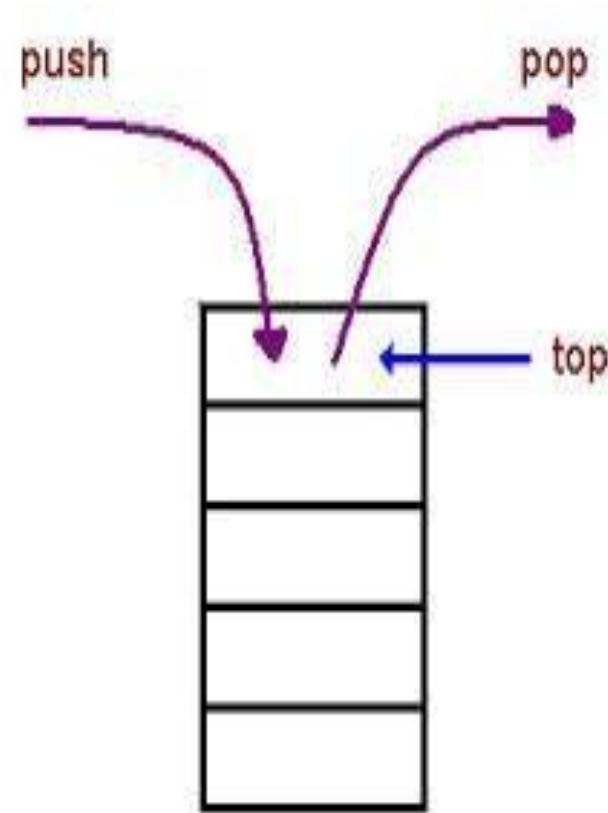
Stack



This will be the
first object to
come out.

Stack Implementation

- We need to know ToS (top of stack)
- Operations done on stack:
 - Push -> insert element at ToS
 - Pop -> retrieve element from ToS



Lab #1

- Create Stack-Module, with push and pop functions:
 - Source Code: StackSim.c
 - Header File: StackSim.h
 - Interfaces: bool push (uint8 data);
 - Interfaces: bool pop (uint8* data);
 - Configuration Parameters: STACK_SIZE = 10
- Create test code in main.c to test:
 - Normal Push and Pop operations
 - Write in case of full stack
 - Read in case of empty stack

8. Algorithms

Outline

- The Course consists of the following topics:
 - Searching Algorithms
 - Linear Search
 - Binary Search
 - Sorting Algorithms
 - Introduction
 - Selection Sort
 - Bubble Sort
 - Merge Sort

Linear Sequential Search

Linear Search (unsorted data): returns location

Initialize location to position of first item

Set found to false

Set moreToSearch to (have not examined last.info())

while moreToSearch AND NOT found

 if item equals location.info()

 Set found to true

 else

 Set location to location.next()

 Set moreToSearch to (have not examined last.info())

 if not found

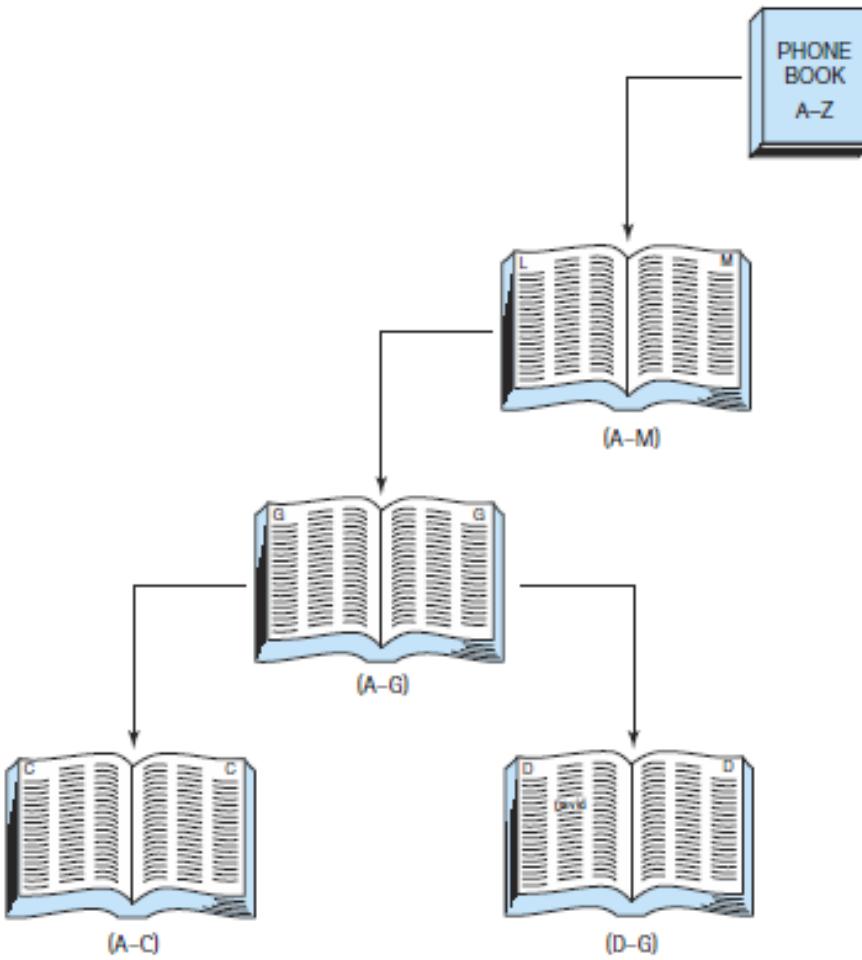
 Set location to NULL

return location

Analysis

- Best-case Scenario: find item at first location
 - $O(1)$
- Worst-case Scenario: Not found , or found but in last location.
 - $O(N)$
- Average-case Scenario:
 - $O(N/2)$
- How can knowing that a list is sorted, enhance linear search?

Binary Search



Binary Search

- Data in list must be sorted and stored sequentially.
- Divide and Conquer approach
- No enhanced performance for very small lists.

Binary Search

isThere (item): returns boolean

Set first to 0

Set last to numItems - 1

Set found to false

Set moreToSearch to (first <= last)

while moreToSearch AND NOT found

 Set midPoint to (first + last) / 2

 compareResult = item.compareTo(midPoint.info())

 if compareResult == 0

 Set found = true

 else if compareResult < 0

 Set last to midPoint - 1

 Set moreToSearch to (first <= last)

 else

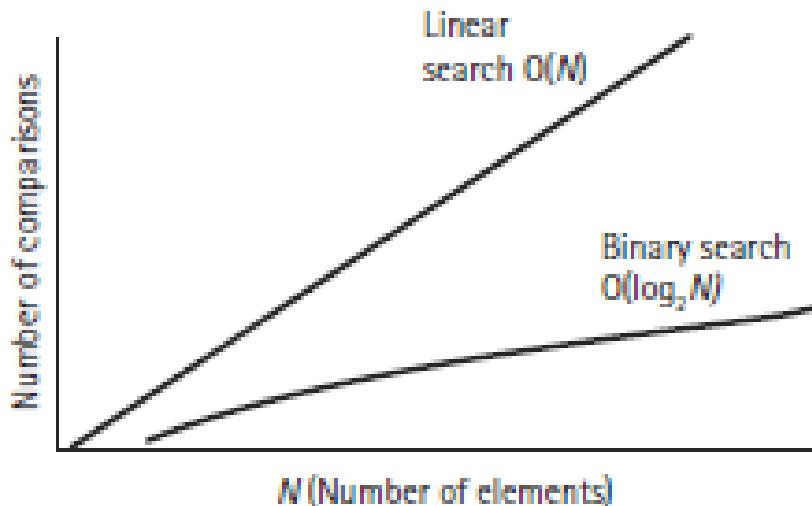
 Set first to midPoint + 1

 Set moreToSearch to (first <= last)

return found

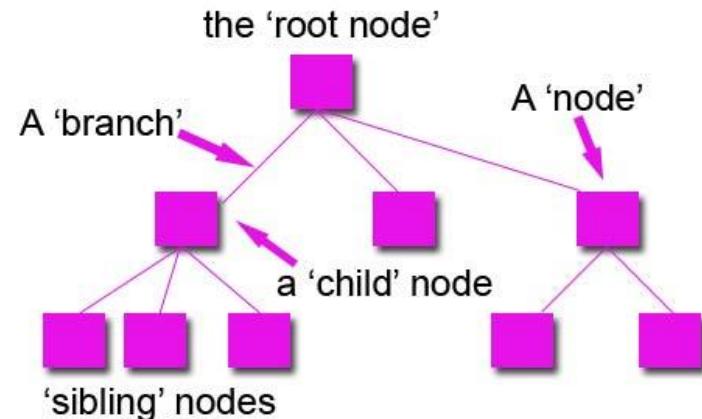
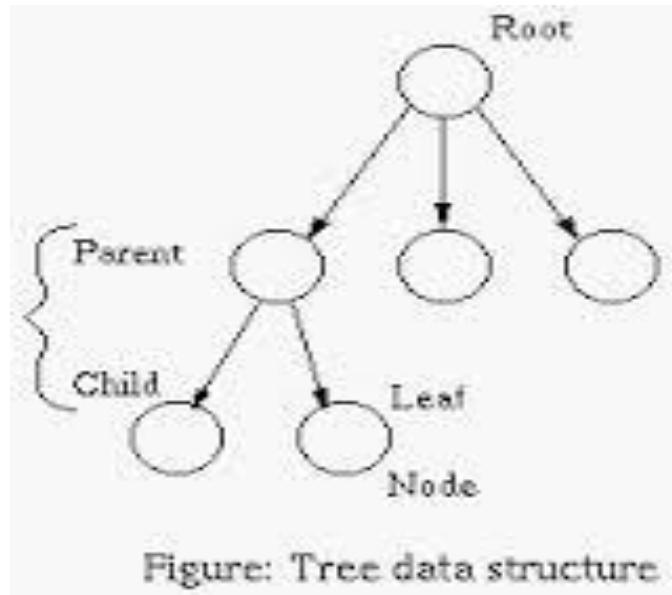
Analysis

- Best-case Scenario: find item at first location
 - $O(1)$
- Worst-case Scenario: Not found , or found but in last location.
 - $O(N)$
- Average-case Scenario:
 - $O(N/2)$



Binary Search Trees

- Trees are non-linear collection of nodes.
- Trees can be empty or have one root and zero or more subtrees
- Example: File system

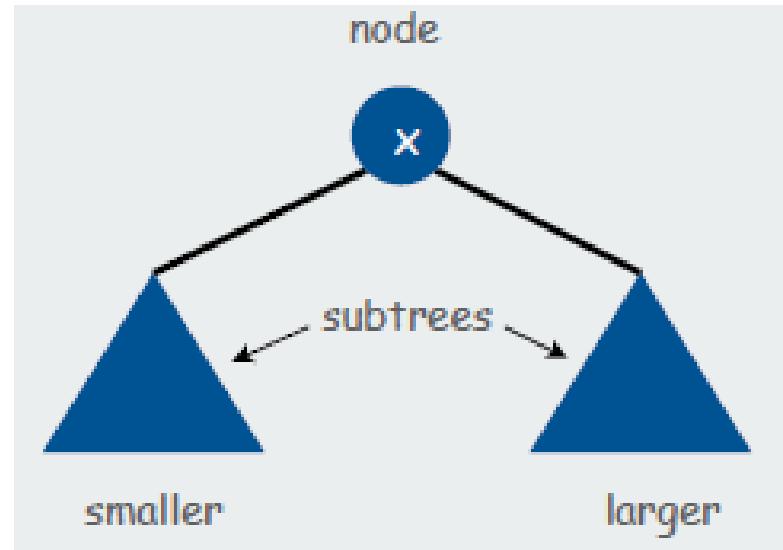
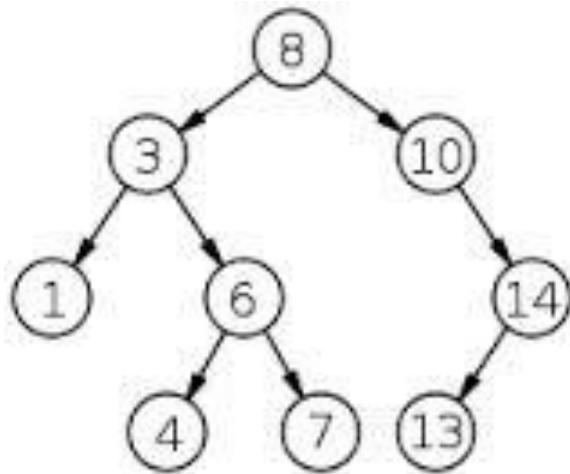


PARTS OF A TREE DATA STRUCTURE

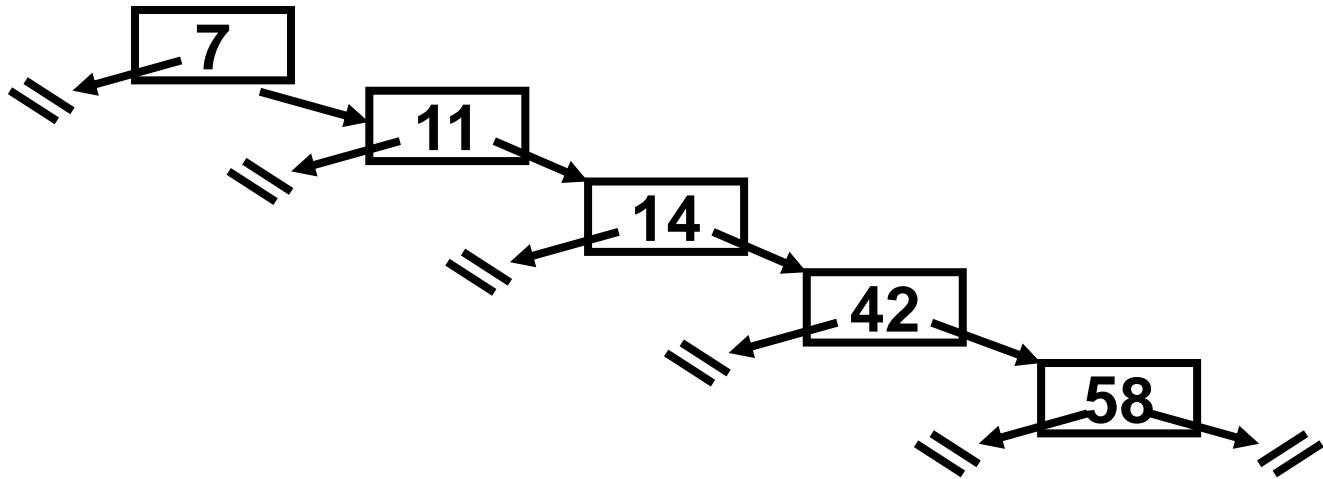
(c)www.teach-ict.com

Binary Search Trees

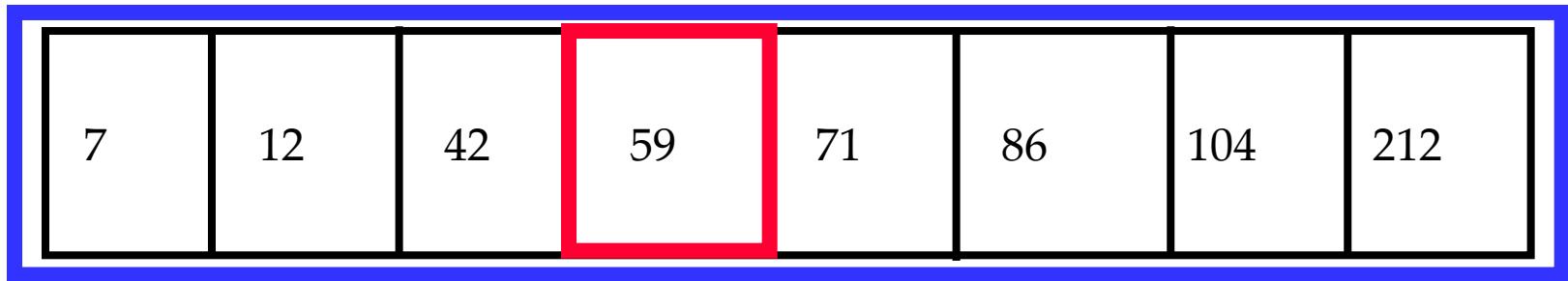
- BST: is a tree where no node can have more than 2 children. And all elements in the left subtree must be smaller than values in the right subtree



Binary Search Trees

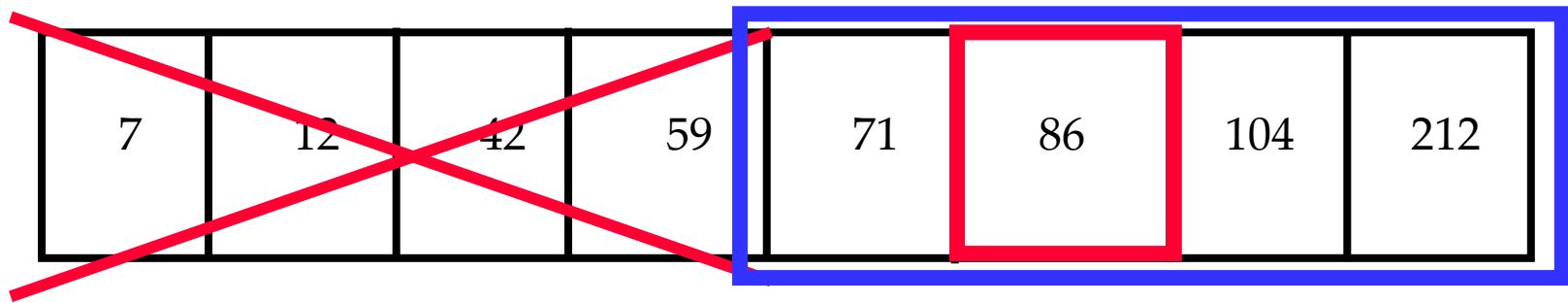


Binary Search Trees



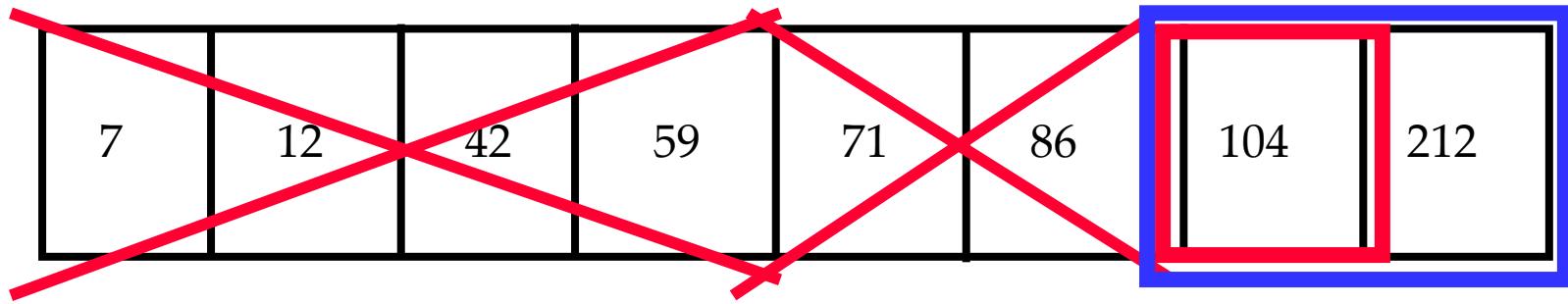
Looking for 89

Binary Search Trees



Looking for 89

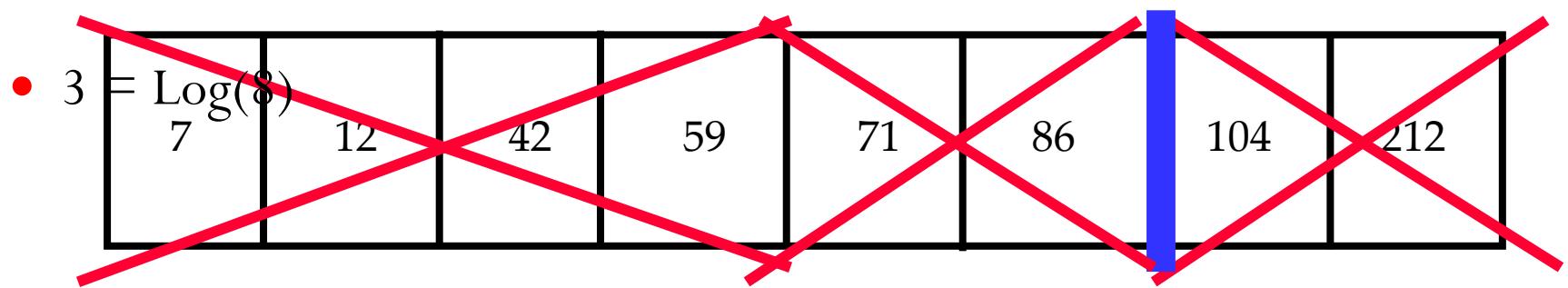
Binary Search Trees



Looking for 89

Binary Search Trees

- 89 not found – 3 comparisons



Lab #1

- Implement a function that searches for a given input value in an array, and return the index at which the value is located.
 - Use Merge Sort Algorithm
 - Function Prototype: *bool GetArrayIndex(uint8 value, uint8* array_index);*
 - The function should return ‘1’ if the value is found, otherwise return ‘0’

Sorting Algorithm

- Sorting a large number of elements can be an extremely time consuming operation.
- The key operation is comparing two values which is related to the number of elements in the list (N)

Selection Sort

values
[0]
126
[1]
43
[2]
26
[3]
1
[4]
113

(a)

values
[0]
1
[1]
43
[2]
26
[3]
126
[4]
113

(b)

values
[0]
1
[1]
26
[2]
43
[3]
126
[4]
113

(c)

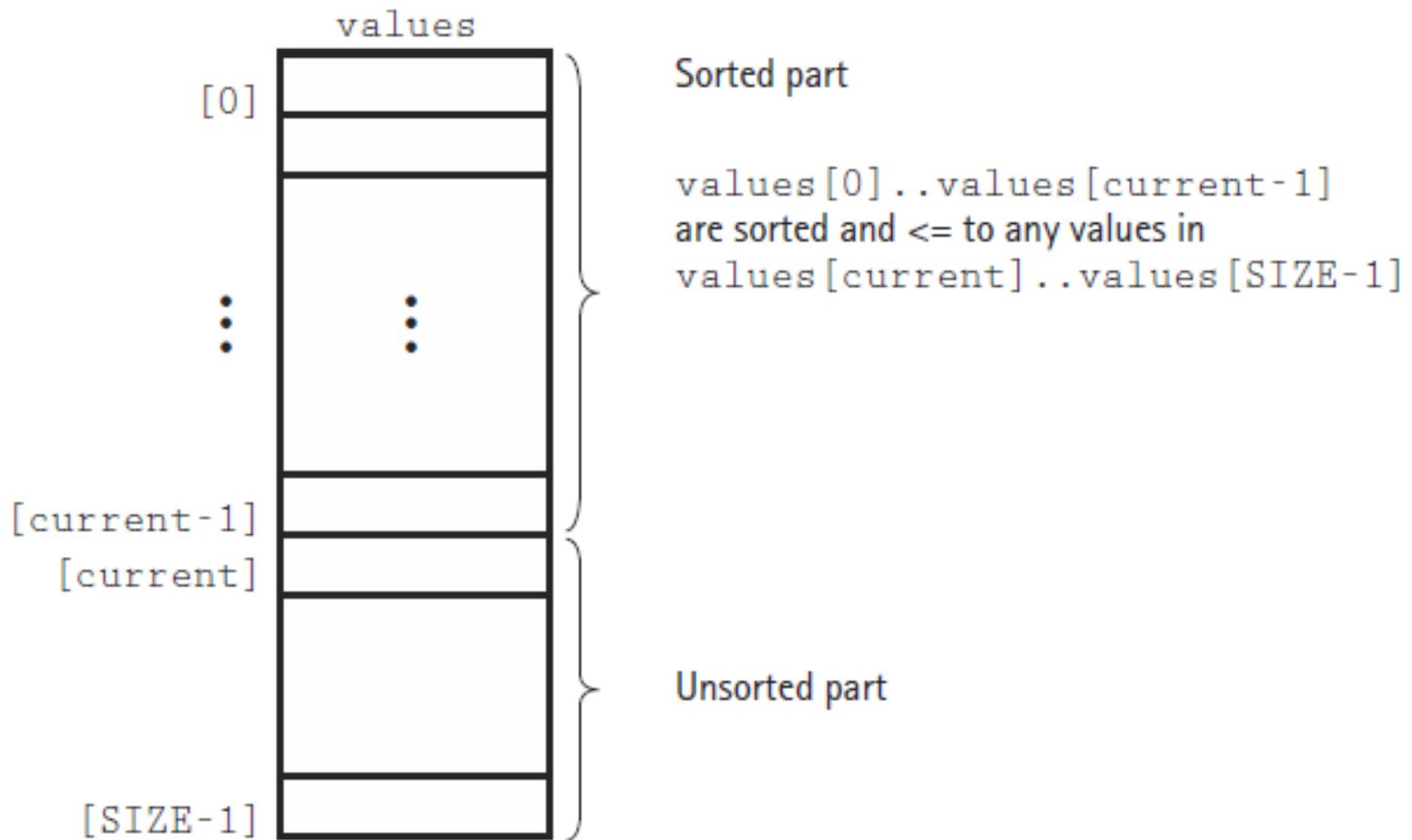
values
[0]
1
[1]
26
[2]
43
[3]
126
[4]
113

(d)

values
[0]
1
[1]
26
[2]
43
[3]
113
[4]
126

(e)

Selection Sort



Selection Sort

SelectionSort

Set current to the index of first item in the array
while more items in unsorted part of array

 Find the index of the smallest unsorted item

 Swap the current item with the smallest unsorted one

 Shrink the unsorted part of the array by incrementing current

Analysis $O(N^2)$

- The minimum finding is run $N-1$ times (for whatever size)
- Inside minimum finding function we have another loop that runs a varying number of times

- 1st time -> $N-1$ times
- 2nd time -> $N-2$ times
- 3rd time -> $N-3$ times
-
- $N-1$ th -> 1 times
- -> $N(N-1)/2$ times

Number of Items	Number of Comparisons
10	45
20	190
100	4,950
1,000	499,500
10,000	49,995,000

- How can we make it a descending sort algorithm?

Bubble Sort

BubbleSort

Set current to the index of first item in the array

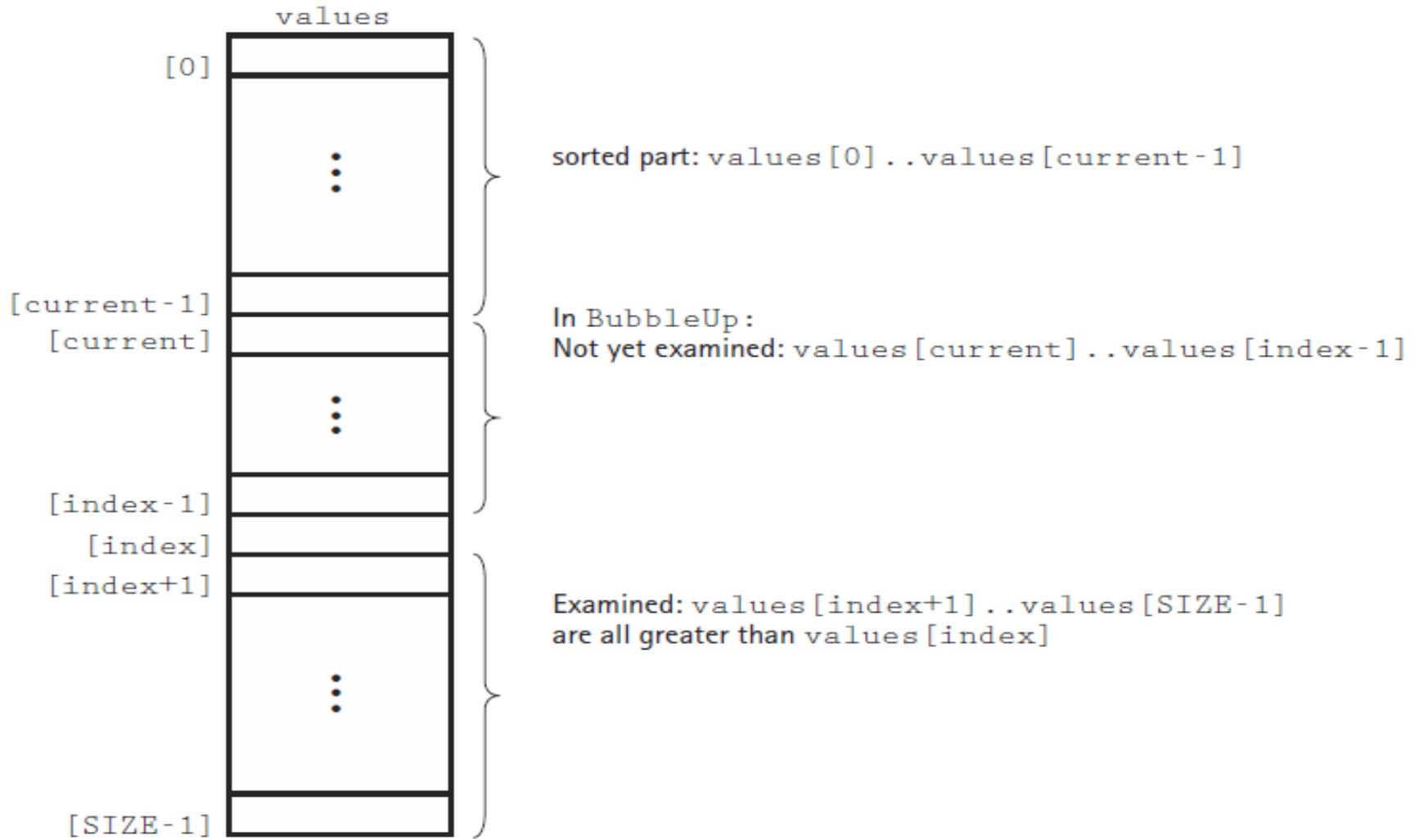
while more items in unsorted part of array

 "Bubble up" the smallest item in the unsorted part,

 causing intermediate swaps as needed

 Shrink the unsorted part of the array by incrementing current

Bubble Sort



Analysis $O(N^2)$

- The minimum finding is run $N-1$ times (for whatever size)
- Inside minimum finding function we have another loop that runs a varying number of times
 - 1st time $\Rightarrow N-1$ times
 - 2nd time $\Rightarrow N-2$ times
 - 3rd time $\Rightarrow n-3$ times
 -
 - $N-1$ th $\Rightarrow 1$ times
 - $\Rightarrow N(N-1)/2$ times
- Similar to Selection but with more overhead of swap with each iteration.
- How can we make Bubble sort shorter?

Short Bubble Sort Analysis

- Best-case Scenario: already sorted array, only one iteration
 - $N-1 \rightarrow O(N)$
- Worst-case Scenario: descendingly sorted array.
 - $N(N-1)/2 \rightarrow O(N^2)$
- Same as Selection but with more overhead (setting/resetting flag and swapping)
- Average-case Scenario:
 - $(N-1)+(N-2)+(N-3)+\dots+(N-K)$
 - K ranges from 1 to N-1
- Why use Short Bubble sort?

Merge Sort

- Previous algorithms are time consuming for very large number of items to sort. What about using “divide and conquer” idea
 - Break the array into 2 and sort each independently then merge them together => faster operation
- More complex algorithms : Called $O(N \log_2 N)$, (MergeSort, QuickSort, HeapSort).

Merge Sort

mergeSort—Recursive

Cut the array in half

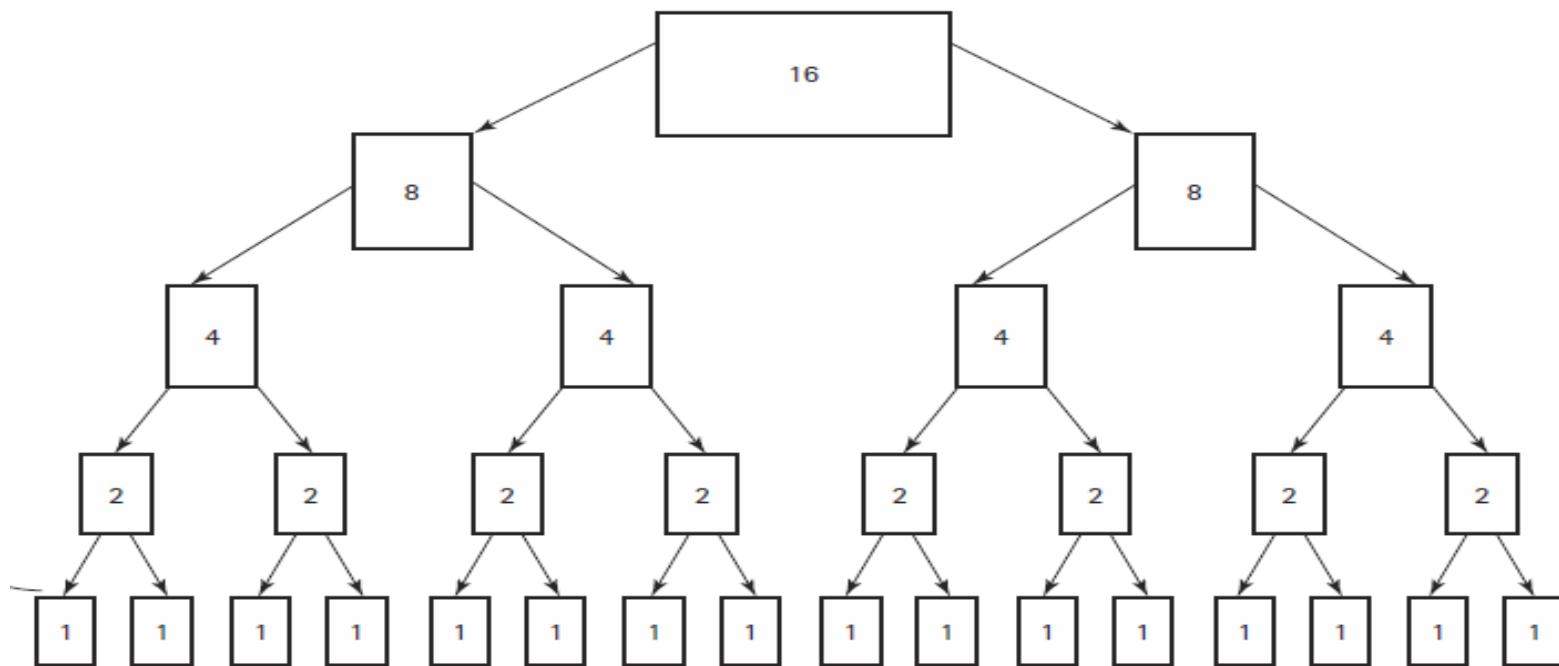
mergeSort the left half

mergeSort the right half

Merge the two sorted halves into one sorted array

Merge Sort

- What is the terminating condition of the MergeSort function?
 - If the array size is less than 2.



Merge Sort

merge (leftFirst, leftLast, rightFirst, rightLast)

(uses a local array, tempArray)

Set saveFirst to leftFirst // To know where to copy back
Set index to leftFirst

while more items in left half AND more items in right half

if values[leftFirst] < values[rightFirst]

 Set tempArray[index] to values[leftFirst]

 Increment leftFirst

else

 Set tempArray[index] to values[rightFirst]

 Increment rightFirst

 Increment index

Copy any remaining items from left half to tempArray

Copy any remaining items from right half to tempArray

Copy the sorted elements from tempArray back into values

Analysis $O(N \log_2 N)$

- Each Level needs $O(N)$ operations to perform merging.
 - How many levels we have? $\log_2 N$
- Main Disadvantage:
 - Stores a temp array of the same size as the array.

N	$\log_2 N$	N^2	$N \log_2 N$
32	5	1,024	160
64	6	4,096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

Lab #2

- Implement a function that sorts a given array of numbers
 - Use Merge Sort Algorithm
 - Function Prototype: *void ArraySort(uint8* array, uint8 arr_len);*

9. Code Portability

Outline

- The Course consists of the following topics:
 - Definitions
 - Portability
 - Techniques

Reusability and Portability

- The length of the development process is critical.
- No matter how high the quality of a software product, it will not sell if it takes 2 years to get it onto the market when a competitive product can be delivered in only one year.

Reusability and Portability

- Reusability
 - Using the components of one product to develop a different product.
- Portability
 - Easily modifying a product as a whole to run it on another hardware / system configuration rather than to recode it from scratch.

Why reuse?

- Minor Reasons

- It is expensive to design, implement, test, and document software
- Only 15% of new code serves an original purpose (average)
- Reuse of parts saves
- Design costs
- Implementation costs
- Testing costs
- Documentation costs

- Major Reasons

- Maintenance
- Maintenance consumes two-thirds of software cost

Portability

- The behavior of a software component should be specified independent of its implementation.
- Example: The hardware, operating system, or bus architecture can be changed without changing the behavior of the application components.
- Consequences:
 - Upgradability of hardware.
 - Portability of software.

Portability

- Reasons to make software portable
 - Commercial off-the shelf software
 - Recoup development costs with new clients that have different hardware/system configurations
 - Every 4 years or so, when the client organization purchases new hardware all of its software must be converted to run on the new hardware.
 - A product is considered portable if its significantly less expensive to adapt the product to run on new computer than to develop it from scratch.

Portability – Potential Incompatibilities

- Hardware Architecture (Instruction Sets)
- Data types (Integer may be represented as 16-bits or 32 bits)
- Operating Systems Compatibility
- Compilers

Techniques for Achieving Portability

- Isolate implementation-dependent pieces
- Use levels of abstraction
- High-level code artifacts
- Low-level code – written for the different types of hardware

Techniques for Achieving Portability

- Write product in high-level language (not assembly)
- Provide installation manual and lists of changes to be made
- Provide routines to handle communications between different SW layers

Lab #1

- Design a layered portable code of a simple embedded system that toggles a led upon button press.
- The design must be configurable to run over different chips with simple configuration

10. Code Optimization

Outline

- The Course consists of the following topics:

- Introduction
- Peephole Optimization
- Optimization Techniques
 - Eliminate Redundant Load/Store
 - Eliminate Dead Code
 - Flow of Control Optimization
 - Algebraic Simplification
 - Reduction in Strength
 - Instruction Selection
 - Code Motion Optimization
 - Recognize Sequence of Products
 - Make Use of Registers
 - Share Common Functions Optimization
 - Eliminate Unneeded Memory Refs

- Important Tools

Introduction

- A transformation to a program to make it run faster and/or take up less space
 - Optimization should be safe, preserve the meaning of a program.
 - Code optimization is an important component in a compiler
-
- Example: peephole optimization.
 - Peephole: a small moving window in the instruction sequence
 - Technique: examine a short sequence of target instructions (peephole) and try to replace it with a faster or shorter sequence

Introduction

- Must understand system to optimize performance
 - How programs are compiled and executed
 - How to measure program performance and identify bottlenecks
 - How to improve performance without destroying code modularity and generality

Introduction

- Performance highly depend on how code is written
 - Algorithms
 - Data representations
 - Loops
 -

Code Optimization Level

- Code optimization can either be high level or low level:
 - High level code optimizations:
 - Example: Loop unrolling
 - Low level code optimizations:
 - Example: Instruction selection
 - Some optimization can be done in both levels:
 - Example: Strength Reduction

Peephole Optimization

- Improvement of running time or space requirement of target program
- Can be applied to intermediate code or target code
- Peephole: is a small sliding window on a program
- Replace instructions in the peephole by faster/shorter sequence whenever possible
- May require repeated passes for best results

Peephole Optimization

- Peephole Optimization Examples:
 - Redundant instruction elimination
 - Flow of control optimization
 - Algebraic simplifications
 - Instruction selection

Eliminate Redundant Load/Store

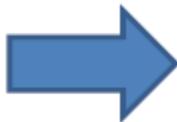
```
LD a, R0  
ST R0, a
```

Optimization is obvious
BUT

Store instruction must not have a label (why?)
-> the load and store must be in the same basic block

Eliminate Dead Code

```
if debug == 1 goto L1  
goto L2  
L1: print debugging information  
L2:
```



```
if debug != 1 goto L2  
print debugging information  
L2:
```

Flow of Control Optimization

```
goto L1  
...  
L1: goto L2
```



```
goto L2  
...  
L1: goto L2
```

```
if a < b goto L1  
...  
L1: goto L2
```



```
if a < b goto L2  
...  
L1: goto L2
```

```
goto L1  
...  
L1: if a < b goto L2  
L3:
```



```
if a < b goto L2  
goto L3  
...  
L3:
```

Algebraic Simplification

- Get rid of expressions like:
 - $X = X + 0$
 - $X = X * 1$

Reduction in Strength

- Reduction in strength

- $X^2 \rightarrow x * x$

- $X * 4 \rightarrow x << 2$

Instruction Selection

- Instruction selection
 - Sometimes some hardware instructions can implement certain operation efficiently.

Code Motion Optimization

- Code Motion
 - Reduce frequency with which computation performed
 - If it will always produce same result
 - Especially moving code out of loop

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[n*i + j] = b[j];
```



```
for (i = 0; i < n; i++) {  
    int ni = n*i;  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
}
```

Recognize Sequence of Products

- Recognize sequence of products

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[n*i + j] = b[j];
```



```
int ni = 0;  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++)  
        a[ni + j] = b[j];  
    ni += n;  
}
```

Make Use of Registers

- Reading and writing registers much faster than reading/writing memory
- Limitation
 - Compiler not always able to determine whether variable can be held in register
 - Possibility of Aliasing

Share Common Functions Optimization

- Share Common Sub expressions
 - Reuse portions of expressions
 - Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */  
up = val[(i-1)*n + j];  
down = val[(i+1)*n + j];  
left = val[i*n      + j-1];  
right = val[i*n      + j+1];  
sum = up + down + left + right;
```

3 multiplications: $i*n$, $(i-1)*n$, $(i+1)*n$

```
int inj = i*n + j;  
up = val[inj - n];  
down = val[inj + n];  
left = val[inj - 1];  
right = val[inj + 1];  
sum = up + down + left + right;
```

1 multiplication: $i*n$

```
leal -1(%edx),%ecx    # i-1  
imull %ebx,%ecx       # (i-1)*n  
leal 1(%edx),%eax     # i+1  
imull %ebx,%eax       # (i+1)*n  
imull %ebx,%edx       # i*n
```

Eliminate Unneeded Memory Refs

- Optimization
 - Don't need to store in destination until end
 - Local variable sum held in register

```
void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}
```

Important Tools

- Measurement
 - Accurately compute time taken by code
 - Most modern machines have built in cycle counters
 - Using them to get reliable measurements is tricky
 - Profile procedure calling frequencies
 - Unix tool gprof
- Observation
 - Generating assembly code
 - Lets you see what optimizations compiler can make
 - Understand capabilities/limitations of particular compiler

Lab #1

- Review your last lab code, and optimize the code following the optimization techniques introduced in this chapter.

11. Standard C Library

Outline

- The Course consists of the following topics:
 - Common Standard Library
 - assert
 - ctype
 - math
 - stdio
 - stdlib
 - time

Common Standard library Headers

- In the C Programming Language, the Standard Library Functions are divided into several header files. Below is the list of header files that we will cover:

■ <assert.h>	Diagnostics Functions
■ <ctype.h>	Character Handling Functions
■ <locale.h>	Localization Functions
■ <math.h>	Mathematics Functions
■ <setjmp.h>	Nonlocal Jump Functions
■ <signal.h>	Signal Handling Functions
■ <stdarg.h>	Variable Argument List Functions
■ <stdio.h>	Input/Output Functions
■ <stdlib.h>	General Utility Functions
■ <string.h>	String Functions
■ <time.h>	Date and Time Functions

assert()

- The C library macro void assert(int expression) allows diagnostic information to be written to the standard error file. In other words, it can be used to add diagnostics in your C program.
- expression - This can be a variable or any C expression. If expression evaluates to TRUE, assert() does nothing. If expression evaluates to FALSE, assert() displays an error message on stderr (standard error stream to display error messages and diagnostics) and aborts program execution.

assert() - Example

```
#include <assert.h>
#include <stdio.h>
int main()
{
    int a;
    char str[50];
    printf("Enter an integer value: ");
    scanf("%d", &a);
    assert(a >= 10);
    printf("Integer entered is %d\n", a);
    printf("Enter string: ");
```

```
scanf("%s", &str);
assert(str != NULL);
printf("String entered is: %s\n", str);
return(0);
}
```

Result:

```
Enter an integer value: 11
Integer entered is 11
Enter string: I love Amit
String entered is: I love Amit
```

ctype.h

- The ctype.h header file of the C Standard Library declares several functions that are useful for testing and mapping characters.
- All the functions accept int as a parameter, whose value must be EOF or representable as an unsigned char.
- All the functions return non-zero (true) if the argument c satisfies the condition described, and zero(false) if not.

ctype.h – Basic Functions

- int isalnum(int c)
 - This function checks whether the passed character is alphanumeric.
- int isalpha(int c)
 - This function checks whether the passed character is alphabetic.
- int isdigit(int c)
 - This function checks whether the passed character is decimal digit.
- int islower(int c)
 - This function checks whether the passed character is lowercase letter.

math.h

- The math.h header defines various mathematical functions and one macro. All the functions available in this library take double as an argument and return double as the result.
- The following example shows the usage of pow() function:

```
#include <stdio.h>
#include <math.h>
int main ()
{
    printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));
    printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

    return(0);
}
```

Result:

Value 8.0 ^ 3 = 512.000000

Value 3.05 ^ 1.98 = 9.097324

stdio.h

- The stdio.h header defines three variable types, several macros, and various functions for performing input and output.
- Let us compile and run that program that will create a file file.txt in the current directory which will have following content:

```
#include <stdio.h>

int main ()
{
    FILE *fp;
    int ch;
    fp = fopen("file.txt", "w");
    for( ch = 33 ; ch <= 100; ch++ )
    {
        putc(ch,fp);
    }
    fclose(fp);
    return(0);
}
```

stdio.h

- This program will read the content of the previously created file and print it :

```
#include <stdio.h>
int main ()
{
    FILE *fp;
    int c;
    fp = fopen("file.txt", "r");
    while(1)
    {
        c = fgetc(fp);
```

```
        if(feof(fp) )
        {
            break ;
        }
        printf("%c", c);
    }
    fclose(fp);
    return(0);
}
```

Result:

```
!"#$%&'()*+,-.
./0123456789:;<=>?@ABCDEFGHI
JKLMNOPQRSTUVWXYZ[\]^_`ab
cd
```

stdlib.h

- The stdlib.h header defines four variable types, several macros, and various functions for performing general functions., see the following sample functions
 - double atof(const char *str)
 - Converts the string pointed to, by the argument str to a floating-point number (type double).
 - int atoi(const char *str)
 - Converts the string pointed to, by the argument str to an integer (type int).
 - long int atol(const char *str)
 - Converts the string pointed to, by the argument str to a long integer (type long int).

stdlib.h

- double strtod(const char *str, char **endptr)
 - Converts the string pointed to, by the argument str to a floating-point number (type double).
- long int strtol(const char *str, char **endptr, int base)
 - Converts the string pointed to, by the argument str to a long integer (type long int).

stdlib.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    int val;
    char str[20];
    strcpy(str, "98993489");
    val = atoi(str);
    printf("String value = %s, Int value = %d\n",
str, val);
    strcpy(str, "AMIT");
    val = atoi(str);
    printf("String value = %s, Int value = %d\n",
str, val);
    return(0);
}
```

Result:

String value = 98993489, Int value = 98993489
String value = AMIT, Int value = 0

time.h

- The time.h header defines four variable types, two macro and various functions for manipulating date and time.

```
struct tm {  
    int tm_sec;      /* seconds, range 0 to 59 */  
    int tm_min;      /* minutes, range 0 to 59 */  
    int tm_hour;     /* hours, range 0 to 23 */  
    int tm_mday;     /* day of the month, range 1 to 31 */  
    int tm_mon;      /* month, range 0 to 11 */  
    int tm_year;     /* The number of years since 1900 */  
    int tm_wday;     /* day of the week, range 0 to 6 */  
    int tm_yday;     /* day in the year, range 0 to 365 */  
    int tm_isdst;    /* daylight saving time */  
};
```

localtime()

- The C library function `struct tm *localtime(const time_t *timer)` uses the time pointed by timer to fill a tm structure with the values that represent the corresponding local time. The value of timer is broken up into the structure tm and expressed in the local time zone.

localtime()

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

int main ()
{
    time_t rawtime;
    struct tm *info;
    char buffer[80];
    time( &rawtime );
    info = localtime( &rawtime );
    printf("Current local time and date: %s", asctime(info));
    return(0);
}
```

Result:

Current local time and date: Fri Sept 11 01:46:05 2015

Lab #1

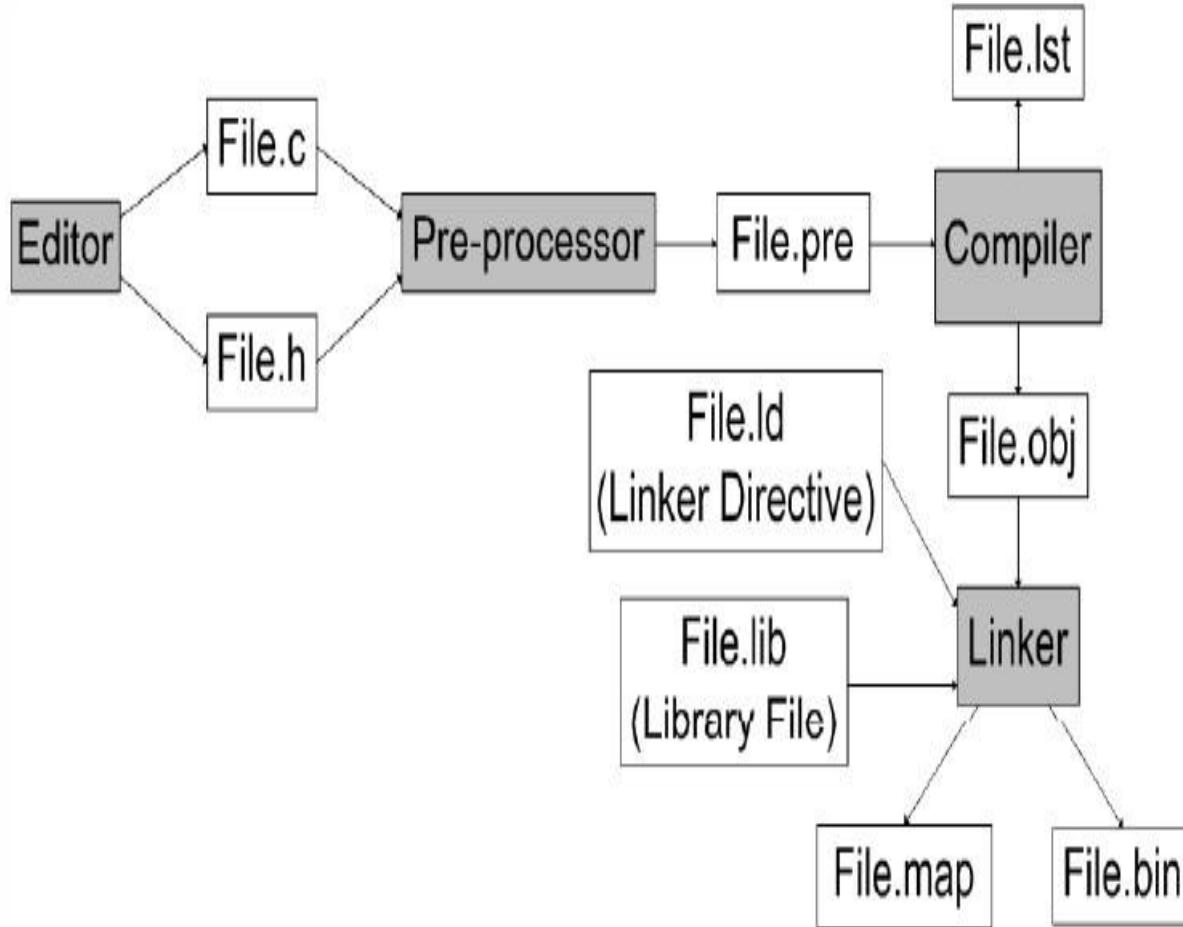
- Implement a program that uses C Standard Library Functions, that reads an input integer from the user, and asserts if its value is more a predefined threshold
- Try different scenarios to test the assert behavior

12. Building Process

Outline

- The Course consists of the following topics:
 - Building Overview
 - Preprocessor
 - Compiler
 - Assembler
 - Linker File
 - Linker
 - BIN, ELF, MAP Files
 - Building Scenario
 - Microcontroller Memory Segments
 - Memory Allocation Scenario
 - Makefile
 - Native and Cross Compilers

Building Overview



Preprocessor

- The input to this phase is the .c File and .h Files
- The preprocess process the preprocessor keywords like #define, #ifdef, #include, etc. and generate a new .pre file or .i file after the
- It is a text replacement process.
- The output of this phase is a C Code without any preprocessor keyword.

Preprocessor

```
#define JAN      My_defs.h  
#define FEB  
#define MAR  
#define PI  
1  
2  
3  
3.1416  
double my_global;
```

My_prog.i

```
#include "my_defs.h"  
Int main()  
{  
    double angle=2*PI;  
  
    printf("%s",month[FEB]);  
  
}
```

My_prog.c

```
#define JAN  
#define FEB  
#define MAR  
#define PI  
1  
2  
3  
3.1416  
double my_global;  
Int main()  
{  
    double angle=2*3.1416;  
    printf("%os",month[2]);  
}
```

Compiler

- The input to this phase C Files without any „#“ statement.
- The compiler parse the code, and check the syntax correctness of the file.
- Convert the C Code into optimized machine language code.
- The output of this phase is object file .o file or .obj file and list file (.lst file or .lss file).
- List File: Contains the corresponding assembly code for each line.

Assembler

- The input to this phase .asm File
- The Assembler converts the assembly code to the corresponding machine language code.
- The output of this phase is object file .o file or .obj file.

Linker File

- The input file to the linker and it contains the memory mapping information of the segments in the microcontroller memory.
- So using the linker file you could place any segment in any place in the memory.
- #pragma is used to change the default segment for specific variable or code

Linker

- The input to this phase multiple .obj Files.
- The Linker merges different object files and library files.
- Linker is used to be sure that every .obj files or .o files have all the external data and functions that is defined.
- And allocate target memory (RAM,ROM and Stack) to give addresses to the variables and function according to .ld file(linker file) and generate .map file and .bin File.
- The output of this phase is the .bin file and the .map file.

What Does a Linker Do?

- Merges object files
- Resolves external references
- Relocates symbols

Bin File

- The bin File which burn in target .
- It may has another names (.hex , .out , .elf , .srec "Motorola").

Executable and Linkable Format (ELF)

- Standard binary format for object files
- Derives from AT&T System V Unix
 - Later adopted by BSD Unix variants and Linux
- One unified format for
 - Relocatable object files (.o),
 - Executable object files
 - Shared object files (.so)
 - Generic name: ELF binaries
- Better support for shared libraries than old a.out formats.

MAP FILE

- The map file is an additional output file that contains information about the location of sections and symbols. You can customize the type of information that should be included in the map file.

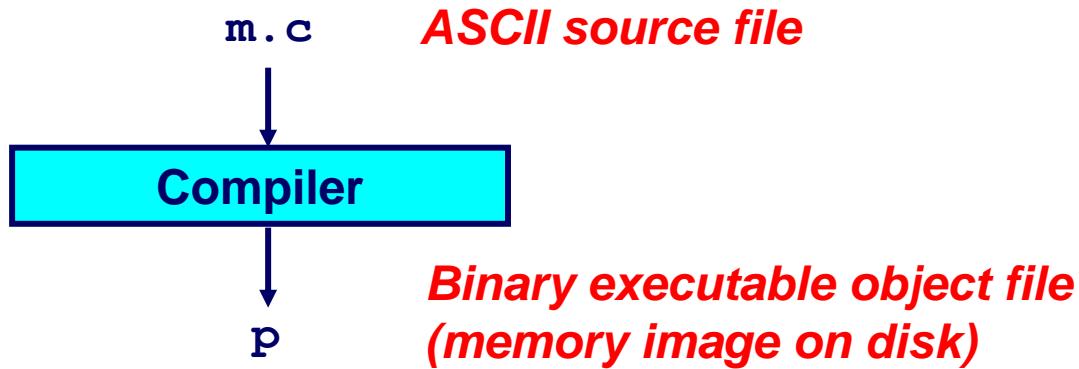
MAP FILE

MEMORY MAP OF MODULE: Test (?C_STARTUP)

START	STOP	LENGTH	TYPE	RTYP	ALIGN	TGR	GRP	COMB	CLASS	SECTION	NAME
<hr/>											
000000H	000003H	000004H	---	---	---	---	---	---	* INTVECTOR_TABLE *		
000004H	00000DH	00000AH	XDATA	REL	WORD	---	---	GLOB	---	?C_INITSEC	
00000EH	00001DH	000010H	CONST	ABS	WORD	---	---	PRIV	---	?C_CLRMEMSEC	
00001EH	00005EH	000041H	DATA	REL	BYTE	---	---	PUBL	FCONST	?FC?TEST	
000060H	00009FH	000040H	DATA	REL	WORD	---	---	PUBL	FCONST	?FC??PRNFMT	
0000A0H	0001CDH	00012EH	CODE	REL	WORD	---	---	PRIV	ICODE	?C_STARTUP_CODE	
0001CEH	00065BH	00048EH	CODE	REL	WORD	---	2	PRIV	NCODE	?PR?SCANF	
00065CH	00098FH	000334H	CODE	REL	WORD	---	2	PUBL	NCODE	?C_LIB_CODE	
000990H	000A25H	000096H	CODE	REL	WORD	---	2	PUBL	NCODE	?PR?TEST	
000A26H	000A57H	000032H	CODE	REL	WORD	---	2	PRIV	NCODE	?PR?PUTCHAR	
000A58H	000A85H	00002EH	CODE	REL	WORD	---	2	PUBL	NCODE	?PR?GETCHAR	
000A86H	000A9FH	00001AH	CODE	REL	WORD	---	2	PUBL	NCODE	?PR?ISSPACE	
000AA0H	000AABH	00000CH	CODE	REL	WORD	---	2	PUBL	NCODE	?PR?GETKEY	
000AACB	000AB3H	000008H	CODE	REL	WORD	---	2	PUBL	NCODE	?PR?UNGET	
008000H	008FFFFH	001000H	DATA	REL	WORD	---	1	PUBL	NDATA	?C_USERSTACK	
009000H	009003H	000004H	DATA	REL	WORD	---	1	PUBL	NDATA0	?ND0?TEST	
009004H	009004H	000001H	DATA	REL	BYTE	---	1	PUBL	NDATA0	?ND0?GETCHAR	
009006H	009055H	000050H	DATA	REL	WORD	---	---	PUBL	FDATA0	?FD0?TEST	
00FA00H	00FBFFFH	000200H	---	---	---	---	---	---	* SYSTEM_STACK *		
00FC00H	00FC1FH	000020H	DATA	---	BYTE	---	---	---	*REG*	?C_MAINREGISTERS	

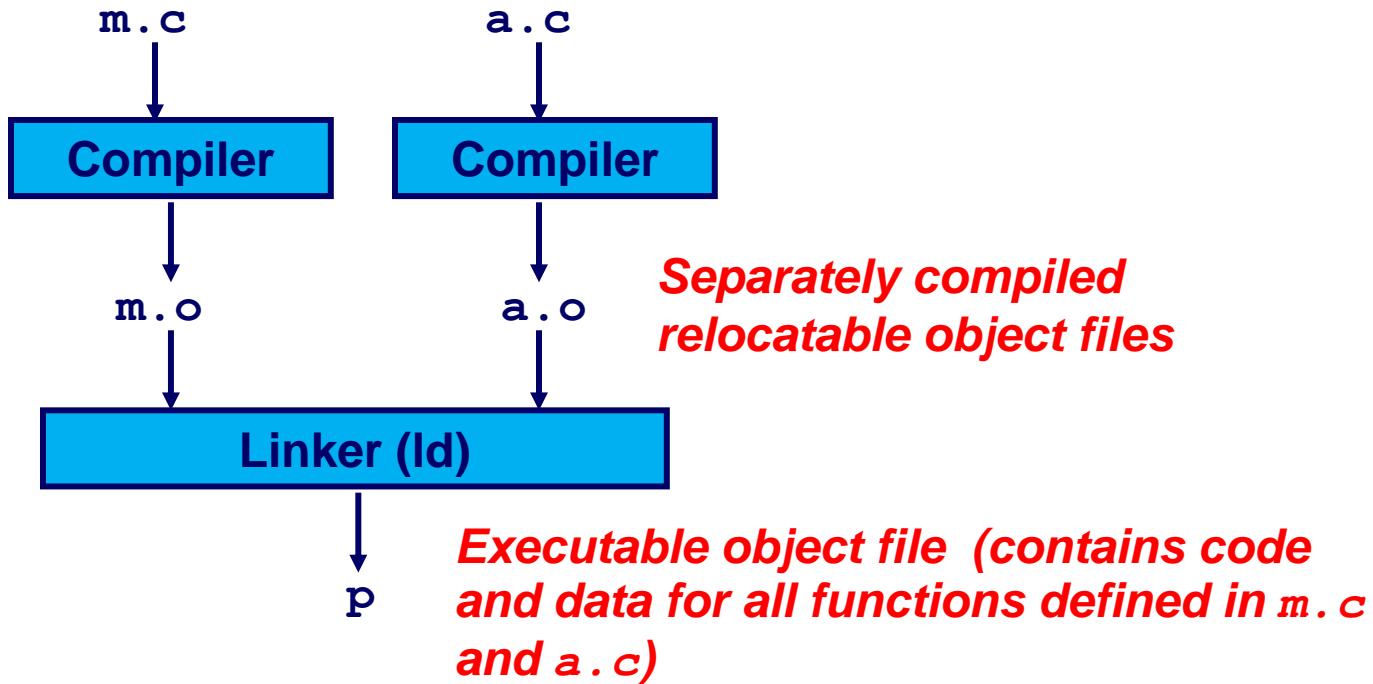
Building Scenario

- A Simplistic Program Translation Scheme



Building Scenario

- A Better Scheme Using a Linker



Microcontrollers Memory Segments

- The computer program memory is organized into the following:
 - Data Segment (.data + .bss + Heap + Stack)
 - Constant Segment (.rodata)
 - Code segment (.text)



**Data Memory
SRAM**

**Program Memory
Flash EEPROM**

Microcontrollers Memory Segments

- **.data Segment**
 - Holds initialized variables
- **.bss Segment**
 - Holds uninitialized variables
- **.stack Segment**
 - Holds the stack
- **.rodata Segment**
 - Holds the constant data
- **.text Segment**
 - Holds the program code

Example C Program

m.c

```
int e=7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

```
extern int e;

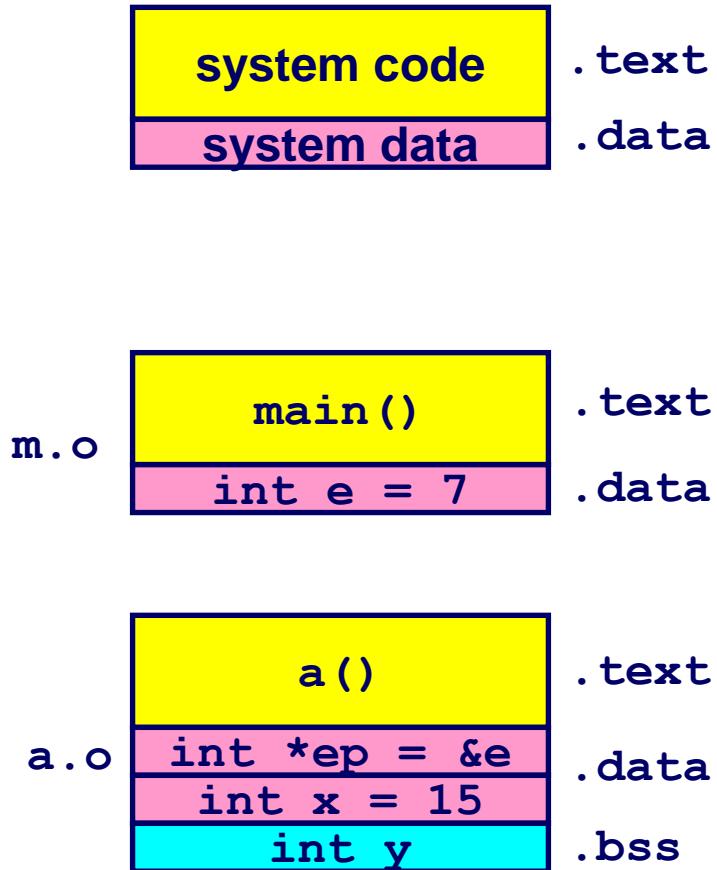
int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}
```

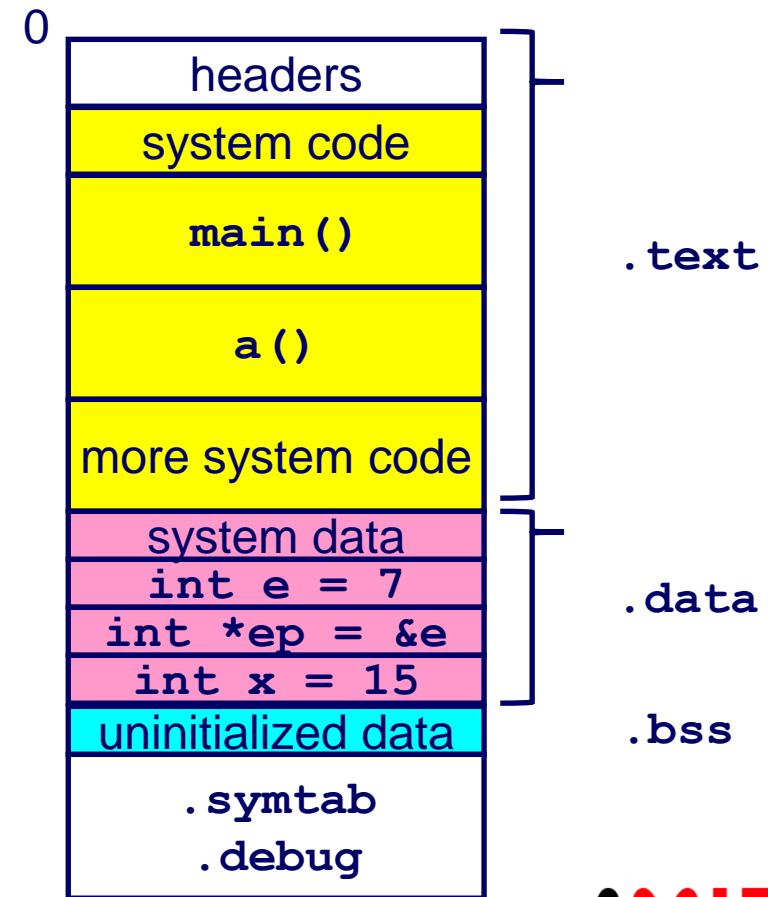
Merging Object Files

- Merging Re-locatable Object Files into an Executable Object File

Relocatable Object Files

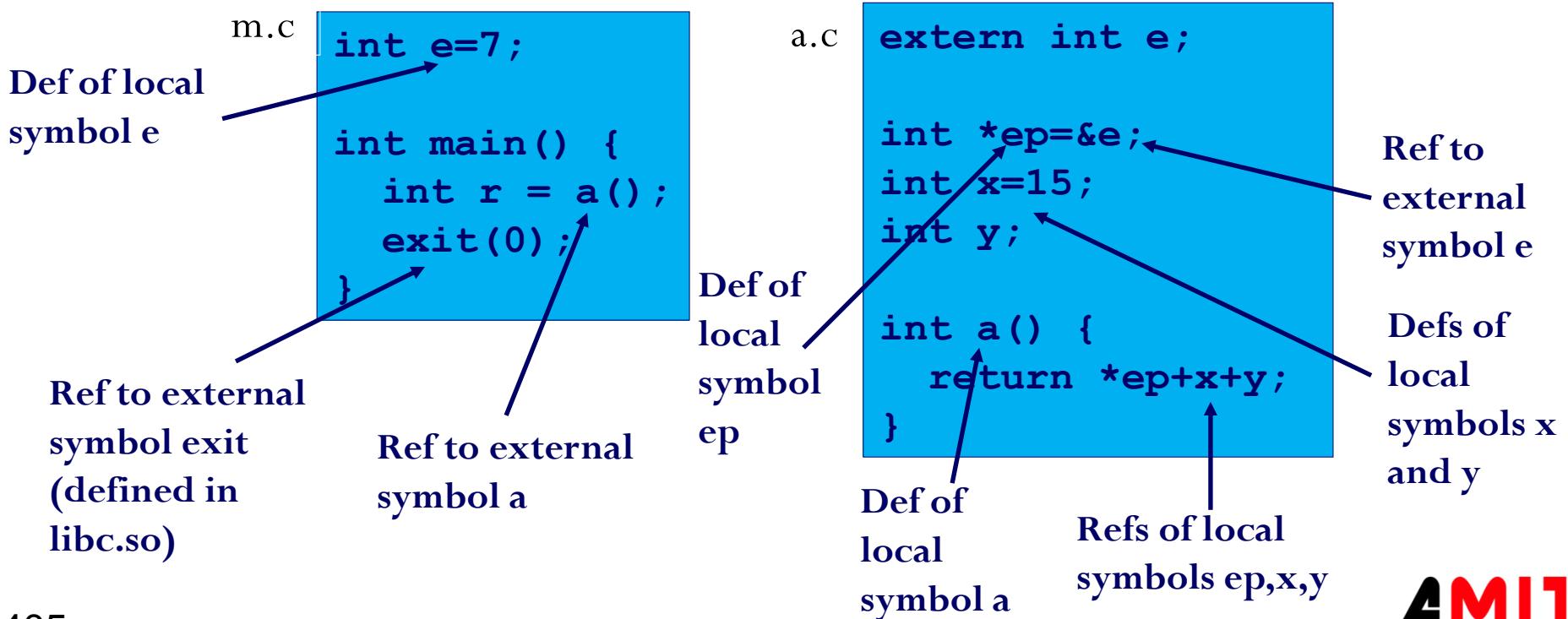


Executable Object File



Relocation

- Relocating Symbols and Resolving External References
 - Symbols are lexical entities that name functions and variables.
 - Each symbol has a value (typically a memory address).
 - Code consists of symbol definitions and references.
 - References can be either local or external.



m.o Relocation Info

m.c

```
int e=7;

int main() {
    int r = a();
    exit(0);
}
```

Disassembly of section .text:

```
00000000 <main>: 00000000 <main>:
 0: 55                      pushl  %ebp
 1: 89 e5                   movl   %esp,%ebp
 3: e8 fc ff ff ff         call   4 <main+0x4>
 4: R_386_PC32    a
 8: 6a 00                   pushl  $0x0
 a: e8 fc ff ff ff         call   b <main+0xb>
 b: R_386_PC32    exit
 f: 90                      nop
```

Disassembly of section .data:

```
00000000 <e>:
 0: 07 00 00 00
```

a.o Relocation Info (.text)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}
```

Disassembly of section .text:

00000000 <a>:

0:	55	pushl %ebp
1:	8b 15 00 00 00	movl 0x0,%edx
6:	00	
7:	a1 00 00 00 00	movl 0x0,%eax
c:	89 e5	movl %esp,%ebp
e:	03 02	addl (%edx),%eax
10:	89 ec	movl %ebp,%esp
12:	03 05 00 00 00	addl 0x0,%eax
17:	00	
18:	5d	popl %ebp
19:	c3	ret

3: R_386_32 ep

8: R_386_32 x

14: R_386_32 y

a.o Relocation Info (.data)

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}
```

Disassembly of section .data:

00000000 <ep>:

0: 00 00 00 00

0: R_386_32 e

00000004 <x>:

4: 0f 00 00 00

Executable (.text)

- Executable After Relocation and External Reference Resolution
(.text)

```
08048530 <main>:  
 8048530: 55          pushl  %ebp  
 8048531: 89 e5        movl   %esp,%ebp  
 8048533: e8 08 00 00 00 call   8048540 <a>  
 8048538: 6a 00        pushl  $0x0  
 804853a: e8 35 ff ff ff call   8048474 <_init+0x94>  
 804853f: 90          nop  
  
08048540 <a>:  
 8048540: 55          pushl  %ebp  
 8048541: 8b 15 1c a0 04 movl   0x804a01c,%edx  
 8048546: 08            
 8048547: a1 20 a0 04 08 movl   0x804a020,%eax  
 804854c: 89 e5        movl   %esp,%ebp  
 804854e: 03 02        addl   (%edx),%eax  
 8048550: 89 ec        movl   %ebp,%esp  
 8048552: 03 05 d0 a3 04 addl   0x804a3d0,%eax  
 8048557: 08            
 8048558: 5d          popl   %ebp  
 8048559: c3          ret
```

Executable (.data)

- Executable After Relocation and External Reference Resolution(.data)

m.c

```
int e=7;

int main() {
    int r = a();
    exit(0);
}
```

a.c

```
extern int e;

int *ep=&e;
int x=15;
int y;

int a() {
    return *ep+x+y;
}
```

Disassembly of section .data:

```
0804a018 <e>:
804a018: 07 00 00 00

0804a01c <ep>:
804a01c: 18 a0 04 08

0804a020 <x>:
804a020: 0f 00 00 00
```

Makefile

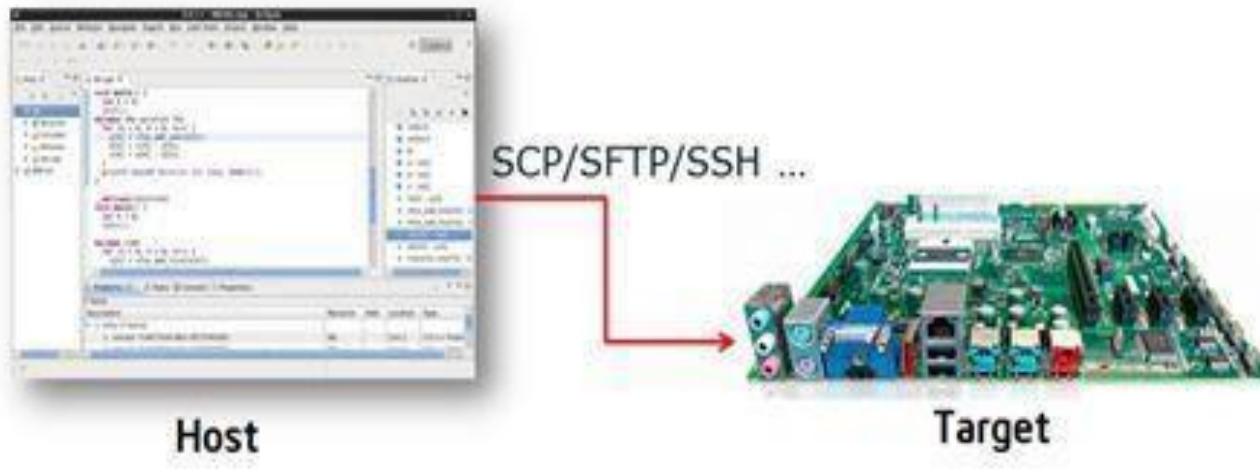
- You can imagine how tedious the build process could be if you had a large number of source code files for a particular project.
- Manually entering individual compiler and linker commands on the command line becomes tiresome very quickly.
- In order to avoid this, a makefile can be used.

Makefile

- A makefile is a script that tells the make utility how to build a particular program. (The make utility is typically installed with the other GNU tools.)
- The make utility follows the rules in the makefile in order to automatically generate output files from a set of input source files.
- Makefiles might be a bit of a pain to set up, but they can be a great timesaver and a very powerful tool when building project files over and over (and over) again.

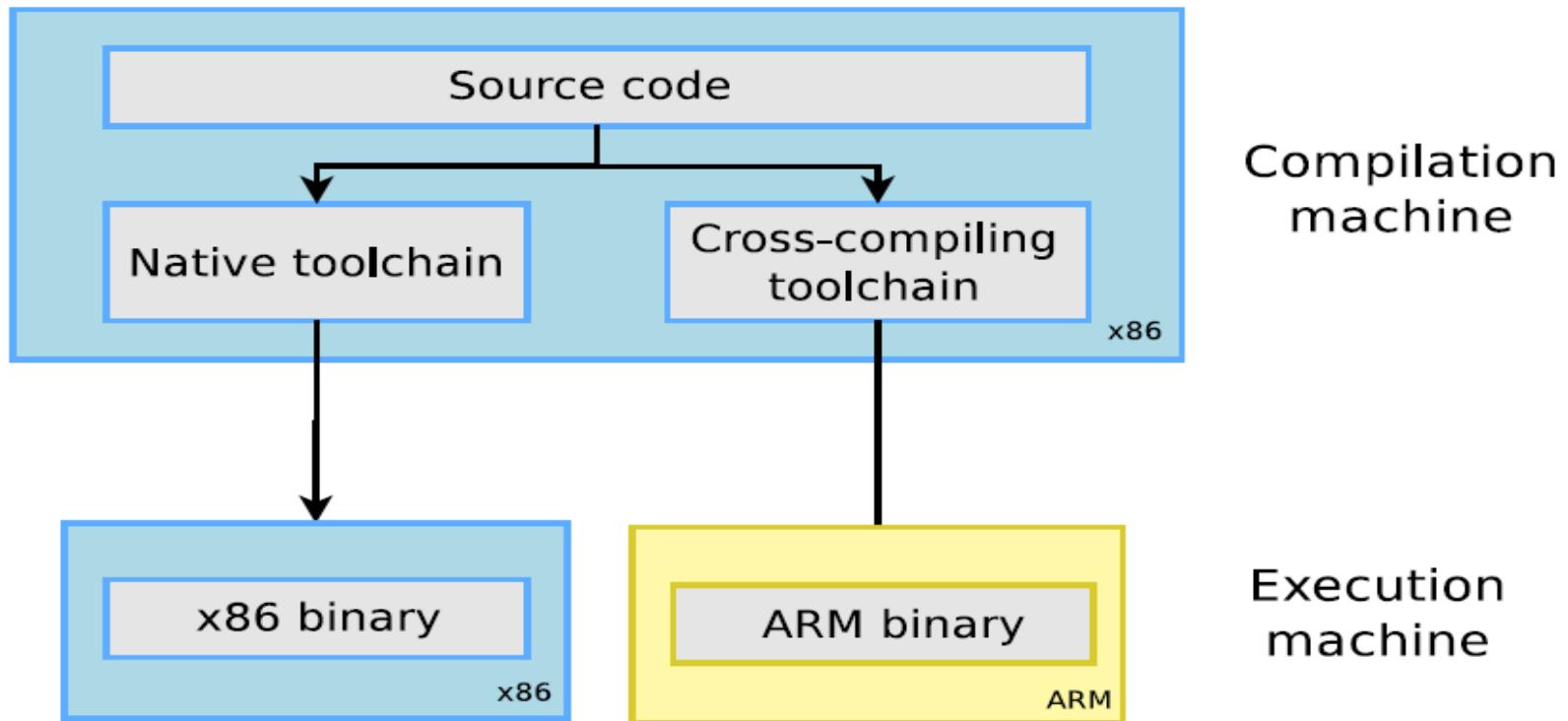
Native and Cross Compilers

- Regular C programming needs native compiler runs on your workstation or your PC and generates code for your workstation, usually x86 like Intel and AMD.
- Embedded C programming needs Cross Compiler that runs on the development machine, but generates code for the target.



Native and Cross Compilers

- Development Environment is different from target environment.
- Need for cross platform development and debugging tools.



Lab #1

- Using your last lab project, list the following:
 - Memory taken by Program Code
 - Memory taken by Initialized Variables
 - Memory taken by Uninitialized Variables
 - Memory taken by Constants

13. Guidelines for Writing Good Code

Outline

- The Course consists of the following topics:
 - Steps to write good code
 - Lesson Learned recommendations
 - MISRA
 - Selected MISRA Rules

Steps to Write Good Code

- Writing good code can be achieved by following:
 - Follow Expert's Lesson Learned Recommendations
 - Follow Coding Style Standards, e.g. MISRA
 - Create a Modular Design, and Layered Architecture
 - Write a Portable Code
 - Use Proper Naming Convention

Lesson Learned Recommendations

- Use descriptive variable names
- Use lots of parentheses and/or spaces in expressions
- Use shotgun initialization
 - If you don't know the correct initial value for a variable when you declare it then don't initialize it to some arbitrary value like zero
- Don't use magic numbers
- Use typedef's
- Define your own integer types
- Use the exact type that you require
- Place all local declarations at the start of functions
- Never use goto
- Always return an error status

C MISRA Definition

- It stands for Motor Industries Software Reliability Association.
- MISRA guidelines are thoroughly followed especially in automotive industry, as they represent one of the most popular standards for developing secure software.
- It is a software development standard for the C programming language. Its aims are to facilitate code portability and reliability in the context of embedded systems.
- The C guidelines are intended to be applied during the development of
- software used in safety-critical applications.

C MISRA Definition

- The rules can be divided logically into a number of categories:
 - Avoiding possible compiler differences, for example, the size of a C integer may vary but an INT16 is always 16 bits. (C99 standardized on int16_t.)
 - Avoiding using functions and constructs that are prone to failure, for example, malloc may fail.
 - Produce maintainable and debuggable code, for example, naming conventions and commenting.
 - Best practice rules.
 - Complexity limits.

C MISRA Definition

- There are different versions of the MISRA C guidelines:
 - MISRA-C:1998 - Guidelines for the use of the C language in vehicle based software - 127 rules (93 required, 34 advisory)
 - MISRA-C:2004 - Guidelines for the use of the C language in critical systems - 141 rules (121 required, 20 advisory)
 - MISRA-C:2012 - Extends support to the C99 version of the C language (while maintaining guidelines for C90), in addition to including a number of improvements that can reduce the cost and complexity of compliance, whilst aiding consistent, safe use of C in critical systems. 143 rules (each of which is checkable using static program analysis) and 16 "directives"

C MISRA Definition

- Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis).
- MISRA is static analysis tool; it works on source code.
- Furthermore, a significant number of tools are available to check your code against MISRA rules. But for sad MISRA checking tools are pretty expensive; Try understand C evaluation version.

C MISRA Definition

- Rules are labeled required or advisory.
- Rules are grouped in sub groups:
 - Environment
 - Character sets
 - Comments
 - Identifiers
 - Types
 - Constants
 - Declarations and definitions
 - Initialization
 - Operators
 - Conversions
 - Expressions
 - Control flow
 - Functions
 - Preprocessing directives
 - Pointers-and-arrays
 - Structures-and-unions
 - Standard libraries

MISRA Rules

- Rule #1 (Required)
 - All code shall conform to ISO 9899 standard C, with no extensions permitted.
- Rule #3 (Advisory)
 - Assembly language shall be encapsulated and isolated

MISRA Rules

- Rule #12 (Advisory)
 - No identifier in one name space shall have the same spelling as an identifier in another name space.

```
sturct student
{
int id; int age;
char * student; //member has the same
structure name
}
```

MISRA Rules

- Rule #13 (Advisory)
 - The basic types of char, int, short, long, float and double should not be used, but specific-length equivalents should be “typedef” for the specific compiler, and these type names used in the code.
 - The storage length of types can vary from compiler to compiler. It is safer if programmers work with types which they know to be of a given length. For example sint16 might be the typedef chosen for a signed 16 bit integer. The known-length substitute types should be defined in a header file which can then be included in all code files.
 - Example :
 - `typedef signed short sint16;`

MISRA Rules

- Rule #14 (Required)
 - The type char shall always be declared as unsigned char or signed char.
- Rule #17 (Required)
 - typedef names shall not be reused.

MISRA Rules

- Rule #19 (Required)
 - Octal constants (other than zero) shall not be used.
 - Any integer constant beginning with a „0“ (zero) is treated as octal.
 - So there is a danger, for example, with writing fixed length constants. For example, the following array initialization for 3- digit bus messages would not do as expected (052 is octal, i.e. 42 decimal):

code[1] = 109; / set to decimal 109 */*

code[2] = 100; / set to decimal 100 */*

code[3] = 052; / set to decimal 42 */*

code[4] = 071; / set to decimal 57 */*

MISRA Rules

- Rule #21 (Required)
 - Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
 - Hiding identifiers with an identifier of the same name in a nested scope leads to code which is very confusing. For example:

sint16 i;

{

sint16 i; / This is a different variable */*

i = 3; / It could be confusing as to which i this refers */*

}

MISRA Rules

- Rule #22 (Advisory)
 - Declarations of objects should be at function scope unless a wider scope is necessary.
- Rule #23 (Advisory)
 - All declarations at file scope should be static where possible.
- Rule #27 (Advisory)
 - External objects should not be declared in more than one file.

MISRA Rules

- Rule #31 (Required)
 - Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

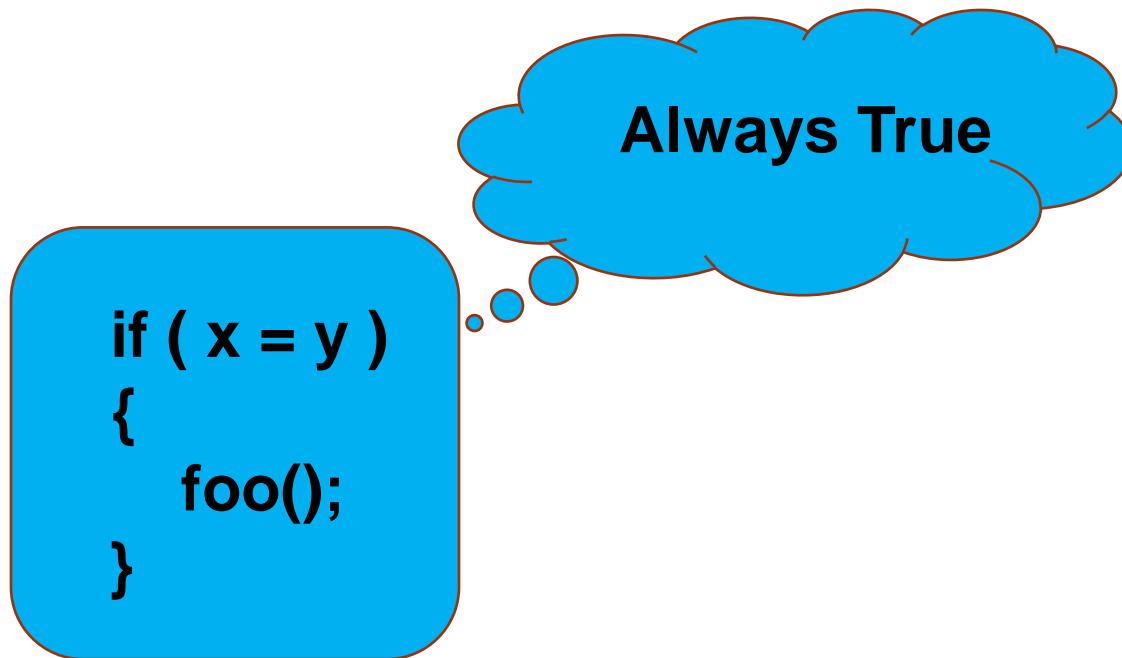
```
sint16 y[3][2] = { 1, 2, 3, 4, 5, 6 }; /* incorrect */
```

```
sint16 y[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } }; /* correct */
```

- Rule #32 (Required)
 - In an enumerator list, the „=” operator shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

MISRA Rules

- Rule #35 (Required)
 - Assignment operators shall not be used in Boolean expressions.



MISRA Rules

- Rule #37 (Required)
 - Bitwise operations shall not be performed on signed integer types.
- Rule #42 (Required)
 - The comma operator shall not be used, except in the control expression of a for loop.

MISRA Rules

```
int x = bool_value ?  
printf("Yes"), 5 : 117;
```

If the condition
is true “Yes” will
be printed and
“x” will equal “5”

```
int func(void)  
{  
int x=3,y=2; x++;  
return x,y;  
}
```

This function
will return the
value of “y”

MISRA Rules

```
int i, j;
for (i=-1,j=0;i=i+1,i<5;j=i+1,printf("%i\n", j))
{
    printf("%i\t", i);
}
```

```
int i = -1;
int j = 0;
i = i + 1;

while(i<5) {
    printf("%i\t", i);
    j = i + 1;
    printf("%i\n", j);
    i = i + 1;
}
```

MISRA Rules

- Rule #43 (Required)
 - Implicit conversions which may result in a loss of information shall not be used.

```
Short num = 0x01FF;  
char var1 = num; /* incorrect */  
char var2 = (char)num; /* correct */
```

- Rule #45 (Required)
 - Type casting from any type to or from pointers shall not be used.
- Rule #49 (Advisory)
 - Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

```
if( x != 0 ) /* Correct way of testing x is non-zero */  
if( x ) /* Incorrect, unless x is effectively Boolean data (e.g. a flag)  
*/
```

MISRA Rules

- Rule #50(Required)
 - Floating point variables shall not be tested for exact equality or inequality.

```
float x = 9.3f/ 3.0f;  
if( x == 3.1F) {  
    printf("yes!\n");  
} else {  
    printf("OMG!\n");  
}
```

Results

OMG!

MISRA Rules

- Rule #52 (Required)
 - There shall be no unreachable code.
- Rule #55 (Advisory)
 - Labels should not be used, except in switch statements.
- Rule 56 (required)
 - The goto statement shall not be used.
- Rule 57 (required)
 - The continue statement shall not be used.
- Rule 58 (required)
 - The break statement shall not be used (except to terminate the cases of a switch statement).

MISRA Rules

- Rule #59 (Required)
 - The statements forming the body of an if, else if, else, while, do ... while or for statement shall always be enclosed in braces.
- Rule #60 (Advisory)
 - All if, else if constructs should contain a final else clause.

MISRA Rules

- Rule 61 (Required)
 - Every non-empty case clause in a switch statement shall be terminated with a break statement.
- Rule 62 (Required)
 - All switch statements should contain a final default clause.
- Rule 63 (Advisory)
 - A switch expression should not represent a Boolean value.
 - If the expression in the switch statement is effectively representing Boolean data, then in effect it can only take two values, and an if, else construct is a better way of representing the two-way choice. Thus expressions of this nature should not be used in switch statements.
- Rule 64 (Required)
 - Every switch statement shall have at least one case.

MISRA Rules

- Rule #70 (Required)
 - Functions shall not call themselves, either directly or indirectly.
- Rule #81 (Advisory)
 - const qualification should be used on function parameters which are passed by reference, where it is intended that the function will not modify the parameter.
- Rule #81 (Advisory)
 - A function should have a single point of exit.
- Rule #87 (Required)
 - #include statements in a file shall only be preceded by other preprocessor directives or comments.

MISRA Rules

- Rule #90 (Required)
 - C macros shall only be used for symbolic constants, function-like macros, type qualifiers and storage class specifiers.

```
/* The following are allowed */
#define PI 3.14159f /* Constant */

#define PLUS2(X) ((X) + 2)      /* Function-like macro */
#define STOR extern             /* storage class specifier */

/* The following are NOT allowed */
#define SI_32 long             /* use typedef instead */
#define STARTIF if(              /* very bad */
```

MISRA Rules

- Rule 91 (Required)
 - Macros shall not be #define and #undef within a block.
- Rule 92 (Advisory)
 - #undef should not be used.
- Rule #96 (Required)
 - In the definition of a function-like macro the whole definition, and each instance of a parameter, shall be enclosed in parentheses.

```
#define MUL(x,y) ((x)*(y))
```

MISRA Rules

- Rule 101 (Advisory)
 - Pointer arithmetic should not be used.
- Rule 102 (Advisory)
 - No more than 2 levels of pointer indirection should be used.
- Rule 105 (Required)
 - All the functions pointed to by a single pointer to function shall be identical in the number and type of parameters and the return type.
- Rule 106 (Required)
 - The address of an object with automatic storage shall not be assigned to an object which may persist after the object has ceased to exist.
- Rule #118 (Required)
 - Dynamic heap memory allocation shall not be used.

Lab #1

- Select a previous lab project, and do the following:
 - Check your project against “Lesson Learned Recommendations”
 - Fix the raised points

Thank You

