C Programming Essentials
Unit 1: Sequential Programming

CHAPTER 2: ELEMENTARY PROGRAMMING

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

Tokens in C Language

Language Features

Basic Built-In Types
int
char
float
double
bool

Derived Types
struct
enum
array
union

Qualifiers
const
volatire

Type Modifiers
short (short int)
long (long int)
signed *(by default)*
unsigned (unsigned int)

Assignment
=

Punctuation	
{	{
(	)
,	;

Pointers
*
& (reference)

Memory Types
code
stack
heap

Flow Control	
if	else
for	while

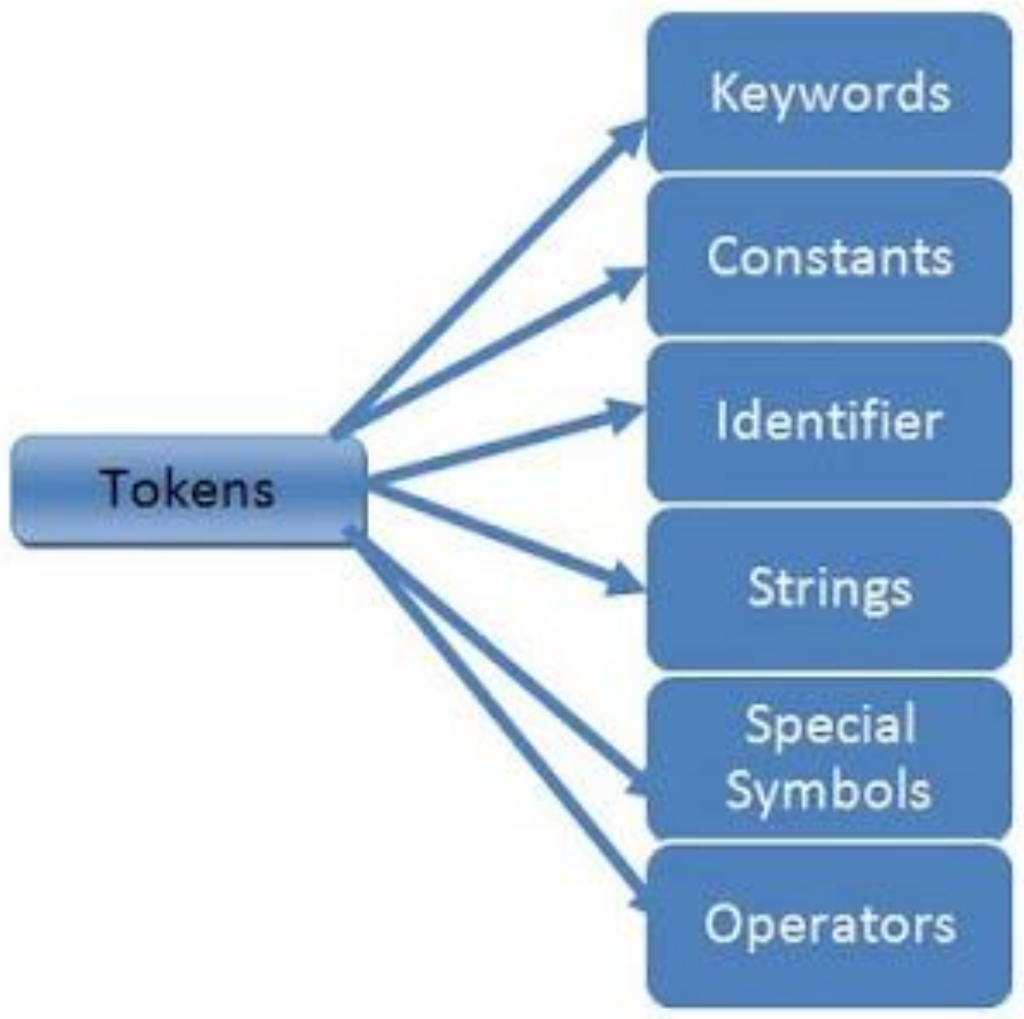
Memory Management
malloc()
free()
realloc()
sizeof()

Operators			
+	-	*	/
%	++	--	>
<	<=	>=	==
!=	>>	<<	

#include <stdio.h>
printf()
scanf()

Calling Convention
cdecl
stdcall
fastcall

Pre-processor Macros
#include
#define
#ifdef
#endif



Ingredients in C Language

- Keywords
- Identifiers (variables, functions)
- Constants and Literals
- String Literals
- Operators
- White space characters

Tokens in C

Keywords

- These are reserved words of the C language. For example **int, float, if, else, for, while** etc.

Identifiers

- An Identifier is a sequence of letters and digits, but must start with a letter. Underscore (_) is treated as a letter. Identifiers are case sensitive. Identifiers are used to name variables, functions etc.
- Valid: **Root, _getchar, __sin, x1, x2, x3, x_1, If**
- Invalid: **324, short, price$, My Name**

Constants

- Constants like **13, ‘a’, 1.3e-5** etc.

Tokens in C

String Literals

- A sequence of characters enclosed in double quotes as "…". For example "13" is a string literal and not number 13. 'a' and "a" are different.

Operators

- Arithmetic operators like `+, -, *, / ,%` etc.
- Logical operators like `||, &&, !` etc. and so on.

White Spaces

- Spaces, new lines, tabs, comments (A sequence of characters enclosed in /* and */) etc. These are used to separate the adjacent identifiers, kewords and constants.

C
Keywords

Keywords or reserved words are tokens that carries special meanings. Programmer are not allowed to change their definition.

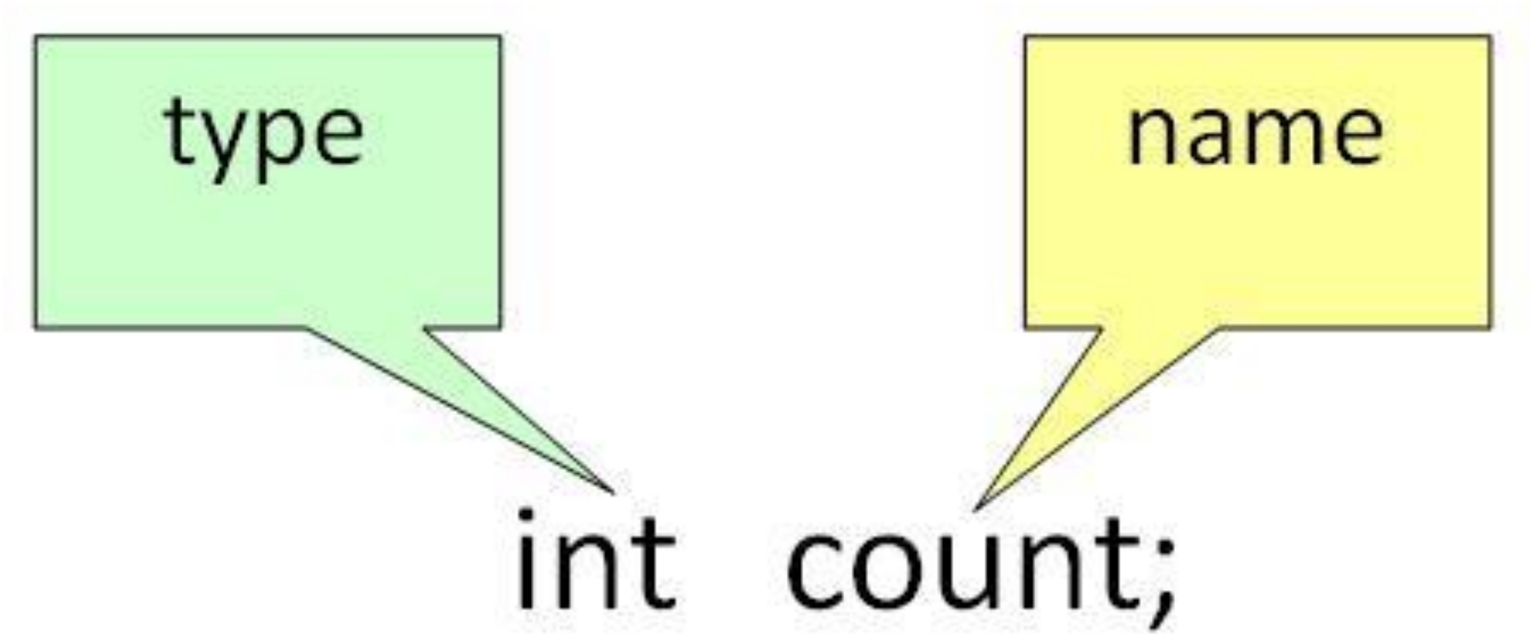
auto	**double**	**int**	**struct**
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sixeof	volatile
do	if	static	

LECTURE 2

Data types

Variable Declaration

Declare a variable which is the data storage for C programs.

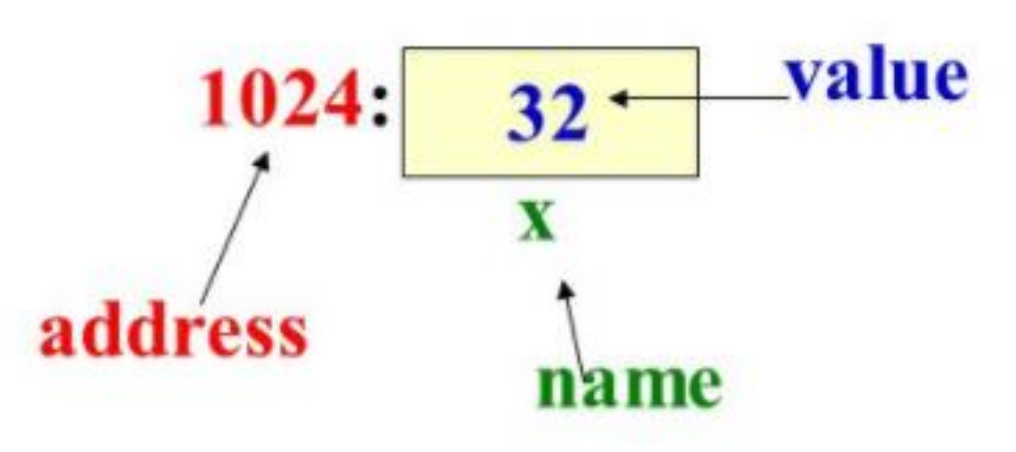


```c
#include<stdio.h>

int main()
{
    int num1;
    num1 = 5;

    float num2;
    num2 = 2.5;

    double num3;
    num3 = 125.24683579;

    char letter;
    letter = 'a';

    return 0;
}
```


Variable Initialization

Variable initialization can be combined with declaration in one statement: int num1 = 5;

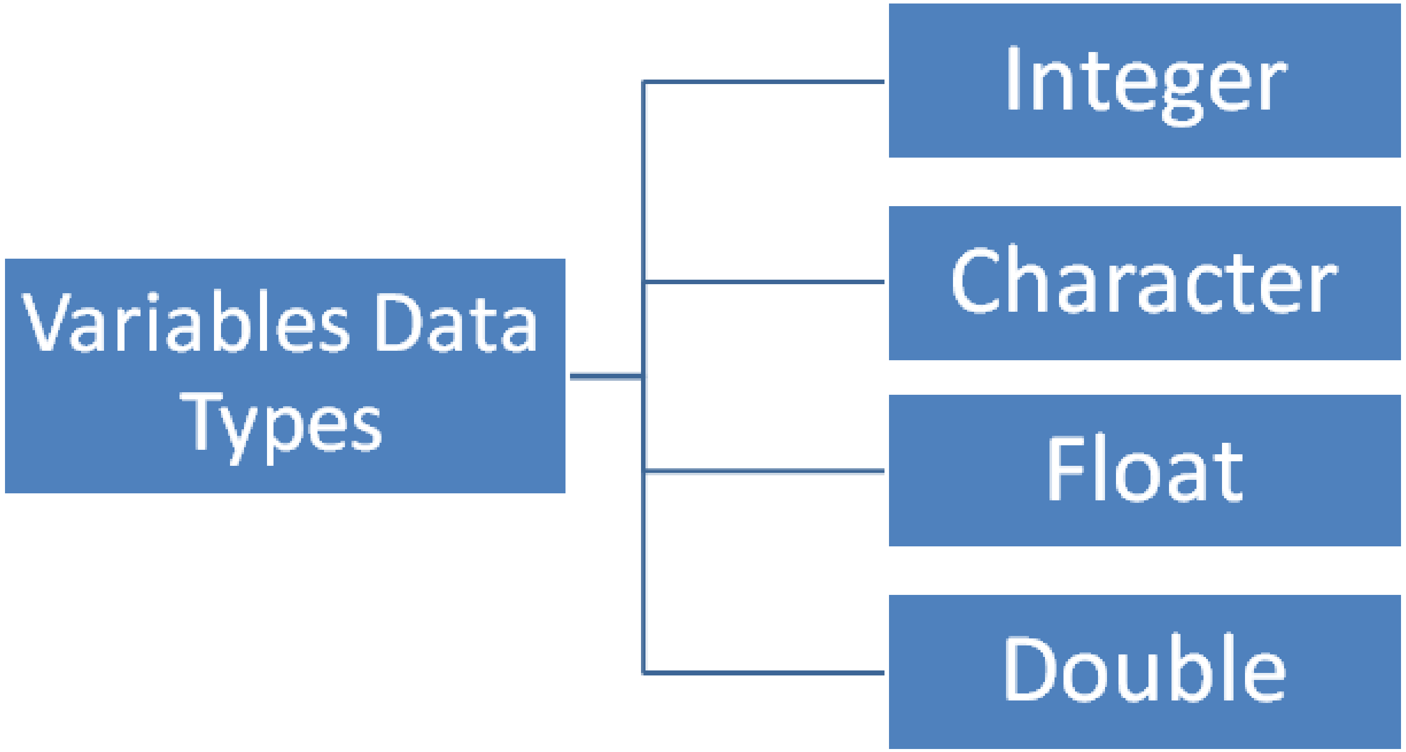
In early version of C-compilers, the declaration and initialization need to be separated.



Variables

Variables are used to store data in a C program. Variables have data types. Data of different types are stored and interpreted in different ways.

The basic data types in a programming language are integer, character, logic value and floating point number.

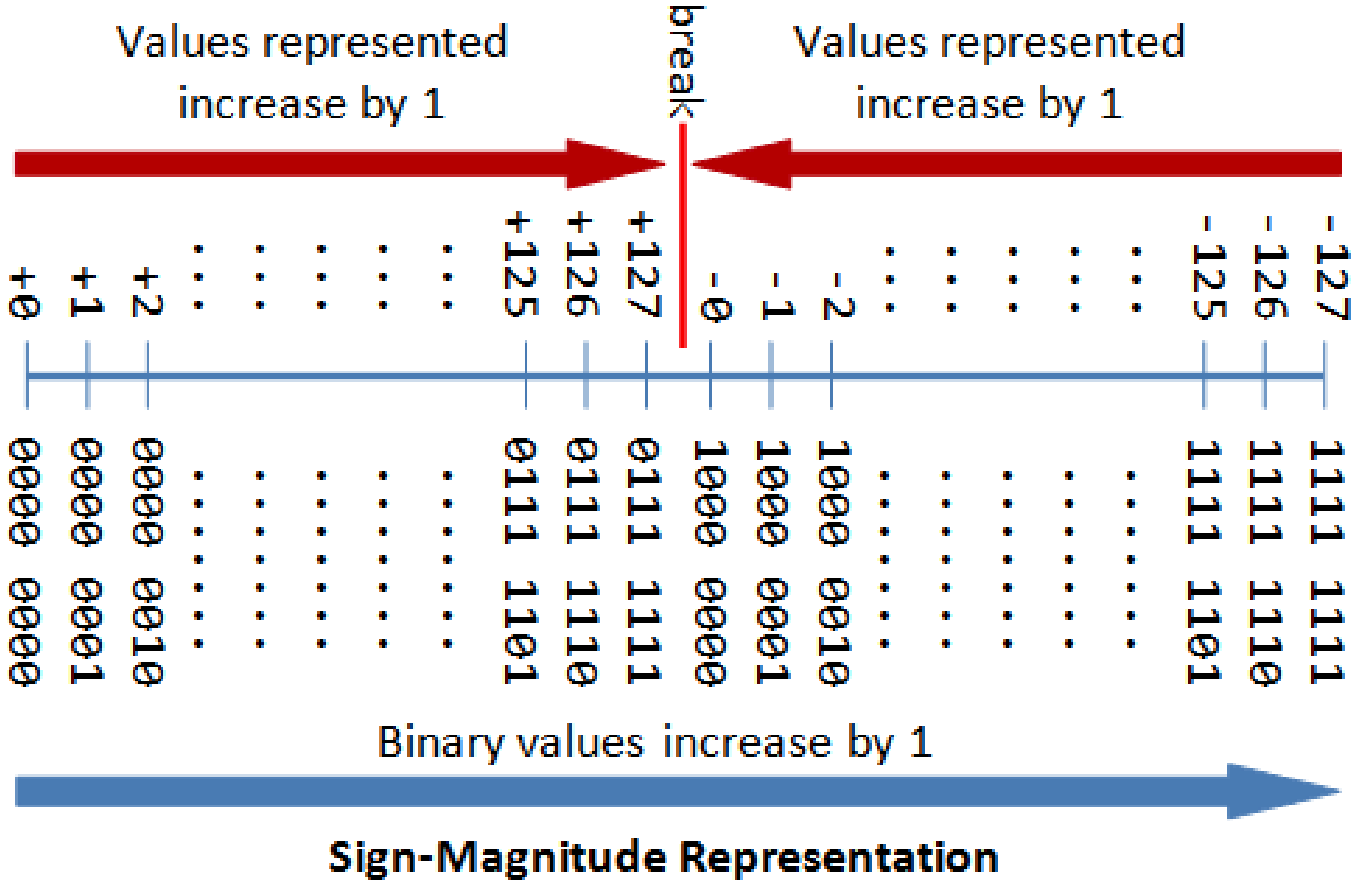


C Integers

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

Signed Integer

C language uses 2's complement for signed integer representation.



Unsigned Integer

Unsigned Integer uses weighted bit vector format.

UNSIGNED NUMBERS	BINARY	HEX.	SIGNED NUMBERS
0	0000 0000	00	0
1	0000 0001	01	+1
2	0000 0010	02	+2
127	0111 1111	7F	+127
128	1000 0000	80	-128
129	1000 0001	81	-127
254	1111 1110	FE	-2
255	1111 1111	FF	-1

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

C Floating Pointer Number Types

Character Type in C

They are integers with special interpretation.

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

Type	Size (bits)	Range
char or signed char	8	-2^7 to $2^7 -1$
unsigned char	8	0 to $2^8 -1$

	$□0	$□1	$□2	$□3	$□4	$□5	$□6	$□7	$□8	$□9	$□A	$□B	$□C	$□D	$□E	$□F	
$0□		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	CR	♫	☼	$0□
$1□	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼	$1□
$2□	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	$2□
$3□	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	$3□
$4□	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	$4□
$5□	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	x	$5□
$6□	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	$6□
$7□	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂	$7□
$8□	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å	$8□
$9□	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ	$9□
$A□	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»	$A□
$B□	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐	$B□
$C□	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧	$C□
$D□	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀	$D□
$E□	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩	$E□
$F□	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■		$F□

$□0 $□1 $□2 $□3 $□4 $□5 $□6 $□7 $□8 $□9 $□A $□B $□C $□D $□E $□F

Boolean Data Type

```c
#include <stdbool.h>

bool  bool1  = true;
bool  bool2  = false;
```

Important!
Compiler needs this or it won't know about "bool"!

`bool` added to C in 1999

Many programmers had already defined their own Boolean type

- **To avoid conflict `bool` is disabled by default**

Demo Program
initialization.c

Go GCC!!!

LECTURE 3

Literals

Constants

Numerical Constants

- Constants like 12, 253 are stored as **`int type. No decimal point.`**
- 12L or 12l are stored as **`long int`**.
- 12U or 12u are stored as **`unsigned int`**.
- 12UL or 12ul are stored as **`unsigned long int`**.
- Numbers with a decimal point (12.34) are stored as double.
- Numbers with exponent (12e-3 = 12 x 10^{-3}) are stored as double.
- 12.34f or 1.234e1f are stored as **`float`**.
- These are not valid constants:

  `25,000   7.1e 4        $200 2.3e-3.4 etc.`

Constants

Character and string constants

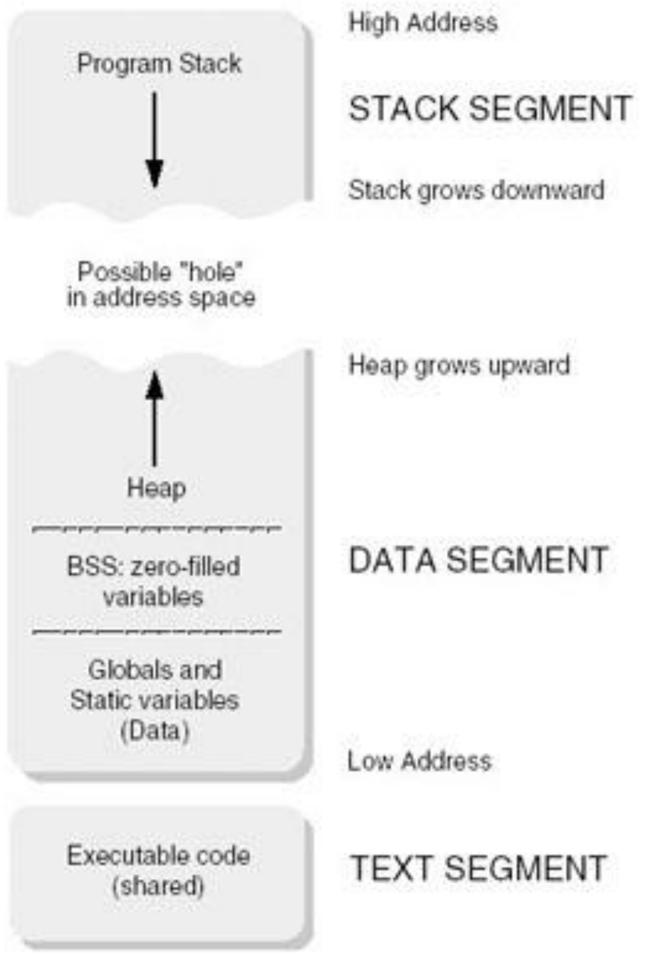
- **‘c’** , a single character in single quotes are stored as char.
  Some special character are represented as two characters in single quotes.
  ‘\n’ = newline, **‘\t’**= tab, **‘\\’** = backslash, **‘\”’** = double quotes.
  Char constants also can be written in terms of their ASCII code.
  ‘\060’ = ‘0’ (Decimal code is 48).
- A sequence of characters enclosed in double quotes is called a string constant or string literal. For example
  “Charu”
  “A”
  “3/9”
  “x = 5”

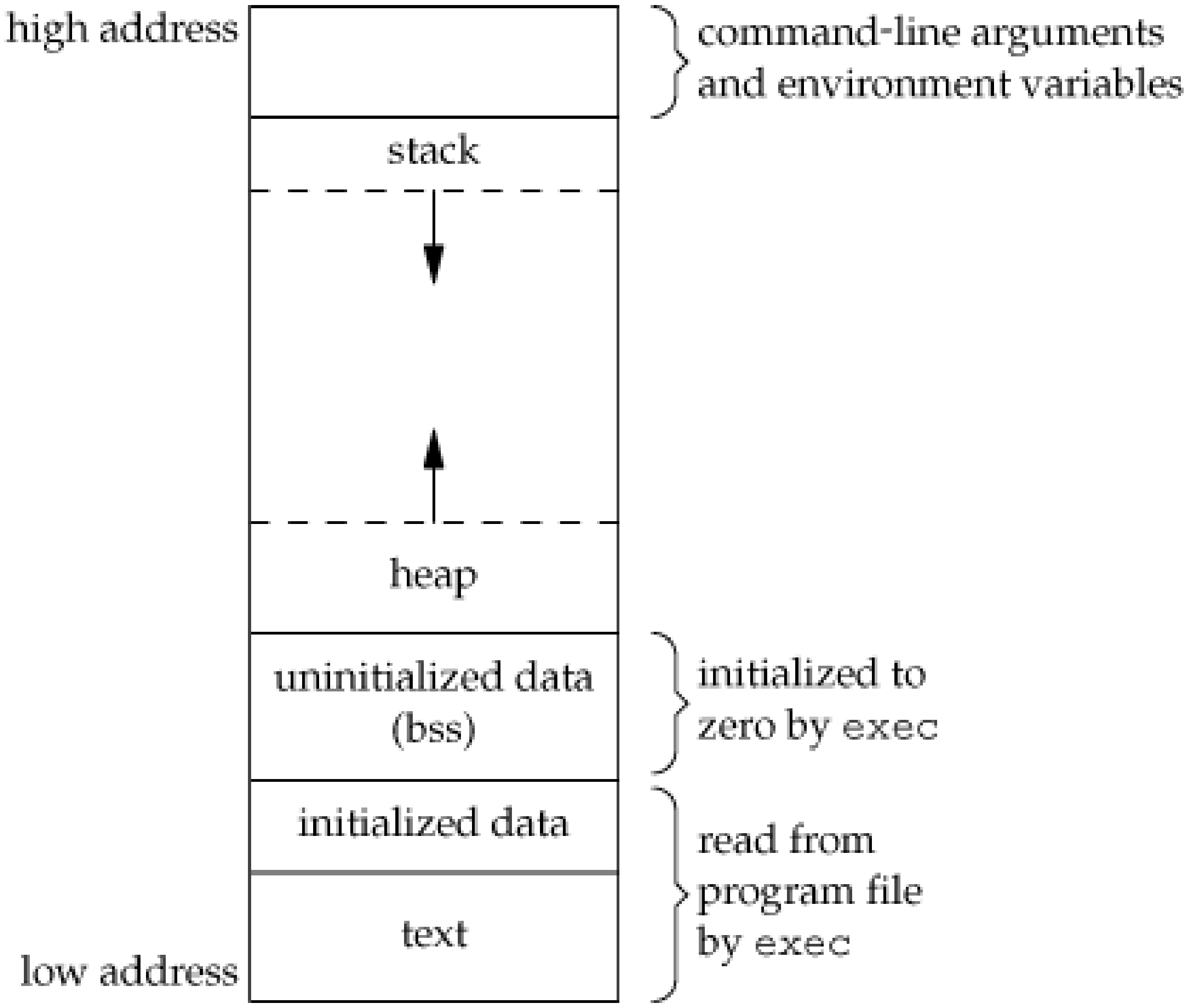
Array of Characters & String

- In C, there is **no string** data type
- But we can create string data type by using array of characters
- We need to declare the **size of array** that is **bigger than the size of string that we want to store at least 1 byte**
  - Because C needs to store **'\0' (zero) at the end of string**
- **String initialization**
  - char a[3] = {'a', 'b', 'c'}; ← **NOT STRING**
  - char a[ ] = "abc"; ← **STRING**
  - char a[20] = "abc"; ← **STRING**

#define DEBUG 0

- Macro is usually used in many situations like a constant. But they are not constants.
- Macro #define is only a Token replacement in the preprocessing stage. That means 0 will be used to replace DEBUG tokens in the program file before the program file is compiled by compiler.
- 0 is a constant but DEBUG is not a variable. It is only a macro word.


Program Stack
High Address
STACK SEGMENT
Stack grows downward
Possible "hole" in address space
Heap grows upward
Heap
BSS: zero-filled variables
DATA SEGMENT
Globals and Static variables (Data)
Low Address
Executable code (shared)
TEXT SEGMENT

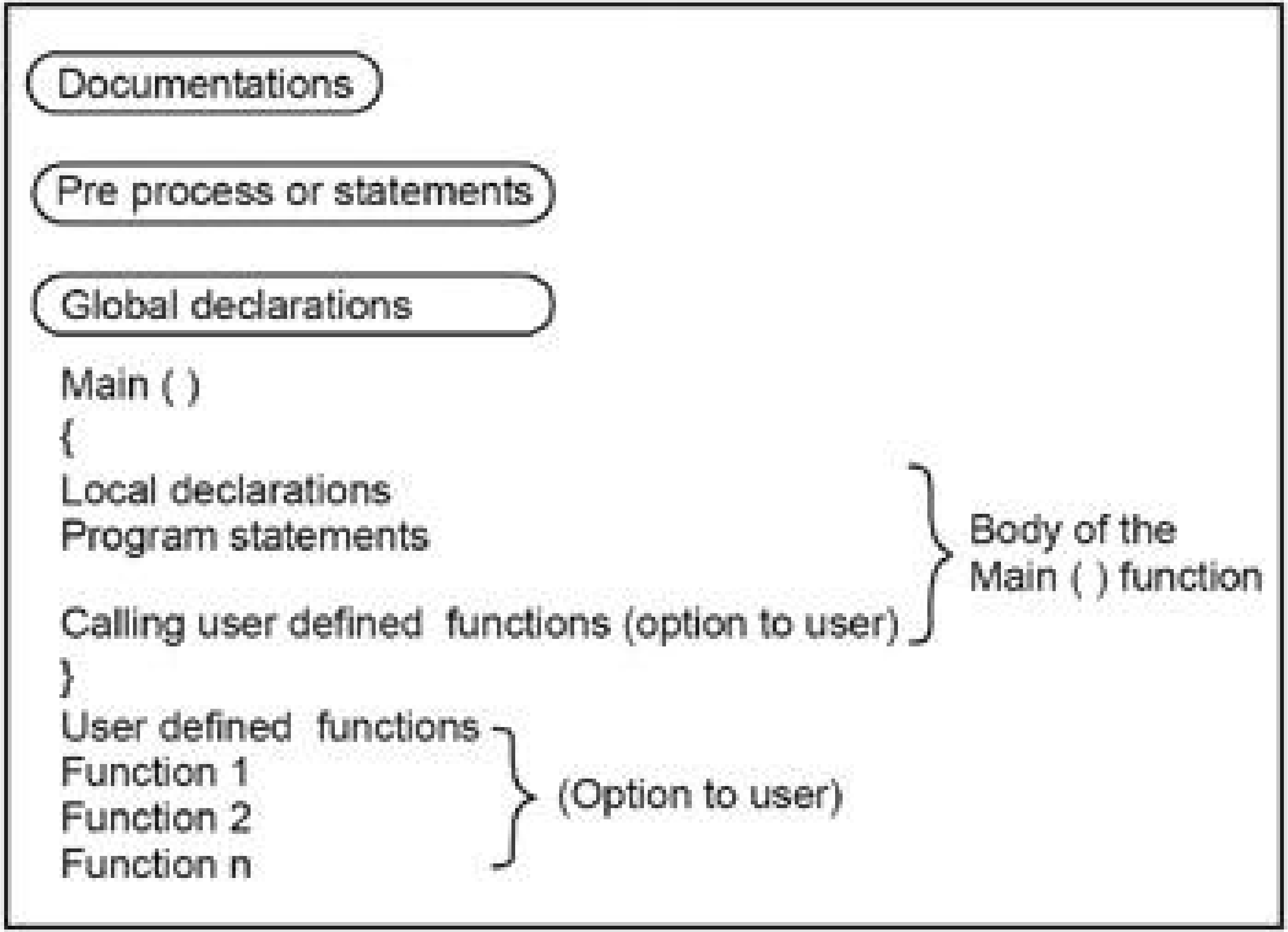

high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

LECTURE 4

My Third Program (Demonstration of program structure)

Basic C Program Structure

1. Document Section
2. Preprocessor Section
   - Inclusion
   - Macro
3. Globals
   - Extern
   - Headers
   - Global variables and functions
4. main() // optional
5. Sub-programming declared in header



Variables

Naming a Variable

- Must be a valid identifier.
- Must not be a keyword
- Names are case sensitive.
- Variables are identified by only first 32 characters.
- Library commonly uses names beginning with _.
- Naming Styles: Uppercase style and Underscore style
- `lowerLimit  lower_limit`
- `incomeTax   income_tax`

Declarations

Declaring a Variable

- Each variable used must be declared.
- A form of a declaration statement is

```
data-type var1, var2,…;
```

- Declaration announces the data type of a variable and allocates appropriate memory location. No initial value (like 0 for integers) should be assumed.
- It is possible to assign an initial value to a variable in the declaration itself.

```
data-type var = expression;
```

- Examples

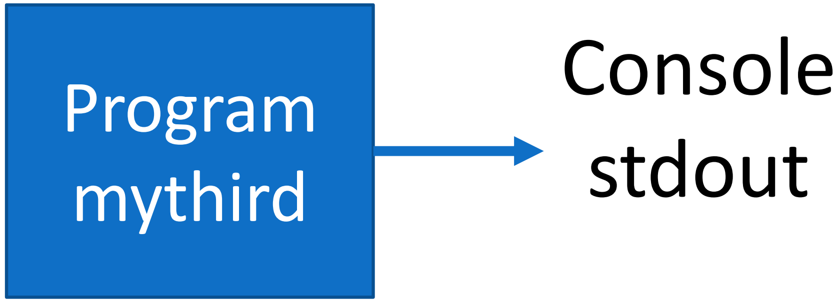
```
int sum = 0;
char newLine = '\n';
float epsilon = 1.0e-6;
```

Demo Program:

mythird.c

```c
#include <stdio.h>
int x;
int main(void){
    x =3;
    printf("X=%d\n", x);
    return 0;
}
```

Go gcc!!!



Global and Local Variables

Global Variables

- These variables are declared outside all functions.
- Life time of a global variable is the entire execution period of the program.
- Can be accessed by any function defined below the declaration, in a file.

```
/* Compute Area and Perimeter of  a circle */
#include <stdio.h>
float pi = 3.14159;  /* Global */

int main(void) {
  float  rad;      /* Local */
  float area;
  float peri;

  printf( "Enter the radius: " );
  scanf("%f" , &rad);

  if ( rad > 0.0 ) {
    float area = pi * rad * rad;
    float peri = 2 * pi * rad;

    printf( "Area = %f\n" , area );
    printf( "Peri = %f\n" , peri );
  }
  else
    printf( "Negative radius\n");

  return 0;
}
```

Global and Local Variables

Local Variables

- These variables are declared inside some functions.
- Life time of a local variable is the entire execution period of the function in which it is defined.
- Cannot be accessed by any other function.
- In general variables declared inside a block are accessible only in that block.

```
/* Compute Area and Perimeter of  a circle */
#include <stdio.h>
float pi = 3.14159;  /* Global */

Int main(void) {
  float  rad;      /* Local */
  float area;
  float peri;

  printf( "Enter the radius: " );
  scanf("%f" , &rad);

  if ( rad > 0.0 ) {
    float area = pi * rad * rad;
    float peri = 2 * pi * rad;

    printf( "Area = %f\n" , area );
    printf( "Peri = %f\n" , peri );
  }
  else
    printf( "Negative radius");

  return 0;
}
```

Demo Program:
computearea.c

Go GCC!!!

LECTURE 5

Number System I (Integer Type)

Integral Types

- Integers are stored in various sizes. They can be signed or unsigned.

- **Example**

  Suppose an integer is represented by a byte (8 bits). Leftmost bit is sign bit. If the sign bit is 0, the number is treated as positive.

  Bit pattern 01001011 = 75 (decimal).

  The largest positive number is 01111111 = $2^7 - 1 = 127$.

  Negative numbers are stored as two's complement or as one's complement.

  -75 = 10110100 (one's complement).

  -75 = 10110101 (two's complement).

Integral Types

- **`char`** Stored as 8 bits. Unsigned 0 to 255.
  Signed -128 to 127.
- **`short int`** Stored as 16 bits. Unsigned 0 to 65535.
  Signed -32768 to 32767.
- **`int`** Same as either short or long int.
- **`long int`** Stored as 32 bits. Unsigned 0 to 4294967295.
  Signed -2147483648 to 2147483647

Examples of Integer Constants			
type	hexadecimal	octal	decimal
char	\0x41	\0101	N.A.
int	0x41	0101	65
unsigned int	0x41u	0101u	65u
long	0x41L	0101L	65L
unsigned long	0x41UL	0101UL	65UL
long long	0x41LL	0101LL	65LL
unsigned long long	0x41ULL	0101ULL	65ULL

Integer Constant

Ob	binary
O	Octal
Ox	Hexadecimal
default	Decimal

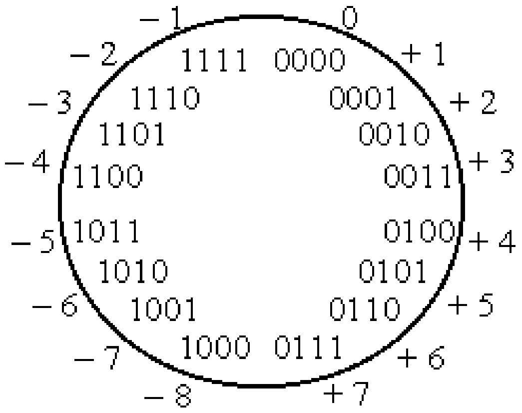
Specific integral type limits

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
`int8_t`	Signed	8	1	-2^{7} which equals −128	$2^{7} - 1$ which is equal to 127
`uint8_t`	Unsigned	8	1	0	$2^{8} - 1$ which equals 255
`int16_t`	Signed	16	2	-2^{15} which equals −32,768	$2^{15} - 1$ which equals 32,767
`uint16_t`	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
`int32_t`	Signed	32	4	-2^{31} which equals −2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
`uint32_t`	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
`int64_t`	Signed	64	8	-2^{63} which equals −9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
`uint64_t`	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

C99 #include <stdint.h> Standardized Integer Formats

Twos complement arithmetic

- Subtraction = negation and addition
  - Ignore the carry
    - Same as a full rotation around the wheel



Add		Invert and add		Invert and add	
4	0100	4	0100	– 4	1100
+ 3	+ 0011	– 3	+ 1101	+ 3	+ 0011
= 7	= 0111	=1	1 0001	– 1	1111
		drop carry	= 0001		

Integer : X

X's 1's Complement:

$(2^n-1)- X$

X's 2'x Complement:

$(2^n-1)-X+1 = 2^n-X$

By Ignoring 2^n,

The number become -X

One's Complement

Invert all bits. Each 1 becomes a 0, and each 0 becomes a 1.

Original Value		One's Complement
0	→	1
1		0
1010	→	0101
1111		0000
11110000	→	00001111
10100011		01011100
11110000 10100101	→	00001111 01011010

Two's Complement

First, find the one's complement of a value, and then add 1 to it.

Original Value	11000010	10000000	10101011
One's Complement	00111101	01111111	01010100
Add 1	00111101 + 1	01111111 + 1	01010100 + 1
	00111110	10000000*	01010101

Two's Complements

* This is not an error. This is a contrived problem to show that it is possible for a two's complement to match the original value.

Addition and Subtraction (Examples)

- Subtraction of Numbers in Twos Complement

(a) (+2) − (+7)

```
  0010
+ 1001
  1011 = -5
```

(b) (+5) - (+2)

```
  0101
+ 1110
  1011 =3
```

(c) (-5) - (+2)

```
  1011
+ 1110
  11001 =-7
```

(d) (+5) - (-2)

(e) (+7) - (-7)

(f) (-6) - (+4)

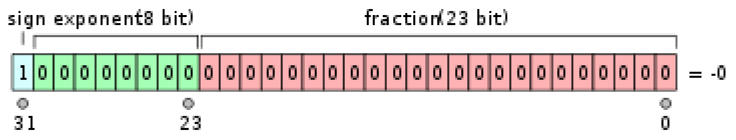
LECTURE 6

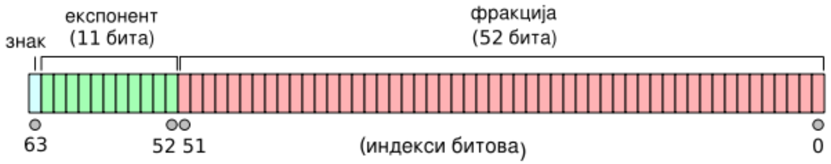
Number System II (Floating point number)

Floating Point Numbers

Floating Point Data Types

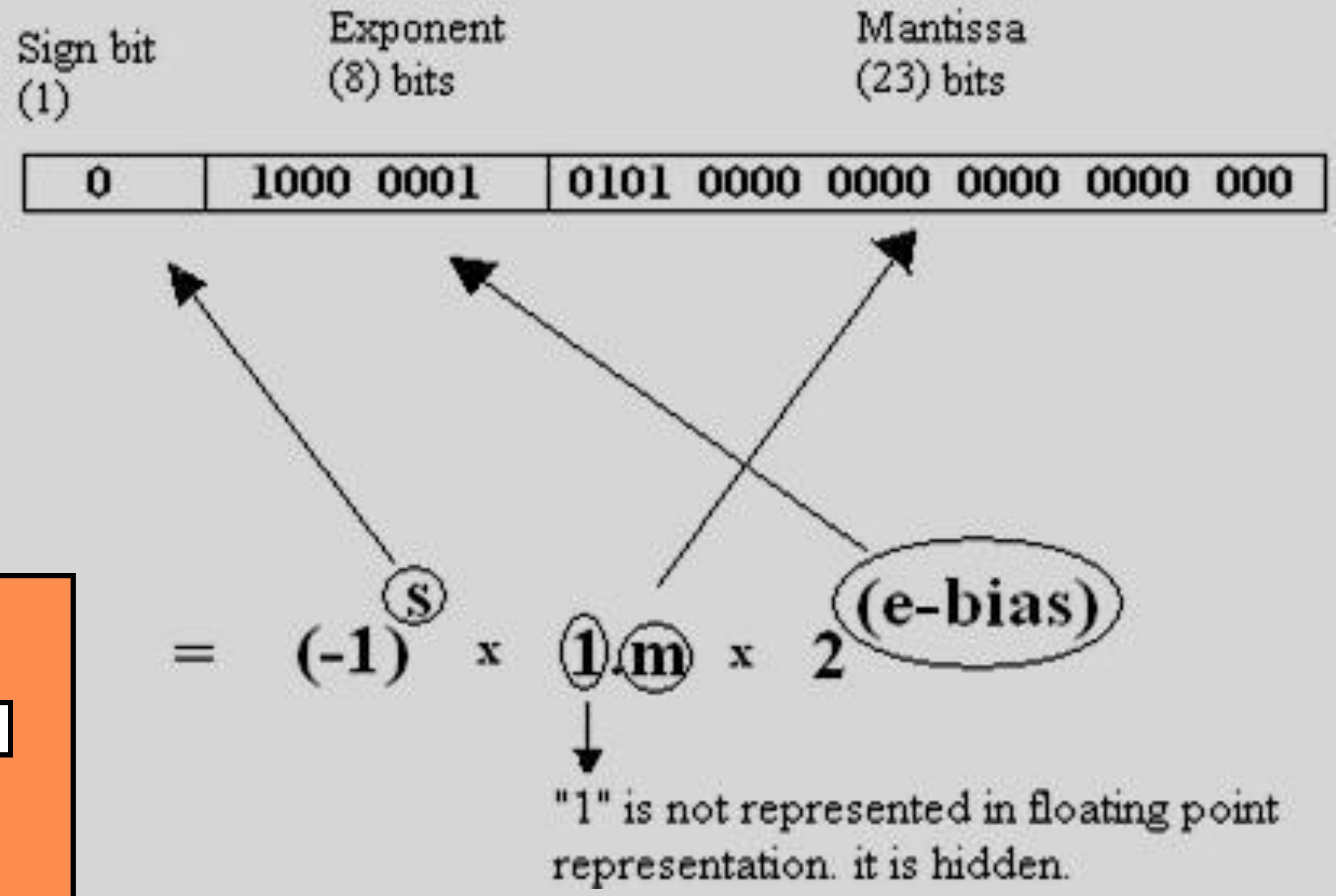
C type	IEE754 Name	Bits	Range
float	Single Precision	32	-3.4E38 .. 3.4E38
double	Double Precision	64	-1.7E308 .. 1.7E308





- Floating point numbers are rational numbers. Always signed numbers.
- **float** Approximate precision of 6 decimal digits .
  - Typically stored in 4 bytes with 24 bits of signed mantissa and 8 bits of signed exponent.
- **double** Approximate precision of 14 decimal digits.
  - Typically stored in 8 bytes with 56 bits of signed mantissa and 8 bits of signed exponent.
- One should check the file limits.h to what is implemented on a particular machine.

Compile with: `gcc -std=c99 -mfpmath=387 -o test_c99_fp -lm test_c99_fp.c`



IEEE 754 Floating Point Standard

s	e=exponent	m=mantissa
1 bit	8 bits	23 bits

$$\text{number} = (-1)^{s} * (1.m) * 2^{e-127}$$

Floating Point Arithmetic

Representation $\pm 0.d_1 d_2 \cdots d_p \times B^e$

- All floating point numbers are stored as
- such that d_1 is nonzero. B is the base. p is the precision or number of significant digits. e is the exponent. All these put together have finite number of bits (usually 32 or 64 bits) of storage.
- Example
- Assume B = 10 and p = 3.
- `23.7       = +0.237E2`
- `23.74      = +0.237E2`
- `37000      = +0.370E5`
- `37028      = +0.370E5`
- `-0.000124 = -0.124E-4`

Floating Point Arithmetic

Representation

- $S_k = \{ x \mid B^{k-1} <= x < B^k \}$. Number of elements in each Sk is same. In the previous example it is 900.
- Gap between seuccessive numbers of Sk is B^{k-p}.
- B1-p is called machine epsilon. It is the gap between 1 and next representable number.
- Underflow and Overflow occur when number cannot be represented because it is too small or too big.
- Two floating points are added by aligning decimal points.
- Floating point arithmetic is not associative and distributive.

LECTURE 7

Arithmetic Operators

Operators in C Language

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Unary +/- Operators

- In these unary + - expressions

  + expression e.g. +3

  - expression e.g. -3

  the expression operand must be of arithmetic type. The result is the value of the operand after any required integral promotions for the unary plus ('+') operator, or negative of the value of the operand after any required integral promotions for the unary minus ('-') operator.

- Floating point negation is internally executed using the fneg function.
- Note that both '+' and '-' operators also have a binary form.

Arithmetic Operators

- `+`, `-` , `*`, `/` and the modulus operator %.
- + and – have the same precedence and associate left to right.

```
3 - 5 + 7 = ( 3 - 5 ) + 7 ≠ 3 - ( 5 + 7 )
3 + 7 - 5 + 2 = ( ( 3 + 7 ) - 5 ) + 2
```

- `*`, `/`, `%` have the same precedence and associate left to right.
- The `+`, `-` group has lower precendence than the `*`, `/` `%` group.

```
3 - 5 * 7 / 8 + 6 / 2
3 - 35 / 8 + 6 / 2
3 - 4.375 + 6 / 2
3 - 4.375 + 3
-1.375 + 3
1.625
```

Precedence

- Take a look at the math formula below:

```
// Define some variables
double x, y, answer;

// Set values in each
x = 4.0;
y = 2.0;
answer = x + y * 3.0 / 4.0 - 1.5; // answer now holds the value 4.0
```

Did you think this would give 3.0 because?
4.0 + 2.0 = 6.0
*6.0 * 3.0 = 18.0*
18.0 / 4.0 = 4.5
4.5 - 1.5 = 3.0

- If you expected the answer to be 3.0 you may be surprised to find that **answer** now holds the value 4.0 instead of 3.0 because of operator **Precedence**. In C++ math operations multiplication and division are always done first moving from left to right in the equation. Next addition and subtraction are done moving from left to right in the equation.

Precedence

- There is a way around this problem. The answer is to use parentheses. Math operations always follow precedence, but they work from the innermost set of parentheses outward. For example, if the formula given in the above example had been written as:
  answer = ((x + y) * 3.0 / 4.0) - 1.5; The order of calculation would have been as follows:
  - Add 4.0 (x) and 2.0 (y) (the inner most set of parentheses) to get 6.0
  - Multiply 6.0 times 3.0 to get 18.0, then divide by 4.0 to get 4.5. (the outer most set of parentheses)
  - Subtract 1.5 from 4.5 to get the final answer of 3.0.
- Always use parentheses liberally in math formulas to ensure that the calculations will be performed in the order in which you intended.

Integer Division - / Integer Modulus - %

```
// Define some variables
int a, b, c, answer;

// Set values in each
a = 16;
b = 3;
c = 2;

// Do some math
answer = a / b;    // answer now holds the value 5
answer = a % b;    // answer now holds the value 1;
answer = a / c;    // answer now holds the value 8
answer = a % c;    // answer holds the value 0
```

- In these, remember you are doing integer division.
- 16 divided by 3 gives 5 with a remainder of 1.
- 16 divided by 2 gives 8 with a remainder of 0.

Decimal Division

```
// Define some variables
double a, b, c, d, e, f, answer;

// Set values in each
a = 3.0;
b = 16.0;
c = 2.0;
d = 36.0
e = -2.0;
f = 0.0;

// Do some math
answer = a + b - c;  // answer now holds the value 17.0
answer = b * c;      // answer now holds the value 32.0;
answer = c * d / e;  // answer now holds the value -36.0
answer = d / f;      // answer holds no value at all because
                     //   your program just crashed.
          Oops!  You know you can't divide by zero.
```

In floating point number operations:

1. Division by zero will cause exceptions. Program will crash.

2. Overflow and underflow of addition and subtraction will cause numerical problem. Unpredictable situation will arise. (Program will still be running but the result cannot be guaranteed.)

Arithmetic Operators

- % is a modulus operator. x % y results in the remainder when x is divided by y and is zero when x is divided by y.
- Cannot be applied to float or double variables.
- Example:

```
if ( num % 2 == 0 )
    printf("%d is an even number\n", num)';
else
    printf("%d is an odd number\n", num);
```

Demo:
arithmetic.c

Go GCC!!!

LECTURE 8

Increment and Decrement

Prefix/Postfix Arithmetic Operators

Prefix Increment : ++a

- example:
  - int a=5;
  - b=++a; // value of b=6; a=6;

Postfix Increment: a++

- example
  - int a=5;
  - b=a++; //value of b=5; a=6;

Prefix/Postfix Arithmetic Operators

Prefix Decrement : --a

- example:
    - int   a=5;
    - b=--a; // value of b=4; a=4;

Postfix Decrement: a--

- example
    - int a=5;
    - b=a--; //value of b=5; a=4;

Increment and Decrement Operators

- The operators `++` and `--` are called increment and decrement operators.
- `a++` and `++a` are equivalent to `a += 1.`
- `a--` and `--a` are equivalent to `a -= 1.`
- `Pre-Increment:  ++a op b` is equivalent to `a++; a op b;`
- `Post-Increment: a++ op b` is equivalent to `a op b; a++;`
- `Example`

```
Let b = 10 then
(++b)+b+b  = 33
b+(++b)+b  = 33
b+b+(++b)  = 31
b+b*(++b)  = 132
```

Difference Between Pre/Port Increment & Decrement Operators In C:

Operator	Operator/Description
Pre increment operator (++i)	value of i is incremented before assigning it to the variable i
Post increment operator (i++)	value of i is incremented after assigning it to the variable i
Pre decrement operator (–i)	value of i is decremented before assigning it to the variable i
Post decrement operator (i–)	value of i is decremented after assigning it to variable i

Demo Program:

prepostfix package

Go GCC!!!

LECTURE 9

Augmented Assignment

Assignment Operator (=)

- There are 11 assignment operators in C language. The '=' operator is the simple assignment operator; the other 10 ('*=', '/=', '%=', '+=', '-=', '<<=', '>>=', '&=', '^=' and '|=') are known as compound assignment operators. All of them use the following syntax:
  - expr1 assignment-operator expr2 eg. i = 3;

Assignment Operator (=)

- In the expression expr1 = expr2, expr1 must be a modifiable **lvalue**. The value of expr2, after conversion to the type of expr1, is stored in the object designated by expr1 (replacing expr1's previous value). The value of the assignment expression is the value of expr1 after the assignment. That's why multiple assignments like
  - x = y = z = 10;
  - a = b + 2 * (c = d - 1);
- are possible. Note that the assignment expression is not itself an **lvalue**.

Assignment operators

- The general form of an assignment operator is
- `v op= exp`
- Where v is a variable and op is a binary arithmetic operator. This statement is equivalent to
- `v = v op (exp)`
- `a = a + b` can be written as `a += b`
- `a = a * b` can be written as `a *= b`
- `a = a / b` can be written as `a /= b`
- `a = a - b` can be written as `a -= b`
- `a = a % b` can be written as `a %= b`

Operators	Example/Description
=	sum = 10; 10 is assigned to variable sum
+=	sum += 10; This is same as sum = sum + 10
-=	sum -= 10; This is same as sum = sum – 10
*=	sum *= 10; This is same as sum = sum * 10
/=	sum /= 10; This is same as sum = sum / 10
%=	sum %= 10; This is same as sum = sum % 10
&=	sum&=10; This is same as sum = sum & 10
^=	sum ^= 10; This is same as sum = sum ^ 10

Demo:
augmented.c

Go gcc!!!

LECTURE 10

Relational and Logical Operators and Expression

Relational Operators

- <, <=, > >=, ==, != are the relational operators. The expression

  `operand1 relational-operator operand2`

  takes a value of 1(int) if the relationship is true and 0(int) if relationship is false.
- Example

```
int a = 25, b = 30, c, d;
c = a < b;
d = a > b;
```

value of c will be 1 and that of d will be 0.

Demo Program:

relational.c

Go gcc!!!

Logical Operators

- `&&`, `||` and `!` are the three logical operators.
- `expr1 && expr2` has a value 1 if `expr1` and `expr2` both are nonzero.
- `expr1 || expr2` has a value 1 if `expr1` and `expr2` both are nonzero.
- `!expr1` has a value 1 if expr1 is zero else 0.
- Example
- `if ( marks >= 40 && attendance >= 75 ) grade = 'P'`
- `If ( marks < 40 || attendance < 75 ) grade = 'N'`

Demo Program:
logical.c

Go gcc!!!

C Programming Expression

Expressions	Validity
a + b	Expression is valid since it contain + operator which is binary operator
+ + a + b	Invalid Expression

1. In programming, an expression is any legal combination of symbols that represents a value. [Definition from Webopedia]

2. C Programming Provides its own rules of Expression, whether it is legal expression or illegal expression. For example, in the C language x+5 is a legal expression.

3. Every expression consists of at least one operand and can have one or more operators.

4. Operands are values and Operators are symbols that represent particular actions.

C Expressions

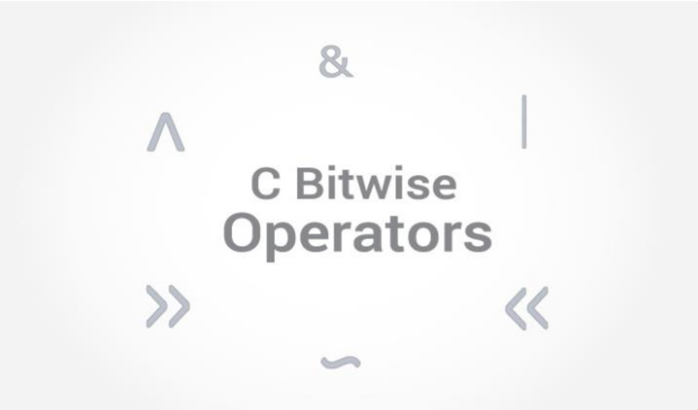
- Expressions in C is any valid combination of operators, constants, functions, and variables.
- Expression is evaluated using precedence & associativity rule.

```
Step 1.   y = 2 * 5 * 5 + 3 * 5 + 7;     (Leftmost multiplication)
              2 * 5 is 10

Step 2.   y = 10 * 5 + 3 * 5 + 7;        (Leftmost multiplication)
              10 * 5 is 50

Step 3.   y = 50 + 3 * 5 + 7;            (Multiplication before addition)
                   3 * 5 is 15

Step 4.   y = 50 + 15 + 7;               (Leftmost addition)
              50 + 15 is 65

Step 5.   y = 65 + 7;                    (Last addition)
              65 + 7 is 72

Step 6.   y = 72                         (Last operation—place 72 in y)
```

Expression can be on the left-hand side or right-hand side.

LECTURE 11

Bitwise operators I (Bitwise bit operators)



Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	Shift left
>>	Shift right

Example for Bitwise Operations

- Consider x=40 and y=80. Binary form of these values are given below.
  x = 00101000
  y= 01010000

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

- All bit wise operations for x and y are given below.
  x&y = 00000000 (binary) = 0 (decimal)
  x|y = 01111000 (binary) = 120 (decimal)
  ~x = 11010111 = -41 (decimal)
  x^y = 01111000 (binary) = 120 (decimal)
  x << 1 = 01010000 (binary) = 80 (decimal)
  x >> 1 = 00010100 (binary) = 20 (decimal)

Bitwise AND - &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001
  ________
  00001000  = 8 (In decimal)
```

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output

```
Output = 8
```

Bitwise OR - |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  ________
  00011101  = 29 (In decimal)
```

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a|b);
    return 0;
}
```

Output

```
Output = 29
```

Bitwise Exclusive-OR - ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
| 00011001
  ________
  00010101  = 21 (In decimal)
```

```c
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

Output

```
Output = 21
```

Bitwise Complement - ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
~ 00100011
  ________
  11011100  = 220 (In decimal)
```

```
#include <stdio.h>
int main()
{
    printf("complement = %d\n",~35);
    printf("complement = %d\n",~-12);
    return 0;
}
```

Output

```
complement = -36
Output = 11
```

Demo Program:
bitwise.c

Go gcc!!!

LECTURE 12

Bitwise operators II (Bitwise Shift Operators)

Left shift (<<)

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

```
212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 =11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Arithmetic right shift (>>)

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)
212>>2 = 00110101 (In binary) [Right shift by two bits]
212>>7 = 00000001 (In binary)
212>>8 = 00000000
212>>0 = 11010100 (No Shift)
```

Shift Operators

```
#include <stdio.h>
int main()
{
      int num=212, i;
      for (i=0; i<=2; ++i)
            printf("Right shift by %d: %d\n", i, num>>i);

       printf("\n");

       for (i=0; i<=2; ++i)
            printf("Left shift by %d: %d\n", i, num<<i);

       return 0;
}
```

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Logical right shift (>>>)

A logical right shift is the converse to the left shift. Rather than moving bits to the left, they simply move to the right. For example, shifting the number 12:

```
00000000 00000000 00000000 00001100
```

to the right by one position (`12 >>> 1`) will get back our original 6:

```
00000000 00000000 00000000 00000110
```

Logical right shift (>>>)

Lost bits are gone

However, a shift cannot reclaim "lost" bits. For example, if we shift this pattern:

```
00111000 00000000 00000000 00000110
```

to the left 4 positions (`939,524,102 << 4`), we get 2,147,483,744:

```
10000000 00000000 00000000 01100000
```

and then shifting back (`(939,524,102 << 4) >>> 4`) we get 134,217,734:

```
00001000 00000000 00000000 00000110
```

We cannot get back our original value once we have lost bits.

Difference Between Arithmetic Shift and Logical Shift

Arithmetic Shift (with sign-extension)

Logical Shift (truncation)

The arithmetic right shift is exactly like the logical right shift, except instead of padding with zero, it pads with the most significant bit. This is because the most significant bit is the *sign* bit, or the bit that distinguishes positive and negative numbers. By padding with the most significant bit, the arithmetic right shift is sign-preserving.

For example, if we interpret this bit pattern as a negative number:

```
10000000 00000000 00000000 01100000
```

we have the number -2,147,483,552. Shifting this to the right 4 positions with the arithmetic shift (-2,147,483,552 >> 4) would give us:

```
11111000 00000000 00000000 00000110
```

or the number -134,217,722.

So we see that we have preserved the sign of our negative numbers by using the arithmetic right shift, rather than the logical right shift. And once again, we see that we are performing division by powers of 2.

Demo Program:
shift.c

Go gcc!!!

LECTURE 13

Casting

Integer Division

```
int a, b, q, r;
q = a / b;
r = a % b;
a == b * (a / b) + (a % b) == b * q + r;
```

Integer Division

- When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. If the quotient a/b is representable, the expression (a/b)*b + a % b shall equal a.
- This is often called ‘‘truncation toward zero’’.
- The usual arithmetic conversions are performed on the operands.

and:

- The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

Type Conversions

- The operands of a binary operator must have a the same type and the result is also of the same type.
- Integer division:

```
c = (9 / 5)*(f - 32)
```

- The operands of the division are both int and hence the result also would be int. For correct results, one may write

```
c = (9.0 / 5.0)*(f - 32)
```

Type Conversions

- In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called Automatic Type Casting.

```
c = (9.0 / 5)*(f - 32)
```

- It is possible to force a conversion of an operand. This is called Explicit Type casting.

```
c = ((float) 9 / 5)*(f - 32)
```

Automatic Type Casting

1. char and short operands are converted to int
2. Lower data types are converted to the higher data types and result is of higher type.
3. The conversions between unsigned and signed types may not yield intuitive results.
4. Example

   `float f; double d; long l;`
   `int i; short s;`
   `d + f` f will be converted to `double`
   `i / s` s will be converted to `int`
   `l / i` i is converted to `long`; `long` result

```
Hierarchy
Double
float
long
Int
Short and
char
```

Note:
Unlike Java, C does not allow automatic casting to string.

Explicit Type Casting

- The general form of a type casting operator is

  (type-name) expression

- It is generally a good practice to use explicit casts than to rely on automatic type conversions.
- Example

```
C = (float)9 / 5 * ( f – 32 )
```

- `float` to `int` conversion causes truncation of fractional part
- `double` to `float` conversion causes rounding of digits
- `long int` to `int` causes dropping of the higher order bits.

Demo Program:
casting.c

Go gcc!!!

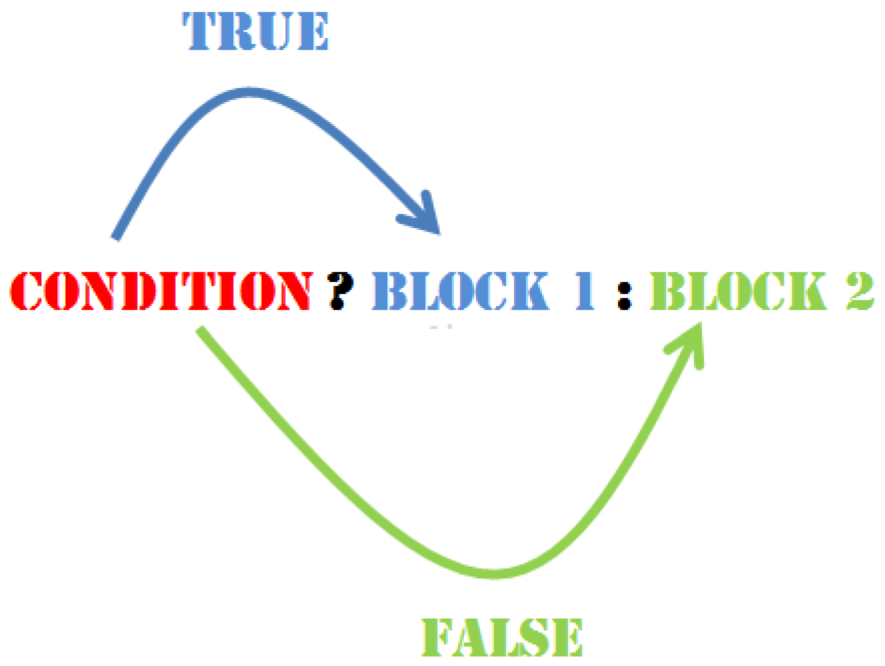
LECTURE 14

Conditional Expression

Conditional Operators [? :]

Ternary Operator Statement in C

- They are also called as Ternary Operator .
- They also called as ?: operator
- Ternary Operators takes on 3 Arguments



Conditional Operator [? :]

A conditional expression is of the form

```
expr1 ? expr2 : expr3
```

The expressions can recursively be conditional expressions.

A substitute for `if-else`

Example :

```
(a<b)?((a<c)?a:c):((b<c)?b:c)
```

What does this expression evaluate to?

Conditional Operator [? :]

A conditional expression is of the form

```
expr1 ? expr2 : expr3
```

The expressions can recursively be conditional expressions.

A substitute for `if-else`

Example :

```
(a<b)?((a<c)?a:c):((b<c)?b:c)
```

This evaluates to min(a,b,c)

Demo Program:

C1.c C2.c

Go gcc!!!

LECTURE 15

Reference and De-reference

What is a pointer?

```
int x = 10;
int *p = NULL;
```

Declares a pointer to an integer

```
p = &x;
```

`&` is **address** operator gets address of x

```
*p = 20;
```

`*` **dereference** operator gets value at `p`

Note: You can also use the asterisk as an operator to dereference a pointer, or as the multiplication operator. Asterisk may be used also as a punctuator for creating pointer types.

Referencing operator ('&')

- In the expression

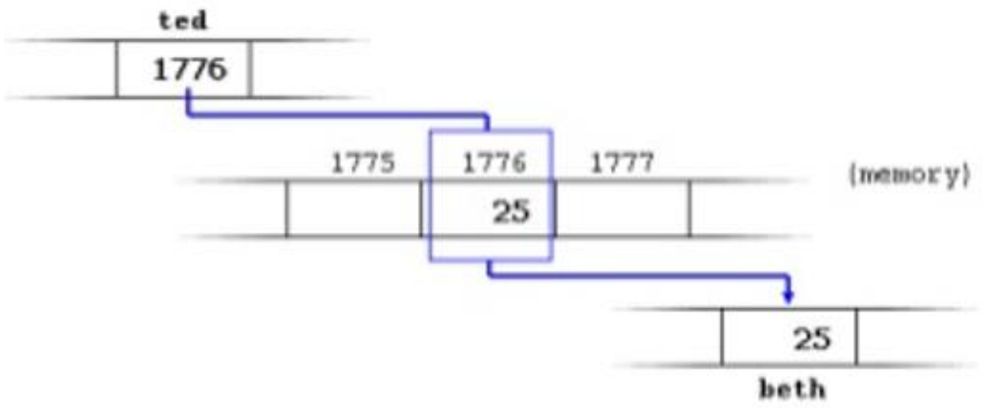
  & expr

  which means "take the address of the expr", the expr operand must be one of the following:
  a function designator; an lvalue designating an object that is not a bit field and is not declared with the register storage class specifier.
- If the operand is of type type, the result is of type "pointer to type".
- The '&' symbol is also used in C as a binary bitwise AND operator.

Dereference Operator (*)

(*) → "Values pointed by"

```
beth = *ted;
```



Note:
* Dereference to a value.
-> Dereference to a value data field.

- Notice the difference:

  & is the reference operator and can be read as "address of"

  * is the dereference operator and can be read as "value pointed by"

Dereferencing operator ('*')

In the expression

    * expr

which means "the object pointed to by expr", the expr must have type "pointer to type," where type is any data type. The result of the indirection is of **type** type.

- If the operand is of type "pointer to function", the result is a function designator. If the operand is a pointer to an object, the result is an lvalue designating that object.
- In the following situations, the result of indirection is undefined:

  The expr is a null pointer.

  The expr is the address of an automatic (local) variable and execution of its block has terminated.

Dereferencing Operator for Object (Struct) (->)

- -> operator is used to find a data field in the struct that the pointer pointing to.
- It can be used in combination of & and *. To find the proper data.

```
struct foo
{
  int x;
  float y;
};

struct foo var;
struct foo* pvar;

var.x = 5;
(&var)->y = 14.3;
pvar->y = 22.4;
(*pvar).x = 6;
```

LECTURE 16

Precedence of Operators

Operator Precedence

- Meaning of   a + b * c ?
-    is it a+(b*c)   or   (a+b)*c ?
- All operators have precedence over each other
- *, / have more precedence over   +, - .
  - If both *, / are used, associativity comes into picture. (more on this later)
  - example :
    - 5+4*3 = 5+12= 17.

Tokens	Operator	Class	Precedence	Associates
names, *literals*	simple tokens	primary	16	n/a
a[k]	subscripting	postfix		left-to-right
f(...)	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	**postfix**		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	**prefix**	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	indirection (*dereference*)	unary		right-to-left

Tokens	Operator	Class	Precedence	Associates
(*type*)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
|	bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right