Fundamentals of Computer and Programming

Lecture 3

C Programming Basics

Instructor: Morteza Zakeri, Ph.D.

(zakeri@aut.ac.ir)

Modified Slides from Dr. Hossein Zeinali and Dr. Bahador Bakhshi

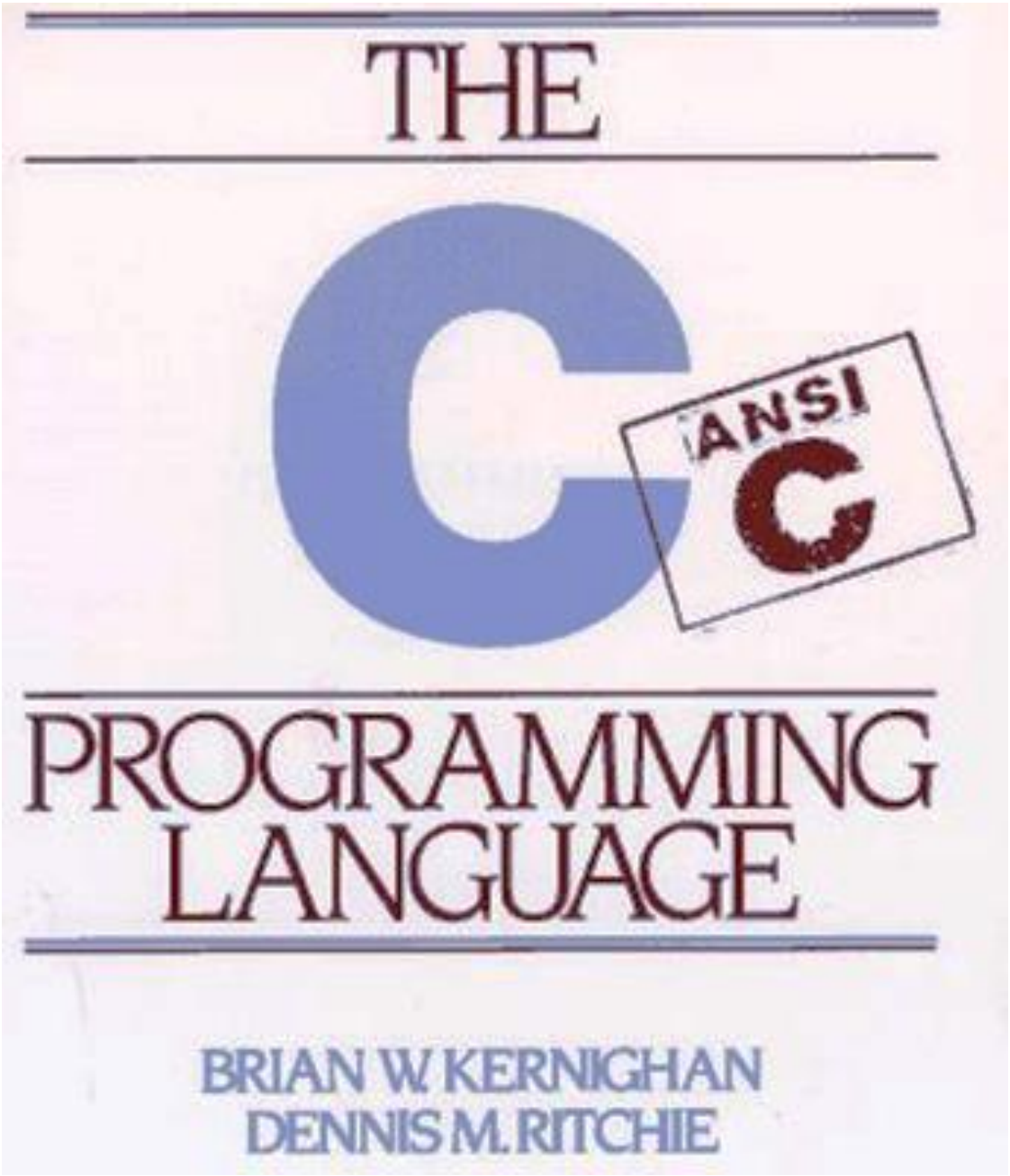
School of Computer Engineering,

Amirkabir University of Technology

Spring 2025

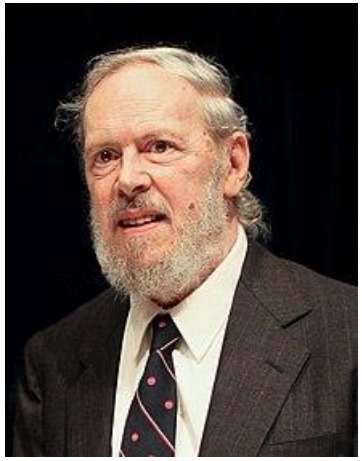
What We Will Learn

- What is the **C**
- Variables
  - Types
- Values
- Casting
- Constants & Definition


THE
C
ANSI
C
PROGRAMMING
LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

The C Language

- *C* is a *general-purpose* programming language
- *C* is developed by *Dennis Ritchie* at *Bell Laboratories* (1972) – Now C18



- *C* is one of the widely used languages
  - Application development
  - System programs, most operating systems are developed in C: **Unix**, **Linux**
  - Many other languages are based on it

Programming in C Language

- **C** programming language
  - A set of notations for representing programs
- **C** standard libraries
  - A set of developed programs (functions)
- **C** programming environment
  - A set of tools to aid program development

The First Example

- Write a program that prints

  "Hello the CE juniors :-)"

The First C Program

```
#include <stdio.h>


int main(void){
  printf("Hello the CE juniors :-) \n");
  return 0;
}
```

General Rules

- C is case sensitive: **`main`** is not **`MaIn`**
- A "**;**" is required after each statement
- Each program should have a **`main`** function

```
int main(void){…
void main(void){…
main(){…
int main(int argc, char ** argv){…
```

  - Program starts running from the main
- You should follow *coding styles* (beautiful code)

General Rules: Spaces

Equal Statements

`int main(void){`	`int main ( void) {`
`printf("abc");` `return 0;`	`printf ( "abc" ); return 0;`
`return 0;`	`return 0;`

General Rules: Spaces

Not Equal Statements

`int main(void){`	`intmain(void) {`
`printf("abc def");`	`printf("abcdef");`

Comments

```
/* Our first


C program */
#include <stdio.h>
int main(void){
  //This program prints a simple message
  printf("Hello the CE juniors :-) \n");
  return 0;
}
```

The First C Program

- You should
  - Develop the source code of program
  - Compile
  - Run
  - Debug
- All of them can be done in IDE
  - Code::Blocks, Dev-C++
  - CLion
  - VS Code, Eclipse,

What We Will Learn

- What is the C
- Variables
  - Types
- Values
- Casting
- Constants & Definition

Variables

- "write a program to calculate the sum of two numbers given by user"
- Solving problems
  - Input data → Algorithm → Output date
- What we need
  - Implementing the algorithm
    - Named Functions
    - We will discuss later
  - Storing the input/output data
    - Variables

Variables (cont'd)

- Data is stored in the main memory

- Variables
  - Are the name of locations in the main memory
    - We use names instead of physical addresses
  - Specify the coding of the location
    - What do the "01"s means?
    - What is the type of data?

Variables

- Variables in the C

  <Qualifier> <Type> <Identifier>;

- <Qualifier>
  - Is optional
  - We will discuss later
- <Type>
  - Specifies the coding
- <Identifier>
  - Is the name

Types: Integers

- Integer numbers
  - Different types, different sizes, different ranges

Type	Size	Unsigned	Signed
`short`	16Bits	$[0, 2^{16} - 1]$	$[-2^{15}, 2^{15} - 1]$
`int`	32Bits	$[0, 2^{32} - 1]$	$[-2^{31}, 2^{31} - 1]$
`long` or `long int`	32/64 Bits	$[0, 2^{32\|64} - 1]$	$[-2^{31\|63}, 2^{31\|63} - 1]$
`long long` or `long long int`	64 Bits	$[0, 2^{64} - 1]$	$[-2^{63}, 2^{63} - 1]$

Types: Float and Double

- Floating point number
  - float 32 bits
  - double 64 bits
  - long double 96 bits

- Limited precision
  - float: 8 digits precision
    - 1.0 == 1.00000001
  - double: 16 digits precision
    - 1.0 == 1.0000000000000001

Types: Char

- Character
  - Type: **char**
- Single letters of the alphabet, punctuation symbols
- Should be single quotation
  - ‘a’, ‘^’, ‘z’, ‘0’, ‘1’, ‘\n’, ‘\”, ‘\0’

Types: Booleans

- `#include <stdbool.h>`
- Logics (Boolean): **`bool`**
- Only two values: **`false`** , **`true`**

Signed and Unsigned Types

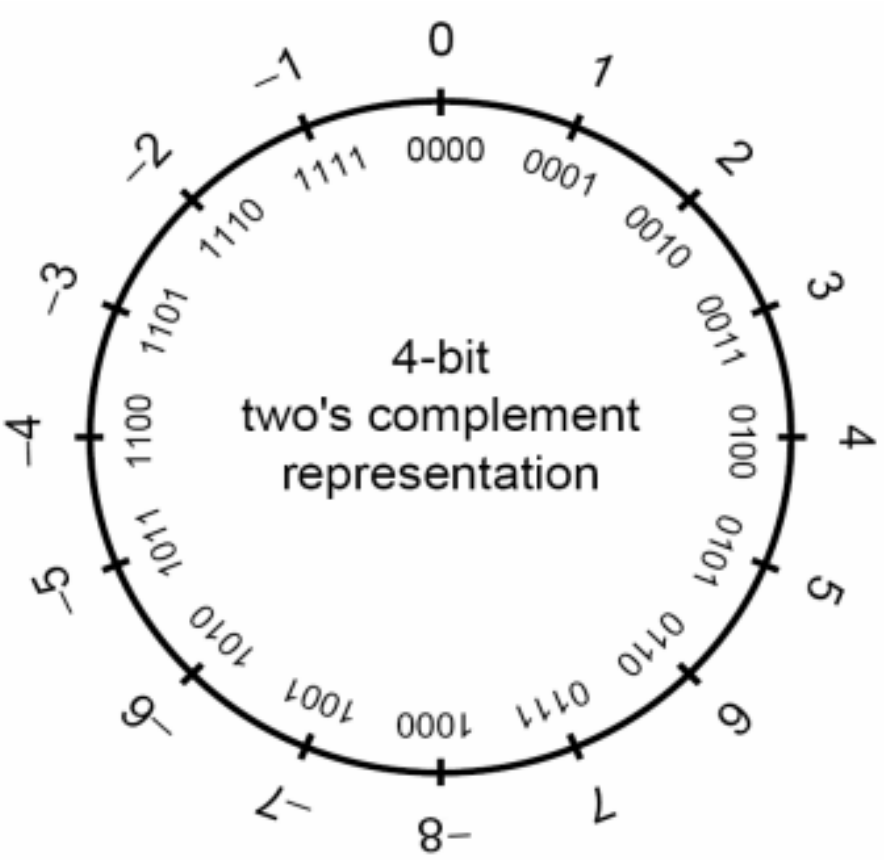
- Integers in **C** and **C++** are either signed or unsigned.
- For each signed type there is an equivalent unsigned type.

Signed Integers

- Signed integers are used to represent positive and negative values.
- On a computer using two's complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.

Signed Integer Representation

Tow's Complement (ranges from -2^{n-1} through $2^{n-1}-1$).

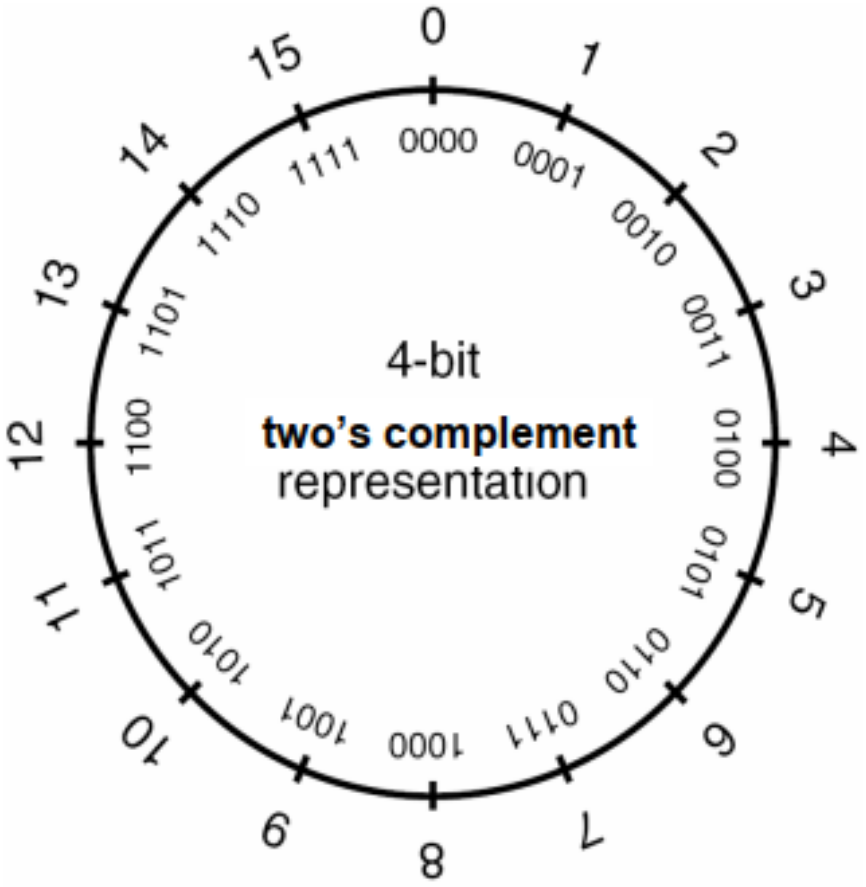


Unsigned Integers

- Unsigned integer values range from zero to a maximum that depends on the size of the type
- This maximum value can be calculated as 2^n-1, where *n* is the number of bits used to represent the unsigned type.

Unsigned Integer Representation

Tow's complement (ranges from 0 through $2^n - 1$)

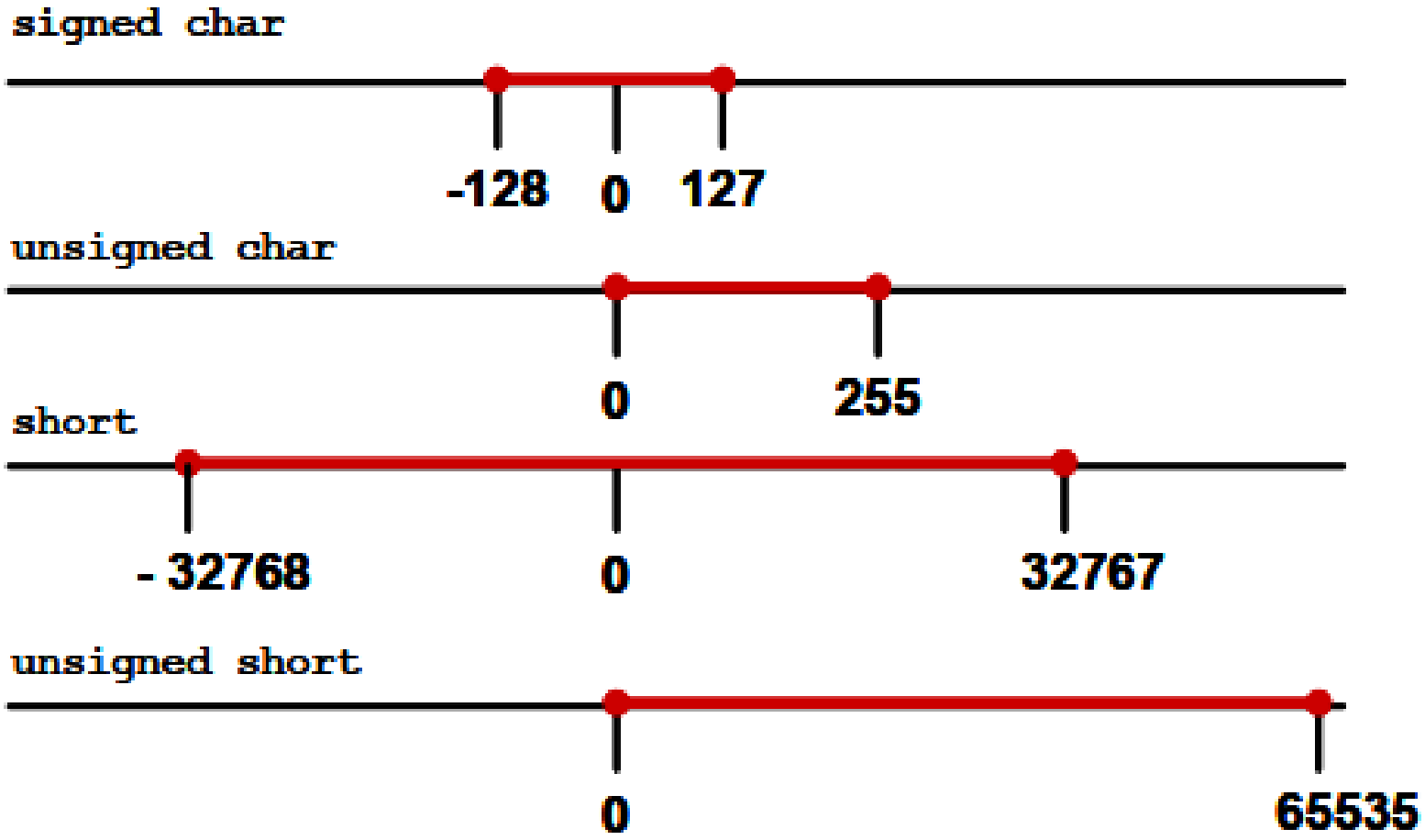


Integer Ranges

- **Minimum** and **maximum** values for an integer type depend on
  - The type's representation
  - Signedness
  - The number of allocated bits
- The **C99** standard sets minimum requirements for these ranges.

Example Integer Ranges

- Char in C is a 1-byte integer.



Signed / Unsigned Characters

The type `char` can be **`signed`** or **`unsigned`**.

- When a **`signed char`** with its high bit set is saved in an integer, the result is a negative number.
- Use **`unsigned char`** for buffers, pointers, and casts when dealing with character data that may have values greater than **127** (**0x7f**).

Overflow and Underflow

- All types have limited number of bits
  - Limited range of number are supported
  - Limited precision
- Overflow
  - Assign a very big number to a variable that is larger than the limit of the variable.
- Underflow
  - Assign a very small number to a variable that is smaller than the limit of the variable.

Example

Overflow Examples

- Example of **signed** and **unsigned** integer overflows:

```
1. int i;
2. unsigned int j;
3. i = INT_MAX;  // 2,147,483,647
4. i++;
5. printf("i = %d\n", i);
6. j = UINT_MAX; // 4,294,967,295;
7. j++;
8. printf("j = %u\n", j);
```

```
#include <limits.h>
```

i=-2,147,483,648

j = 0

Underflow Examples

- Example of **signed** and **unsigned** integer underflows:

```
 9. i = INT_MIN; // -2,147,483,648;
10. i--;
11. printf("i = %d\n", i);

12. j = 0;
13. j--;
14. printf("j = %u\n", j);
```

```
i = 2,147,483,647
```

```
j = 4,294,967,295
```

Variables: Identifier

- The name of variables: identifier
- Identifier is a string (single word) of
  - Alphabet
  - Numbers
  - "_"
- But
  - Cannot start with digits
  - Cannot be the key-words (reserved words)
  - Cannot be duplicated
  - Should not be library function names: `printf`

Variables: Identifier

- Use readable identifiers:
  - Do not use `memorystartaddress`
    - Use `memory_start_address`
  - Do not use `xyz`, `abc`, `z`, `x`, `t`
    - Use `counter`, `sum`, `average`, `result`, `parameter`, …
  - Do not be lazy
    - Use meaningful and readable names

C reserved words

- Cannot be used for identifiers

_Bool	default	if	sizeof	while
_Complex	do	inline	static	
_Imaginary	double	int	struct	
auto	else	long	switch	
break	enum	register	typedef	
case	extern	restrict	union	
char	float	return	unsigned	
const	for	short	void	
continue	goto	signed	volatile	

C++ reserved words

- Cannot use for identifiers

```
asm             bool            catch           class
const_cast      delete          dynamic_cast    explicit
export          false           friend          inline
mutable         namespace       new             operator
private         protected       public          reinterpret_cast
static_cast     template        this            throw
true            try             typeid          typename
using           virtual         wchar_t
```

Variable Identifiers

- Example of **valid** identifiers
  - `student`
  - `Grade`
  - `sum`
  - `all_students`
  - `average_grade_1`

Variable Identifiers

- Example of **valid** identifiers
  - `student`
  - `Grade`
  - `sum`
  - `all_students`
  - `average_grade_1`
- Example of **invalid** identifiers
  - `if`
  - `32_test`
  - `wrong*`
  - `$sds$`

Variables: Declaration (اعلان)

- Reserve memory for variable: declaration
  - <type> <identifier>;
- A variable must be declared before use

```
char test_char;
int sample_int;
long my_long;
double sum, average, total;
int id, counter, value;
```

Variable Type Effect (in complied langs.)

- Important note: the type of variable is *NOT* stored in the main memory
  - After compiling the program → NO type is associated to memory locations!!!
- So, what does do the type?!
  - It determines the "*operations*" that work with the memory location
- *E.g.:*
  - `int x, y, z;` `z = x + y;`
  - `float a, b, c;` `c = a + b;`

Integer + and = Performed by ALU

Variable Type Effect (in complied langs.)

- Important note: the type of variable is *NOT* stored in the main memory
  - After compiling the program → NO type is associated to memory locations!!!
- So, what does do the type?!
  - It determines the "*operations*" that work with the memory location
- *E.g.:*
  - `int x, y, z;`
  - `float a, b, c;`

```
z = x + y;
c = a + b;
```

Integer + and = Performed by ALU

Float + and = Performed by FPU

Variables: Initial Values

- What is the initial value of a variable?
  - In C: we do not know.
  - In C: it is not 0.

We need to assign a value to each variable before use it.

What We Will Learn

- What is the C
- Variables
  - Types
- Values
- Casting
- Constants & Definition

Constants in C

- Values
  - Numeric
    - Integer numbers
    - Float numbers
  - Char
  - Strings
- Symbolic constant
- Constant variables

Values

- Variables
  - Save/restore data (value) to/from memory
- Declaration specifies the type and name (identifier) of variable
- Assigning value to the variable: assignment
  - <identifier> = <value>;
  - Compute the <value> and save result in memory location specified by <identifier>

Values: Examples

```
int i, j;
long l;
float f;
double d;

i = 10;
j = 20;
f = 20.0;
l = 218;
d = 19.9;
```

Value Types

- Where are the values stored?!

```
int x = 20;
x = 30 + 40;
```

  - In main memory
    - There is a logical section for these constant values

- So, we need to specify the type of the value
  - The coding of 01s of the value

- The type of value is determined from the value itself

Values (literals): Integers

- Valid integer values

  10, -20, +400; //**Decimal (base 10)** integer literal

  0x12A, 0X12A; //**Hexadecimal (base 16)** integer literal

  017; //**Octal (base 8)** integer literal

  5000L; // **long int** integer literal

- Invalid integer values

10.0, -+20, -40 0, 600,000, 5000 L, 019;

Binary-Hex and Hex-Binary: Examples

- HEX: base 16
  - The letters that stand for hexadecimal numbers above 9 can be upper or lower case – both are used.

- **More binary-hex conversions*:**
  - **101110100010 = 1011 1010 0010 = 0x BA2.**
  - **101101110.01010011 = (000)1 0110 1110 . 0101 0011 = 0x 16E.53.**
  - **1111111101.10000111 = (00)11 1111 1101 . 1000 0111 = 0x 3FD.87.**
- **To convert hex-binary, just go the other direction!**
  - **0x 2375 = (00)10 0011 0111 0101 = 10001101110101.**
  - **0x CD.89 = 1100 1101.1000 1001 = 11001101.10001001.**
  - **0x 37AC.6 = (00)11 0111 1010 1100.011(0) = 11011110101100.011.**
  - **0x 3.DCAB = (00)11.1101 1100 1010 1011 = 11.11011100101011.**

*** Note that leading zeroes are added or removed as appropriate in the conversion processes.**

Values (literals): Float and Double

- Valid numbers:

  0.2; .5; -.67; 20.0; 60e10; 7e-2

  12.5f; // float literal

  12.5L; // long double literal

- Invalid numbers:

  0. 2; 20. 0; 20 .0; 7 e; 6e; e12

Values (literals): Chars

- Char values
  - Should be enclosed in single quotation
  - ‘a’, ‘^’, ‘z’, ‘0’, ‘1’, ‘\n’, ‘\”, ‘\0’
- Each character has a code: ASCII code
  - ‘A’: 65; ‘a’: 97; ‘1’: 49; ‘2’: 50; ‘\0’ : 0
- Character *vs.* Integer
  - ‘1’ != 1 ; ‘2’ != 2
  - ‘1’ == 49 But 1 == 1

Values (literals): Strings

- String is a set of characters
  - Starts and ends with double quotation: "
- Examples:

```
"This is a simple string"

"This is a cryptic string #$56*(#"
```

Effect of Value Types

- The type of values have the same effect of the type of variables
  - It determines the "*operations*" that work on the values
- E.g.:
  - `int z;` `z = 10 + 20;`
  - `float c;` `c = 1.1 + 2.2;`

Integer + and = Performed by ALU

Float + and = Performed by FPU

Values: Initialization

```
int i = 20;

int j = 0x20FE, k = 90;

int i, j = 40;

char c1 = 'a', c2 = '0';

bool b1 = true;

float f1 = 50e4;

double d = 50e-8;
```

Values: From memory to memory

```
int i, j = 20;

i = j;              // i = 20

double d = 65536;  // d = 65536.0

double b = d;       // b = 65536.0

d = b = i = j = 0;

// j = 0, i = 0, b = 0.0, d = 0.0
```

Basic Input Output

- To **read** something: **scanf**
  - Integer: `scanf("%d", &int_variable);`
  - Float: `scanf("%f", &float_variable);`
  - Double: `scanf("%lf", &double_variable);`

- To **print (show)** something: **printf**
  - Integer: `printf("%d", int_variable);`
  - Float: `printf("%f", float_variable);`
  - Message (string literal): `printf("message");`

What We Will Learn

- What is the C
- Variables
  - Types
- Values
- Casting
- Constants & Definition

Casting

- What is the casting?
  - When the type of variable and value are not the same
  - Example: Assigning double value to integer variable
- It is not a syntax error in C (only warning)
  - But can cause runtime errors
- It is useful (in special situations)
  - But we should be very very careful

Implicit casting

- Implicit (ضمنی)
  - We don't say it
  - But we do it

```
char f2 = 50e6; /* cast from double to char */

int i = 98.01;  /* cast from double to int */
```

Explicit casting

- Explicit (صریح)
  - We say it
  - And we do it

```
int i = (int) 98.1; /* Cast from double to  int */

char c = (char) 90; /* Cast from int to char */
```

Casting effects

- Casting from small types to large types
  - There is not any problem
  - No loss of data

```
int i;
short s;
float f;
double d;
s = 'A';      // s = 65
i = 'B';      // i = 66
f = 4566;     // f = 4566.0
d = 5666;     // d = 5666.0
```

Casting effects (cont'd)

- Casting from large types to small types
  - Data loss is possible
    - Depends on the values

```
float f = 65536;      // 65536.0
double d = 65536;        // 65536.0
short s = 720;           // 720

char c = (char) 65536;      // c = 0
short s = (short) 65536;    // s = 0
int i = 1.22;           // i = 1
int j = 1e23;           // j = ???
```

Casting effects (cont'd)

- Casting to Boolean
  - If value is zero → false
  - If values is not zero → true

```
bool b2 = 'a', b3 = -9, b4 = 4.5;   // true
bool b5 = 0, b6 = false; b7 = '\0'; // false
```

Truncation Errors

- Truncation errors occur when
  - an integer is converted to a smaller integer.
  - type and the value of the original integer is outside the range of the smaller type.
- Low-order bits of the original value are preserved and the high-order bits are lost.

Truncation Error Example

```
1. char cresult, c1, c2, c3;
2. c1 = 100;
3. c2 = 90;
4. cresult = c1 + c2;
```

Adding `c1` and `c2` exceeds the max size of `signed char (+127)`

Truncation occurs when the value is assigned to a type that is too small to represent the resulting value

Integers smaller than `int` are promoted to `int` or `unsigned int` before being operated on

Sign Errors

- Can occur when
  - converting an **unsigned** integer to a **signed integer**.
  - converting a **signed** integer to an **unsigned integer**.

Sign Error Example

```
1. int i = -3;

2. unsigned short u;

3. u = i;

4. printf("u = %hu\n", u);
```

Implicit conversion to smaller unsigned integer

There are sufficient bits to represent the value so no truncation occurs. The two's complement representation is interpreted as a large signed value, however, so **u = 65533**.

What We Will Learn

- What is the C
- Variables
  - Types
- Values
- Casting
- Constants & Definition

Constant Variables!!!

- Constants
  - Do not want to change the value
  - Example: pi = 3.14
- We can only *initialize* a constant variable
  - We MUST initialize the constant variables (why?!)
- **`const`** is a qualifier

```
const int STUDENTS = 38;
const long int MAX_GRADE = 20;
int i;
i = MAX_GRADE;
STUDENTS = 39; //ERROR
```

Definitions

- Another tool to define constants
  - Definition is not variable
    - We define definition, don't declare them
  - Pre-processor replaces them by their values before compiling

```
#define STUDENTS  38
int main(void){
   int i;
   i = STUDENTS;
   STUDENTS = 90; //ERROR! What compiler sees: 38 = 90
```

Definitions

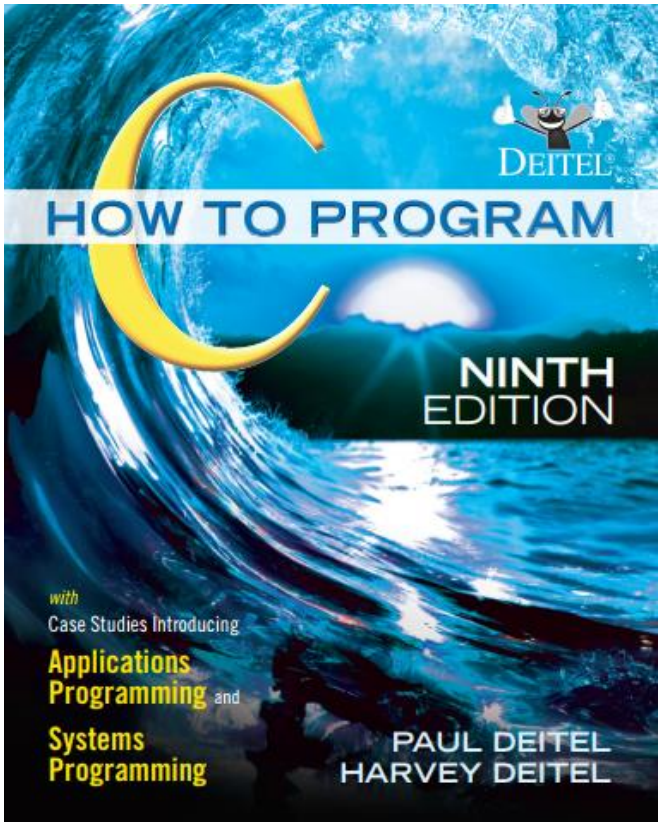
```
#define NAME “Test”

#define AGE (20 / 2)

#define MIN(a, b) (((a)<(b))?(a):(b))

#define MAX(a, b) (((a)>(b))?(a):(b))

#define MYLIB
```

Summary

- Simple programs in C
- Two basics
  - Variables
    - Types
  - Values
    - Types
- Casting
  - The type mismatch
- Constant variables & definitions

Reference

- Reading Assignment: **Chapter 2** of "C How to Program"



Questions

- Which of the following statements about C is FALSE?

  A) C is case-sensitive.

  B) The main function is optional in every program.

  C) Statements must end with a semicolon.

  D) Program execution starts with the main function

- Answer: B

Questions

- What is the size of an int data type in C on most systems?

  A) 16 bits B) 32 bits

  C) 64 bits D) Depends on the system

- Answer: D

- What is the correct format specifier for reading an integer value using scanf?

  A) %i B) %d C) %f D) %c

- Answer: B

Questions

- Which of the following scenarios would likely result in data loss during casting?

  A) Casting a double to float

  B) Casting a float to int

  C) Casting an int to char

  D) All of the above

- Answer: D