CENG 140 – C Programming

Introduction

Asst. Prof. Selim Temizer
C Programming Language - Remarks

- C is a simple language, very few commands, small and easy to learn
  - Just 5-6 pages of specification/grammar
- C is a powerful language
  - Enables you to directly access hardware
  - Language of Operating System development
- “Number of statements / Work done by CPU” ratio is high
  - You have to tell everything to the computer
  - A little lower level when compared to some other languages
  - Compare with Java, Python, Matlab, Scripting Languages
    - Most interpreted languages usually have higher ratios
Our Learning Goals for CENG 140

• C language is very helpful in understanding how computers work
• We will study how our C programs are processed by computers
  • Think like the compiler / computer
• At MIT, corresponding course is 6.001 SICP
  • Structure and Interpretation of Computer Programs
  • They used Lisp, Java, Python, ... as a tool to study structure & interpretation
• We will learn
  • C programming language, professionally
  • Structure of C programs
  • How computers execute our C programs
  • Compiled languages behave similarly in general, so our benefit is more than C
My Teaching Observations

- Taught C to high school students preparing for coding competitions
  - They liked visual descriptions

- Taught C to Food Engineering students (CENG 230)
  - They ranked 2\textsuperscript{nd} after EEE students (with good attendance to lectures)
  - They ranked 7\textsuperscript{th} or 8\textsuperscript{th} (with poor attendance to lectures)

- Taught C to co-workers at IT and Defence companies
  - They liked visual examples of how computers execute our programs
First Step – Let’s Talk about Computers (From a Programmer’s Point)

• Everything is composed of 0’s and 1’s
  • Binary number system (cf. decimal, octal, hexadecimal number systems)
  • Bit, byte, word, double word
  • How to represent:
    • integers, large integers, negative numbers (sign bit, one’s complement, two’s complement), real numbers, letters, musical notes, machine language commands
  • How to store in memory, how to transmit
    • Big-endian
    • Little-endian
  • Reading: Appendix D & E

• Components of a computer
  • CPU
    • 32 bit, 64 bit
  • Memory (RAM)
    • Transistors (on/off)
    • Stream of bytes
  • Storage (Hard Disk)
    • Head, sector, track, ferromagnetic material
    • Stream of bytes
  • Keyboard/Terminal (Input)
    • Stream of bytes, buffered
  • Screen/Terminal (Output)
    • Stream of bytes, buffered/unbuffered
  • Other peripherals
Chapter 1, Part 1 – Computer Hardware

http://www.c-jump.com/CIS24/Slides/Hardware/lecture.html

https://www.amazon.com/PC-Intern-Encyclopedia-Programming-Developers/dp/1557553041
Computer Hardware - Notes

• Everything is stored as bits and bytes
  • CPU registers: specialized, fastest, electronic storage
  • Memory (RAM): specialized, fast, electronic storage
  • Floppy Disk, Hard Disk: slow, electro-mechanic, permanent storage
  • Flash Disk: slow, electronic, permanent storage

• Size of registers in CPU
  • Natural size of data that can be processed by CPU in a single time unit
  • 32-bit, 64-bit architectures
  • The size of a “word” usually corresponds to the natural size of an architecture

• Bus
  • Address Bus: Size is large enough to address all bytes in memory
  • Data Bus: Size is usually equal to the architecture’s natural size
Chapter 1, Part 2 - Software

• Computer Programs
  • Operating Systems (Linux, Unix, Windows, DOS, etc.)
  • Application Programs (Games, MS Office, Web Browsers, etc.)

• Programming Languages
  • Machine language (specific to the hardware)
  • Assembly language (using assemblers)
  • High level languages
    • Compiled languages (using compilers, for example C/C++)
    • Interpreted languages (using interpreters, for example PHP, Perl, Matlab)
    • Compiled halfway to bytecode, and then interpreted (for example Java, C#)
Levels of Programming Languages

High-level program

```java
class Triangle {
    float surface()
    return b*h/2;
}
```

Low-level program

```assembly
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Executable Machine code

```
0001001001000101
0010010011010100
10101101001...
```

Machine Code in Hex

| 27BB0001 | lda gp, main |
| 23BD8050 | lda gp, main |
| 23DEFFF0 | lda sp, -16(sp) |
| A61D8018 | ldq r16, 8(gp) |
| A77D8010 | ldq r27, printf |
| 47E0F411 | mov 7, r17 |
| B75E0000 | stq r26, (sp) |
| 6B5B4000 | jsr r26, printf |
| 27BA0001 | ldah gp, main |
| A75E0000 | ldq r26, (sp) |
| 23BD8050 | lda gp, main |
| 47FF0400 | clr r0 |
| 23DE0010 | lda sp, 16(sp) |
| 6BFA8001 | ret r26 |

High-Level Code

```c
main()
{
    int a, b, c;
    a = 3;
    b = 4;
    c = a + b;
    printf("\n%d\n", c);
}
```
C Language

• Designed by Dennis Ritchie, 1972
  • As the systems language for UNIX operating system
  • AT&T Bell Laboratories
  • Belongs to “Algol” family of programming languages

• History
  • Algol 60 (1960)
  • CPL (1963)
  • BCPL (1969)
  • B (1970)
  • C (1972)
  • ANSI C (1989)
Chapter 1, Part 3 – Programming Process

• Problem Definition
  • Purpose, Inputs, Outputs (Requirements Engineering)

• Program Design
  • Devise an algorithm (clearly defined steps to produce output from input)
  • Top-down design, stepwise refinement

• Program Coding
  • Express algorithm in a programming language
  • In C: each task can be implemented as a function
  • Structured programming: sequential, selective, repetitive control structures

• Program Compilation and Execution (We will run examples together)
  • Translate to machine code: (preprocess,) compile, link

• Program Testing and Debugging
  • Desk checking, compile-time errors, run-time errors, logical errors

• Program Documentation
  • Requirements, input/output, top-down design, algorithms, source, tests, user’s guide
  • Self-documenting, comments
GNU C Compiler : gcc

- **--help**: Display gcc help information on stdout (also: man gcc)
- **–E**: Stop after preprocessing (stdout: preprocessed c, extension: .i)
- **–S**: Stop after compiling (file: assembler code, extension: .s)
- **–c**: Stop after assembling (file: object code, extension: .o)
- **-o <file>**: place output in file
Chapter 2 – Sequential Structure

• Structure of a C program file
  • Building blocks: variables (memory storage, hold data) and functions (process data)
  • Standard Library and Third Party Libraries
  • Preprocessor directives and comments (handled by preprocessors)
  • Multiple-file C projects

• How to compile and run a C program
  • Preprocess (preprocessor, to pure C code)
  • Compile (compiler, to assembly language code)
  • Assemble (assembler, to object/machine code)
  • Link (linker, to executable file, with sections text, data, bss)
  • Load and start at main (loader and operating system)
    • Extra memory regions: stack, heap, command-line arguments and environment variables
Function Definitions - Syntax

```
ReturnType FunctionName ( Type1 Parameter1Name, Type2 Parameter2Name )
{
    Variable Definitions, External Variable/Function Declarations
        Type var1Name, var2Name, var3Name, ... ;
        extern Type globalVar1Name, globalVar2Name, ... ;
        extern ReturnType otherFunctionName ( Type1, Type2, Type3 ) ;
    Statements
        Expression1 ;
        Expression2 ;
        ...
}
```
Definition vs. Declaration

• Definition
  • Purpose: To create building blocks of applications (variables and functions)!
  • Variables: Storage space is allocated in memory, might also be initialized
  • Functions: Implementation is provided (to be compiled into machine code)

• Declaration
  • Purpose: To let the compiler know that a building block is created elsewhere
    (Otherwise, the compiler will be confused and may not be able to compile)
  • Variables: The storage is/will be created elsewhere, you may trust us to use it
  • Functions: The implementation is/will be created elsewhere, trust us to use it

• Attention:
  • Most people/books/articles use the two terms inconsistently!

Definition / Declaration Example

myLibrary.c

```c
int result;

int add ( int a , int b )
{
    return ( a + b );
}
```

myApplication.c

```c
#include <stdio.h>

extern int result;
extern int add( int , int ) ;

int main ( void )
{
    result = add( 3 , 5 ) ;
    printf( "Sum is equal to : %d\n" , result ) ;
    return 0;
}
```

• Compile one by one, and then link
  • gcc –Wall –c myLibrary.c
  • gcc –Wall –c myApplication.c
  • gcc –Wall myLibrary.o myApplication.o

• Or, compile together in a single command
  • gcc –Wall myLibrary.c myApplication.c
Notes on Functions - 1

• Functions might be defined to have parameters
  • When calling them, we pass arguments
  • Sometimes people use parameter / argument inconsistently
• Arguments can be any valid expressions
  • They are evaluated first, before the called function is invoked
  • Warning: The order of evaluation might be compiler dependent
    • Example: Unexpected behavior might happen in a call like \texttt{add( a++, ++a )};
    • A copy of their values are passed to the function (call-by-value / pass-by-value)
      • C language does NOT support call-by-reference / pass-by-reference mechanism

• Function name is the address of the starting byte of the function’s machine code in memory
Notes on Functions - 2

• Function **prototype**
  • `ReturnType FunctionName ( Type1 Param1Name, Type2 Param2Name )`
  • Parameter names are actually not needed

• Function **signature**
  • `FunctionName ( Type1 Param1Name, Type2 Param2Name )`
  • `ReturnType` is not included in signature
  • Parameter names are actually not needed

• Sometimes people use **prototype** / **signature** inconsistently
Comments

• Start with character sequence  /*
• End with character sequence  */
• Can be placed wherever a whitespace can be placed
• Simply filtered out by preprocessor/compiler
• Cannot be nested (filtering out is not a smart operation)
  • Filtering starts with /* and ends with the first occurrence of */
• Use comments to convey useful information to readers of your code
  • Nicely commented programs are easier to maintain by yourself and others
  • Don’t use non-informative comments, some useless examples:
    • int i ; /* This is a variable */ → Yes, it is. Trust me, I know!
    • /* Now I will implement the main function below */ → We see it, no need to tell!
Character Set

- Set of characters that may appear in a legal C program
  - Textbook, Section 2.2, Page 27
- Graphic characters
- Non-graphic characters
  - Escape sequences
  - Backslash followed by a letter
- Whitespace characters
  - Space, horizontal tab, vertical tab, newline (and carriage return), formfeed

<table>
<thead>
<tr>
<th>Table 2.1. C character set (graphic characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
</tr>
<tr>
<td>0-9</td>
</tr>
<tr>
<td>A-Z</td>
</tr>
<tr>
<td>a-z</td>
</tr>
<tr>
<td>!</td>
</tr>
<tr>
<td>`</td>
</tr>
<tr>
<td>#</td>
</tr>
<tr>
<td>$</td>
</tr>
<tr>
<td>%</td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td>'</td>
</tr>
<tr>
<td>(</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>;</td>
</tr>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>?</td>
</tr>
<tr>
<td>`</td>
</tr>
<tr>
<td>^</td>
</tr>
<tr>
<td>\</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td>_</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>\</td>
</tr>
<tr>
<td>\a</td>
</tr>
<tr>
<td>\b</td>
</tr>
<tr>
<td>\f</td>
</tr>
<tr>
<td>\n</td>
</tr>
<tr>
<td>\r</td>
</tr>
<tr>
<td>\t</td>
</tr>
<tr>
<td>\v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2.2. C character set (non-graphic characters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
</tr>
<tr>
<td>\a</td>
</tr>
<tr>
<td>\b</td>
</tr>
<tr>
<td>\f</td>
</tr>
<tr>
<td>\n</td>
</tr>
<tr>
<td>\r</td>
</tr>
<tr>
<td>\t</td>
</tr>
<tr>
<td>\v</td>
</tr>
</tbody>
</table>
Data Types

• Textbook, Section 2.3, Page 28

• char (signed, unsigned)
• int (short, long) (signed, unsigned)
• float
• double (long)
• void

• char : 1 byte
• sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
• Precision of float ≤ Precision of double ≤ Precision of long double
• Implementation specific bit widths in: limits.h, float.h
Data Types – Figure 1

- **User Defined**
  - Typedef
  - Enum

- **Primitive/Basic**
  - Integer Type
    - Int
  - Char

- **Derived**
  - Arrays
  - Structures
  - Union
  - Pointers

- **Float Type**
  - Float
  - Double

**DATA TYPES OF "C"**
Data Types – Figure 2

Data types

- Primitive type
  - char
  - int
  - float/double
  - void
- Derived type
  - signed
    - signed char
    - signed short
    - signed int
    - signed long
  - unsigned
    - unsigned char
    - unsigned short
    - unsigned int
    - unsigned long
- User defined type
  - Array
    - struct
    - union
    - enum
    - long double
Data Types – Figure 3

Basic Datatypes (Primary Datatypes)

- Integer
  - Signed
    - int
    - short int
    - long int
  - Unsigned
    - int
    - short int
    - long int
- Floating Point
  - float
  - double
  - long double
- Character
  - char
  - signed char
  - Unsigned Char
# Data Types – Figure 4

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Minimal Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>−127 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>8</td>
<td>−127 to 127</td>
</tr>
<tr>
<td>int</td>
<td>16 or 32</td>
<td>−32,767 to 32,767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16 or 32</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>signed int</td>
<td>16 or 32</td>
<td>Same as int</td>
</tr>
<tr>
<td>short int</td>
<td>16</td>
<td>−32,767 to 32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>16</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>signed short int</td>
<td>16</td>
<td>Same as short int</td>
</tr>
<tr>
<td>long int</td>
<td>32</td>
<td>−2,147,483,647 to 2,147,483,647</td>
</tr>
<tr>
<td>long long int</td>
<td>64</td>
<td>−(2(^{63}−1)) to 2(^{63}−1) (Added by C99)</td>
</tr>
<tr>
<td>signed long int</td>
<td>32</td>
<td>Same as long int</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>32</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>64</td>
<td>2(^{64}−1) (Added by C99)</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>1E−37 to 1E+37 with six digits of precision</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>1E−37 to 1E+37 with ten digits of precision</td>
</tr>
<tr>
<td>long double</td>
<td>80</td>
<td>1E−37 to 1E+37 with ten digits of precision</td>
</tr>
</tbody>
</table>
Variables and Constants

- A computer program manipulates two kinds of data – variables and constants. (Textbook, Section 2.4, Page 29)

- **Variables**: Programs use variables to store values
  - Every variable has a type
  - Must be defined/declared before used
  - Variable names obey identifier naming rules (next slide)

- **Constants**: An entity whose value does not change during execution
  - 5 types: integer, floating-point, character, string, enumeration
  - Enumeration is discussed in Chapter 12 of textbook
Identifiers (User Given Names)

• Names of variables, functions, definitions
• Must begin with a letter or underscore ( "_" )
• May contain letters, underscore, decimal digits ( 0-9 )
  • Corollary: Cannot contain whitespaces
• Case sensitive
• Significant length is compiler-dependent
  • ANSI C permits at least 31 significant characters
• Don’t use 32 standard keywords, standard library function names, etc.
  • List of standard keywords is in next slide
### 32 Standard Keywords in ANSI C

<table>
<thead>
<tr>
<th>auto</th>
<th>break</th>
<th>case</th>
<th>char</th>
<th>const</th>
<th>continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
<td>enum,</td>
<td>extern</td>
</tr>
<tr>
<td>float</td>
<td>for</td>
<td>goto</td>
<td>if</td>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td>register</td>
<td>return</td>
<td>short</td>
<td>signed</td>
<td>sizeof</td>
<td>static</td>
</tr>
<tr>
<td>struct</td>
<td>switch</td>
<td>typedef</td>
<td>union</td>
<td>unsigned</td>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constants

• Integer Constants
  • Decimal, Octal (0...), Hexadecimal (0x..., 0X...)
  • Type is normally `int`, if doesn’t fit then `long`
  • Possible suffix: u, U (unsigned), ul, UL (unsigned long)

• Floating-Point Constants
  • Written with a decimal point and/or in scientific notation (e, E)
  • Type is `double`
  • Possible suffixes: f, F (float), l, L (long double)
Constants

• Character Constants
  • Single character within apostrophes: ‘a’, ‘B’, ‘?’
  • Octal or Hexadecimal byte-sized bit pattern: ‘\ooo’, ‘\\xhh’
  • Null character (zero): ‘\0’
  • Type is int (ASCII or EBCDIC code)
  • Escape character: backslash (\)

• String Constants
  • Zero or more characters (may also be non-graphic) within double quotation marks
  • String can be continued by putting a backslash at the end of the line
  • Adjacent string constants are concatenated at compile time
  • Escape character: backslash (\)
  • Compiler automatically puts ‘\0’ at the end of each string (storage = chars + 1 byte)
  • ‘Z’ is not the same as “Z”
  • Type is address (char *, pointer, address of the first byte of the string)
Arithmetic Operators

• Operator: A symbol that causes specific mathematical or logical manipulations to be performed

• Unary: +, -, increment (++), decrement (--) 

• Binary: +, -, *, /, %

• Integer division is truncated
  • 8 / 3 will give you an integral result → 2
  • 8.0 / 3 or 8 / 3.0 or 8.0 / 3.0 will give you a real number → 2.666667

• Remainder (%) operator takes only integral operands
Expressions and Statements

• Expression: A combination of
  • Constants
  • Variables
  • Operations (arithmetic, logic, …)
  • Function calls
  • …

• All expressions have a value
  • To compute the value, refer to precedence and association rules (Appendix B)
  • Or use balanced parentheses

• Almost everything in C is an expression

• Statement (roughly): An expression followed by a semicolon
Precedence and Associativity of Operators

- The C Programming Language
  - ANSI C, 2nd Edition
  - Kernighan & Ritchie
  - Page 53, Table 2-1

![Operator Precedence Diagram]

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>ASSOCIATIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>~ + + -- + - * &amp; (type)</td>
<td>right to left</td>
</tr>
<tr>
<td>sizeof</td>
<td>left to right</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td>!</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>:= = = : /= %= &amp;= ^=</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &lt;= &gt;= &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>
Assignment Statements

• Assignment **expression**: variable = **expression**
• Assignment **statement**: assignment expression followed by semicolon
• Value of **expression** is assigned to the storage space (box) of variable
• Left side can only be an “L-Value”
  • A variable can be used as an l-value (box) or an r-value (the value in the box)
  • Constants are always r-values
• **Value of assignment expression**: whatever value is stored in variable
  • Assignment expressions can be chained and can be used in other expressions
• **x = y** has a different meaning/result than **y = x**
Increment (++) and Decrement (--) Operators

• Unary, Operates on a variable, that is, requires an L-Value (box)
  • For example the following usage is wrong (compile-time error):  \( a = ++b++ \);

• Prefix usage (increment/decrement before using)
  • Example: \( \text{int } i = 4; \text{ printf} \left( \text{ “%d” , ++i } \right); \)
  • First increment/decrement the variable (and then use the resulting value)
  • The value of the expression is the incremented/decremented value

• Postfix usage (increment/decrement after using)
  • Example: \( \text{int } i = 4; \text{ printf} \left( \text{ “%d” , i++ } \right); \)
  • The value of the expression is the original value (use the original value)
  • Then, as a side effect, increment/decrement the variable
Increment (++) and Decrement (--) Operators

• Examples
  • int i = 8; int k = ++i;
    • /* i becomes 9, k is assigned 9 */
  • int i = 8; int k = i++;
    • /* k is assigned value of expression (8), i becomes 9 */
  • int x = 3; int y = 5; int k = --x + y--;
    • /* x becomes 2, k is assigned 7 (2+5), y becomes 4 */

• Warning
  • Since the increment/decrement operator modifies its operand, use of such an operand more than once within the same expression can produce unexpected results.
  • int a = 10; int b = ++a + ++a + ++a; /* b might be assigned 39 or 37 */
    • /* ++ has higher precedence, a might be incremented 3 times first, then summed up */
Compound Assignment Operators

• C provides 10 compound assignment operators (for compact notation)
  • +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=  

• Semantics
  • variable operator= expression is equivalent to
  • variable = variable operator ( expression )

• Value of compound assignment is equal to whatever value is assigned to variable

• Example
  • i *= a + 1 ; is equivalent to i = i * (a + 1) ;
Nested Assignments

• C permits multiple assignment expressions in a single statement
  • Since assignment expressions have values, they can be used like any \texttt{r-value}

• Assignment operators are right-associative
  • \texttt{i = j = k = 0;} is interpreted as \texttt{i = ( j = ( k = 0 ) );}

• Examples from textbook
  • \texttt{i += j = k;} is treated as \texttt{i += ( j = k );}
  • \texttt{i = j += k;} is treated as \texttt{i = ( j += k );}
#include <stdio.h>

- Pure C language does not have language constructs for input/output

- ANSI C has a rich standard library
  - We will use `printf` and `scanf` functions from the standard library
  - Their definitions are in the compiled standard library
  - Their declarations are in `stdio.h` header file
• Example
  • int a = 4;
  printf("a = %d\n", a);

• printf( control string , arg1 , arg2 , ... ) ;
  • Ordinary characters are sent to output without any modification
  • Escape sequences may be used (like ‘\n’ , ‘\t’ , ... )
  • Conversion specifications (placeholders) are replaced with argument values
  • Returns number of characters written (its value is an int)
Scanf

d, i A decimal integer is expected in the input. The corresponding argument should be a pointer to an int.

f, e A floating-point number is expected in the input. The corresponding argument should be a pointer to a float. The input can be in the standard decimal form or in the exponential form.

c A single character is expected in the input. The corresponding argument should be a pointer to a char. Only in this case, the normal skip over the whitespaces in input is suppressed.

- Example
  - int a;
  - scanf( "%d", &a );

- scanf( control string , arg1 , arg2 , ... );
  - Space or whitespace within control string matches with whitespace in the input
  - Even if no space in control string, skips over whitespaces in the input to match next data item (unless reading a char and unless no corresponding whitespace in control string)
  - Ordinary characters are expected to be matched from the input (uncommon)
  - Requires addresses of (pointers to) variables (storage boxes) to store the value in
  - If the input buffer is empty, lets the user enter data, otherwise uses data in buffer
  - Returns number of arguments successfully read (its value is an int)
Type Conversion

• An expression may contain variables and constants (and calls to functions that return values) of different types.
  • Example: $8 + 3.14 / 2.f$ (integer, double, float)

• Mechanisms of evaluating mixed type expressions
  • Automatic Type Conversion
  • Explicit Type Conversion
  • Type Conversion in Assignments
Automatic Type Conversion

• ANSI C performs all arithmetic operations with just 6 data types:
  • int, unsigned int, long int, float, double, long double
• char and short is implicitly converted to int before an operation
  • This is called automatic unary conversion (always performed)
  • char → int conversion is machine dependent (signed? or unsigned?)
• After unary conversions, automatic binary conversions are applied
  • Lower type is promoted to higher type before the operation proceeds
  • Performed when necessary
• Textbook, Figure 2.4, Page 46
Explicit Type Conversion

• We can force a conversion that is different than automatic conversion:
  • Syntax: `(cast-type) expression`

• Cast is unary operator, has same precedence as other unary operators

• Examples:
  • (int) 12.8 → 12
  • (int) 12.8 * 3.1 → 12 * 3.1 → 37.2
  • (int) (12.8 * 3.1) → (int) (39.68) → 39
  • 3 / 2 → 1
  • (float) 3 / 2 → 1.5
Type Conversion in Assignments

• variable = expression  (or compound assignment operators)

• **Type of expression’s value** is automatically converted to **type of variable**

• Conversion of lower-order type to higher-order type is OK
  • Only the form of representation changes

• Conversion of higher-order type to lower-order type may cause
  • Truncation
  • Loss of information
Simple Macros

• Preprocessor directive #define
• Syntax:    #define macroName sequenceOfTokens
• Macros can also have parameters (will be covered later in the course)
• Preprocessor simply finds and replaces macroName with sequenceOfTokens
  • Exception: in comments and in string constants
  • Be careful about the final text after replacement!
    • Example: #define FIVE 2+3       ( What is the result? :    FIVE * 2 )
    • Suggestion: Always put sequenceOfTokens inside a pair of parantheses
      • #define macroName ( sequenceOfTokens )
• Previously defined macros can be used in new macros
• Can appear anywhere in the program
  • Usually collected together at the beginning of the program
• Convention: Macro names are written in uppercase letters
Simple Macros - Example

• If we bury important values in code
  • Difficult to understand by others (what are those magic numbers?)
• Instead: define symbolic constants (like PI)

```c
#include <stdio.h>
#define PI 3.141592

void main (void)
{
    float radius ;

    printf( "Enter radius : " ) ;
    scanf ( " %f" , &radius ) ;
    printf( "Area of circle = " , PI * radius * radius ) ;
}
```
Chapter 3 – Selective Structure

- Selective structures allow us to test for a condition, and follow alternative paths depending on the result of the test.
Example

```c
#include <stdio.h>

#define PASSING_GRADE 60

void main (void)
{
    int grade;

    printf( "Enter grade : " );
    scanf ( " %d" , &grade );

    if ( grade >= PASSING_GRADE )
        printf( "Congratulations! You passed the course!\n" );
    else
        printf( "Sorry! You failed the course!\n" );
}
```
Chapter 3 - Outline

• Relational Operators ( < , <= , > , >= , == , != )
• Logical Operators ( && , || , ! )
• Conditional Expression Operator ( ? : )
• Conditional Statements ( if , if else )
• Nested Conditional Statements ( if ... if else ... else ... if else )
• Multiway Conditional Statement ( if ... else if ... else if ... else )
• Constant Multiway Conditional Statement ( switch )
About Boolean Values

• The result of testing a condition may be true or false
• C language does NOT have Boolean values (true, false)
• Integers are used as substitutes for Boolean Values
  • Non-zero values (attention: positive and also negative) are interpreted as true
  • Zero is interpreted as false
• Example:
  • if ( 0 ) ... → false, the body of the if statement does NOT execute
  • if ( 2.5 ) ... → true, the body of the if statement executes

• Value of a true expression is 1, false expression is 0
  • printf( “value of expression = %d\n” , ( 2 > 3 ) ) ;
About Compound Statements (Blocks)

• A **block** is a sequence of **variable declarations/definitions** (at the top) and **statements** (at the bottom) enclosed **within braces**.

• A block can be placed wherever a single statement can be placed
  • No semicolon after closing brace of block

• Some common usages:
  • Grouping statements
    • Fitting more than one statement where only a single statement is allowed
    • Example: Body of if statement, body of loops
  • Opening up a new scope that is enclosed within the current scope
    • Also enables us to define variables, even though we may have statements before in code

```c
if ( grade >= 60 )
{
    saveGradeInDatabase( grade ) ;
    printf( "Congratulations!\n" ) ;
    printf( "You passed the course.\n" ) ;
}
```
Relational Operators

• There are 6 relational operators (Textbook, Table 3.1, Page 69)
• Value of a relational expression is of type int ( 0 or 1 )
• Comparing characters means comparing their ASCII / EBCDIC codes
  • Example: ‘b’ > ‘a’
• Common programming error:
  • Confusing == with =
    • if ( x == 10 ) ... → compares
    • if ( x = 10 ) ... → always true!
  • Suggestion: Put r-value first
    • if ( 10 == x ) ... → compares
    • if ( 10 = x ) ... → compile-time error
Logical Operators

- There are 3 logical operators (Textbook, Table 3.3, Page 71)
- Value of a logical expression is of type int (0 or 1)
- Logical AND: expr1 && expr2
  - First expr1 is evaluated, if zero (false) then SHORT CIRCUITED, result is zero
  - Otherwise, expr2 is evaluated
  - Example: if ( (1>5) && (2/0 >= 3) ) … → false, division by zero not executed
- Logical OR: expr1 || expr2
  - SHORT CIRCUIT evaluation like logical AND
- Logical NOT: ! expr
  - 0 if expr is nonzero (true), 1 if expr is zero (false)
About Evaluation of Logical Expressions

• For all expressions we use: Precedence and Associativity Table

• Attention (exception)
  • Logical AND and logical OR are always evaluated from left to right (and short circuited)
  • Be careful when expression contains a side effect
  • Good example: Textbook, Page 76
  • Suggestion: Always use parentheses to clearly express the exact order of operations you want
Conditional Expression Operator

• The only ternary operator in C language (takes 3 arguments)
• Syntax: `expr1 ? expr2 : expr3`
• Value of conditional expression:
  • First `expr1` is evaluated
  • If nonzero (true), `expr2` is evaluated, and also becomes the value of the whole
  • If zero (false), `expr3` is evaluated, and also becomes the value of the whole
• Example:
  • `larger = x > y ? x : y ; → OK`
  • `c ? x=a : x=b → will be interpreted as ( c ? x=a : x ) = b and is illegal (r-value)`
• Suggestion: Use proper parentheses to avoid unintended errors
Conditional Statements

• If statement
  • Syntax: `if ( expression) statement`
  • Expression is evaluated
    • If nonzero (true), execute statement
    • Otherwise, don’t execute statement
  • A block can be used as statement
  • Flowchart:

```c
#include <stdio.h>

int maximum ( int a , int b )
{
    if ( a >= b )  return a ;
    return b ;
}

void main ( void )
{
    int x , y ;
    scanf ( " %d %d" , &x , &y ) ;
    printf( "Maximum of them = %d\n" , maximum(x,y) ) ;
}
```
Conditional Statements

• If-else statement
  • Syntax: `if ( expression) statement1 else statement2`
  • Expression is evaluated
    • If nonzero (true), execute statement1 (if block)
    • Otherwise, execute statement2 (else block)
  • Block(s) can be used as statement(s)
  • Flowchart:

```c
if ( money >= 25.0 )
{
    printf( "You have enough money!\n" );
    printf( "You can buy a good meal.\n" );
}
else
    printf( "Sorry, not enough money!\n" );
```

Suggestion for if and else bodies (conditional statements):

It is good practice to use curly braces even if you have a single statement. Also easier to understand code when reading.
Nested Conditional Statements

• If-blocks and else-blocks can also contain other if, if-else statements

• There is no limit on the depth of nesting

```c
if ( undergraduate == 1 )
    if ( grade >= 60 )
        printf( "Passed!\n" );
    else
        printf( "Failed!\n" );
else
    if ( grade >= 70 )
        printf( "Passed!\n" );
    else
        printf( "Failed!\n" );
```
Sequence of Nested If Statements

- The nested conditional statement on the left is equivalent to the statement on the right
  - C language guarantees short-circuited left-to-right evaluation of logical expressions

```c
if ( undergraduate == 1 )
    if ( grade >= 60 )
        printf( "Passed!\n" );
```

```c
if ( expression-1 && expression-2 && ... && expression-n )
    statement
```

```c
if ( expression-1)
    .
    .
    if ( expression-n )
        statement
```

```c
if ( undergraduate == 1 && grade >= 60 )
    printf( "Passed!\n" );
```
Dangling Else Problem

Some if statements have else parts, some don’t. Which if statement does and else part belong to?

Example: Intention (on left), reality (on right)

How does the compiler resolve?
  • Else is associated with the closest previous else-less if

Suggestion: Always use braces (blocks) to avoid mistakes

Good examples on Textbook, Page 85-86

Ceng 140 - Section 2

Selim Temizer - Lecture Notes
A special form of nested conditional statements occurs frequently in practice: cascaded *if-else* statements where each *if-else* statement but the last one has another *if-else* statement in its *else* block.

Conditional expressions are evaluated in order.

- If one is found to be true, the statement associated with it is executed, and whole chain terminates.
- If none is true, the final *else*-block is executed.
Constant Multiway Conditional Statement

• General form of the switch statement:

```
switch ( expression )
{
    case value1 : statement1
        break ;
    case value2 : statement2
        break ;

    ... 

    case valueN : statementN
        break ;

    default     : statementD
        break ;
}
```

Roughly equivalent to:

```
if      ( expression == value1 )
    statement1
else if ( expression == value2 )
    statement2
...
else if ( expression == valueN )
    statementN
else
    statement
```
Constant Multiway Conditional Statement

- Expression must be an **integral** expression
- Case values must be **constant integral** expressions
- The use of **default** is optional
- **Default** (if used) does NOT need to be at the end
- **How does it work?**
  - First, expression is evaluated
  - Its value is successively compared with case values
  - If a case value matches the expression value:
    - Execution continues from the point labelled with the corresponding case
    - We use **break** statement to break out of the switch statement
      (otherwise all statements until the end of the switch statement will be executed)
  - If value does not match with any case values:
    - If there is a **default** case, execution continues from the point labelled with default
    - If there is no **default** case, execution continues with the statements coming after switch statement

```java
switch ( expression )
{
    case value1 : statement1
        break ;
    case value2 : statement2
        break ;
    ... }
case valueN : statementN
    break ;
default : statementD
    break ;
}```
Other Notes on Switch Statement

• Textbook, Page 91

No two case values should be the same in a switch statement. However, more than one case value can be associated with a set of statements by listing multiple case values before that particular set of program statements. It is not necessary to terminate each case with a break statement. Omitting the break statement from a particular case causes the execution to continue without regard for the case values and the default label. The break statement, although not logically necessary, should also be placed after the statement associated with default (or the last case value, if default is not at the end) as a matter of good programming practice. This practice avoids inadvertent introduction of a bug in the program when a case value and associated statements are added later on at the end of a switch statement.
```c
#include <stdio.h>

void printDay ( int dayNo )
{
    switch ( dayNo )
    {
    case 1 : printf( "Monday" ) ;
             break ;
    case 2 : printf( "Tuesday" ) ;
             break ;
    case 3 : printf( "Wednesday" ) ;
             break ;
    case 4 : printf( "Thursday" ) ;
             break ;
    case 5 : printf( "Friday" ) ;
             break ;
    case 6 : printf( "Saturday" ) ;
             break ;
    case 7 : printf( "Sunday" ) ;
             break ;
    default : printf( "INCORRECT VALUE" ) ;
    }
}

void main ( void )
{
    int day ;

    printf ( "Enter day no : " ) ;
    scanf ( " %d" , &day ) ;
    printDay( day ) ;
    printf ( "\n" ) ;
}
```
About Math Library

#include <math.h>

• sin, cos, tan, asin, acos, atan, exp, log, log10, pow, sqrt, ceil, floor, fabs, ... (Complete list in Appendix A)

• Attention: trigonometric functions mostly use radians (not degrees)

• When compiling with gcc, don’t forget to add “–lm” as an argument
  • gcc –Wall –ansi –pedantic-errors –lm –o <outputFileName> yourApp.c
#include <math.h>  
#include <stdio.h> 

#define PI 3.141592 

double degreeToRadian ( double degree ) { return degree * PI / 180.0 ; } 

double radianToDegree ( double radian ) { return radian / PI * 180.0 ; } 

void printLine ( double degree ) 
{ 
    double sinValue = sin( degreeToRadian( degree ) ) ; 

    if ( sinValue <= -0.75 ) printf( "X........" ) ; 
    else if ( sinValue <= -0.50 ) printf( ".X........" ) ; 
    else if ( sinValue <= -0.25 ) printf( "..X......" ) ; 
    else if ( sinValue <= 0.00 ) printf( "....X...." ) ; 
    else if ( sinValue <= 0.25 ) printf( ".....X.." ) ; 
    else if ( sinValue <= 0.50 ) printf( "......X." ) ; 
    else if ( sinValue <= 0.75 ) printf( ".......X" ) ; 
    else /*sinValue <= 1.00*/ printf( ".......X" ) ; 
} 

void main ( void ) 
{ 
    double degree ; 

    for ( degree = 0 ; degree < 720 ; degree += 45 ) 
    { 
        printf( "\n" ) ; 
    } 
}
Chapter 4 – Repetitive Structure – Outline

• Overview
• while Loop
• do-while Loop
• for Loop
• Nested Loops
• Loop Interruption (break and continue statements)
• Null Statement
• Comma Operator
Repetitive = Iterative = Loop

• Loop Continuation Condition
  • Controls the number of repetitions
  • Can be tested before or after the body
    • If before, loop body may never execute
      • while, for
    • If after, loop body executes at least once
      • do-while
  • Usually a loop control variable is used

• Loop Body
  • Single statement or a set of statements (compound statement; block)
While Loop

• Syntax :  

```
while ( expression )
```

statement

• Semantics :
  • Expression is evaluated
  • If true (non-zero), statement is executed
  • If false (zero), the loop terminates

• Expression is NOT optional (it has to be provided)

• Block (compound statement) can be used if loop body (statement) needs to perform multiple actions
While Loop – Example

• Example (textbook, page 108)
• getchar returns int
  • char is not necessarily signed
  • EOF usually defined as -1
    • Terminal: CTRL-D (unix/linux), CTRL-Z (dos)
    • File: When file runs out of data
• putchar expects int
• Try variations:
  • Compact variable definitions
  • while (( ch = getchar() ) != EOF )
  • Use scanf/printf for echoing

```c
#include <stdio.h>
void main ( void )
{
  int ch ;
  int count = 0 ;
  ch = getchar() ;

  while ( ch != EOF )
  {
    putchar( ch ) ;
    count++ ;
    ch = getchar() ;
  }

  printf( "Total characters echoed = %d\n" , count ) ;
}
```
Infinite Loop

• A loop that never terminates
  • Might be a logical mistake
  • Might be due to use of = instead of ==
  • Might be due to real numbers
  • Might be intentional
    • Example: A computer game’s main loop
      • while ( 1 )
        {
          getUserInput() ;
          applyToGame();
          sleep( 100 /* milliseconds */ ) ;
        }
      • Will be broken by a break statement
      • Will be broken by a return statement

```c
int n , sum = 0 ;
scanf( "%d" , &n ) ;  /* What if n < 0 */
while ( n != 0 )
{
  sum += n--; 
}
printf( "Sum = %d
", sum ) ;
```
Do-While Loop

• Syntax:   do  
              statement  
              while ( expression ) ;

• Semantics:
  • Execute statement
  • Evaluate expression
    • If true (non-zero), execute statement
    • If false (zero), the loop terminates

• Expression is NOT optional (it has to be provided)
• Block (compound statement) can be used if loop body (statement) needs to perform multiple actions

Practical (Common) Use

```
loop initialization

do
    statement-1
    while ( expression ) ;
statement-2
```
Do-While Loop – Example

- Example (textbook, page 113)
- Also works properly for input: 0
  - Try solving with `while` loop
  - Will need special handling for input 0

```c
#include <stdio.h>

void main ( void )
{
    int number , digits = 0 , sum = 0 ;

    printf( "Enter a non-negative integer : " ) ;  
    scanf ( "%d" , &number ) ;  

    do 
    {  
        sum    += number % 10 ;  
        number /= 10          ;  
        digits++              ;
    } 
    while ( number > 0 ) ;

    printf( "Number of digits = %d
        \n" , digits ) ;  
    printf( "Sum    of digits = %d\n" , sum    ) ;
}
```
For Loop

• Syntax:  for ( expr1 ; expr2 ; expr3 )  
  statement

• Semantics:
  • Evaluate expr1
  • Evaluate expr2
    • If true (non-zero), execute statement
    • Then evaluate expr3
    • If false (zero), the loop terminates

• expr1, expr2 and expr3 are optional

• Block (compound statement) can be used if loop body (statement) 
  needs to perform multiple actions

/* Roughly equivalent to this: */
expr1 ;
while ( expr2 )
{
  statement
  expr3 ;
}
For Loop – Example

• Usually (but not necessarily) :
  • expr1 is for initialization(s)
  • expr2 is loop continuation condition
  • expr3 is re-initialization expression

• If expr2 is omitted
  • It is assumed to be true
  • Infinite loop
    • Might be broken by break, return

• Infinite loop examples
  • for ( ; ; ) ;
  • for ( ; ; ) {}
  • while ( 1 ) ;
  • while ( 1 ) {}

```c
#include <stdio.h>

int factorial ( int positiveNumber )
{
    int i , result = 1 ;
    for ( i = 1 ; i <= positiveNumber ; i++ )
    {
        result *= i ;
    }
    return result ;
}

void main ( void )
{
    printf( "5! = %d\n" , factorial( 5 ) ) ;
}
```

Examples

• The following are roughly equivalent
  (variable definitions are shown only in the first example)

```c
int i, sum = 0;
for (i = 1; i <= n; i++)
    sum += i;
printf("%d\n", sum);
```

```c
i = 1;
while (i <= n)
    {
        sum += i;
        i++;
    }
printf("%d\n", sum);
```

```c
i = 1;
for ( ; i <= n; )
    {
        sum += i;
        i++;
    }
printf("%d\n", sum);
```
Examples

• The following are roughly equivalent
  (the second example uses a dummy variable, x, of the same type as the expression)

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

```plaintext
for (x=1; x; x= (expression))
  statement
```
Nested Loops

• The **statement** part of a loop might be / might contain any other loop

• There is no limit on the depth of nesting

```c
#include <stdio.h>
void main ( void )
{
    int row , col ;
    for ( row = 1 ; row < 10 ; row++ )
    {
        for ( col = 1 ; col < 5 ; col++ )
        {
            printf ( " X" ) ;
        }
        printf ( "\n" ) ;
    }
}
```

Side Note: I wish I was sponsored by: SpaceX for the code example on this slide 😊
Loop Interruption

• Sometimes we may need to
  • Exit the loop using another mechanism than the loop termination condition
  • Skip just the current iteration, but continue with the loop execution

• Almost all languages provide `break` and `continue` statements
  • `break` (inside a loop body) exits the loop
  • `continue` (inside a loop body) skips the remaining statements in the loop body
Break Statement

- **break** statement in a loop body transfers the program control to the exit point of the loop
- If loops are nested, **break** statement applies to the **innermost** loop (whose body it appears in)
- **break** statement can only be used inside **loops** or **switch** statements
- In case of **nested** statements:
  - Example: loop inside another loop, switch inside loop, loop inside switch
  - **break** applies to the **innermost** statement (which directly contains break)
Break Statement – Example

• Finding sum of prime numbers between 10 and 100

```java
for (i = 10; i <= 100; i++)
{
    /* check if i is prime */
    for (j = 2; j <= sqrt(i); j++)
        if (i % j == 0) /* i is not prime */
            break;
    if (j > sqrt (i)) /* i is prime */
        sum += i;
}
```

• It is a good idea to pay attention to values of variables after the loop finishes
Continue Statement

• `continue` statement causes skipping of the remaining statements in a loop body
  • `while`, `do-while` loops: loop continuation condition is evaluated next
  • `for` loops: re-initialization expression is evaluated next

• If loops are nested, `continue` statement applies to the innermost loop (whose body it appears in)

• `continue` statement can only be used inside loops

• When `continue` statement is used, a `while` loop may not be equivalent to the `for` loop
Sum of integers from 1 to n
(Integers divisible by 5 should not be included in the sum)

For loop is not equivalent to while loop when we use continue statement in the body

It is possible to avoid most uses of continue statement by using an appropriate if statement

```c
for (i = 1; i <= n; i++)
    if (i % 5 != 0)
        sum += i;
```

```c
i = 1;
while (i <= n)
    {
        if (i % 5 == 0) continue;
        sum += i;
        i++;
    }
```
Break and Continue – Example

• Counting number of tab characters in user input

```c
    count = 0;
    while ( (c = getcharO) != EOF)
    {
        if (c != '\t' && c != '\n') continue;
        if (c == '\n') break;
        if (c == '\t') count++;
    }
```
Null Statement

• A solitary semicolon ;
• Can be placed wherever a regular statement can appear
• Execution of null statement has no effect
  • May seem useless
  • Sometimes necessary because C language syntax requires a statement
• Example: Counting number of characters in input

```
for ( count=0 ; getchar() != EOF ; count++ )
;
```

• Good programming practice: place null statement on a line by itself
Comma Operator

- **Syntax:** $expr_1, expr_2$ (can be extended: $e_1, e_2, e_3, ..., e_N$)
- **Evaluation:**
  - Expressions are evaluated from left to right
  - The type and value of whole expression is the type and value of right operand (left operand is evaluated only for the side effect, its value is discarded)
- **Purpose:** combining two related expressions into one expression, making programs more compact
- **Comma operator has the lowest precedence of all operators**
  - Can safely be used to turn any list of expressions into a single expression
Comma Operator – Examples

/* Assume */
int i;
float x;

/* Expressions */
i = 1
x = i + 1

/* Combined into single expression */
i = 1, x = i + 1
/* type: float, value: 2 */

/* Swapping values of x and y */
temp = x;
x = y;
y = temp;

/* Swapping values of x and y */
temp = x, x = y, y = temp;
/* Evaluated left to right */

while ((c = getchar()) != EOF) putchar(c);
can be rewritten as
while (c = getchar(), c != EOF) putchar(c);
separating the reading of the character from testing the end-of-file.

while (printf("Enter two integers : "),
scanf("%d %d", &n1, &n2) != EOF)
{
    printf("Sum = %d\n", n1 + n2);
}

while (scanf("Enter two integers : ",
%d %d", &n1, &n2) != EOF)
{ /*...*/
    printf("Sum = %d\n", n1 + n2);
}  /*...*/
Chapter 5 – Functions

• Overview
• Function Definition
• Function Call
• Function Prototypes
• Block Structure
• External Variables
• Storage Classes
• Separate Compilation and Data Abstraction
• Recursion
Overview

- C applications usually consist of many functions in multiple files
- Functions (and logically grouped functions in files) can individually be
  - Developed, tested, debugged, compiled, maintained
- Sample function: cubesum
  - Type: int
  - Function name: cubesum
  - Parameters ("formal parameters")
  - Function body
    - Local variables
    - Function statements

```c
int cubesum(int number)
{
    int sum, residue, digit;
    residue = number;
    sum = 0;
    do
    {
        digit = residue % 10; /* rightmost digit */
        sum += digit * digit * digit;
        residue /= 10; /* after removing this digit */
    }
    while (residue != 0);
    return sum;
}
```
Narcissistic Cubes (uses “cubesum”)

```c
#include <stdio.h>
define MAX 5

int main(void)
{
    int i, count;
    int cubesum(int number);

    for(i=1, count=0; count < MAX; i++)
        if (i == cubesum(i))
            {
            printf("%d\n", i);
            count++;
            }
    return 0;
}
```
Function Definition – Syntax (Review)

ReturnType FunctionName ( Type1 Parameter1Name, Type2 Parameter2Name )
{
    Variable Definitions, External Variable/Function Declarations
    Type var1Name, var2Name, var3Name, ... ;
    extern Type globalVar1Name, globalVar2Name, ... ;
    extern ReturnType otherFunctionName ( Type1, Type2, Type3 ) ;

    Statements
    Expression1 ;
    Expression2 ;
    ...
}

Ceng 140 - Section 2  Selim Temizer - Lecture Notes
Function Definition - Notes

• If return type is omitted, it is taken to be `int`
• Function names obey the `identifier naming rules` that we saw before
• Local variables only exist, and accessible, within their functions
• Local variable names `shadow` identically named variables outside
• Function parameters are also (initialized) local variables of the function
• Function statements are any valid C statements
• Function execution terminates when
  • Either, execution reaches the `closing brace` of the function body
  • Or, a `return` statement is encountered
Return Statement

• Syntax: return expression; or return;
• In first form; value of the expression is returned to the calling function
• If type of expression does not match type of function, it is converted

```c
int trunc(void)
{
    return 1.5;
}
```

• If function return type is void:
  • Cannot use first form (error)
• If function return type is not void:
  • If second form is used, value returned is unpredictable
• More than one return statement can exist in a function body
Function Call (Expression)

- Syntax: `functionName (argumentList)`
- Arguments are also called “actual arguments”
- Use empty parentheses if function does not take any parameters
- Arguments are separated with commas
  - Note: This is not the comma operator! It is a syntactic entity
  - If you need to use comma operator, use parentheses
- Definitions:
  - Calling function (caller): Function which calls another function
  - Called function (callee): Function which is called
- The type/value of the function-call expression is the type/return value of the called function
- Calling function may choose to ignore the value returned by called function
Memory Segments (Formal Coverage)

• A compiled and linked executable file stores:
  • TEXT (machine code representation of the functions)
  • DATA (initialized global/static data)
  • BSS (uninitialized global/static data)
  • Also dynamic library links, debugging info, symbol tables, etc.

• When an executable file is started (double click or type its name)
  • OS loader loads TEXT, DATA and BSS in their own memory segments
  • Additional regions are allocated: CL Arguments & Env. Variables, HEAP, STACK

• HEAP segment houses dynamically allocated memory regions
• STACK segment is used to manage control flow of the application
Memory Layout of C Programs

https://www.geeksforgeeks.org/memory-layout-of-c-program/
Call Stack and Stack Frames (Activation Records)

• Each function call adds a stack frame to the STACK segment
• A stack frame (activation record) usually contains
  • Argument values passed to the function
  • Function’s own local variables
  • Return address
  • Storage for intermediate/temporary results (of long expressions)
  • Possibly some other housekeeping data, like
    • A pointer to the previous frame (and/or the size of the current frame)
    • Some saved register values that need to be restored after the function call
    • Some storage for return value (return value may also be returned through registers)
    • etc.
• Exact stack frame format is compiler/system dependent
Stack Frame – Sample Images from Internet

- **Argument 1**
- **Argument 2**
- **Return Address**
- **Saved %ebp**
- **Local Var 1**
- **Local Var 2**
- **Local Var 3**

**Caller's frame**

- **%ebp**
- **Current Frame**

**Stack growth**

**Current frame pointer - RBP**

**Current stack pointer - RSP** (stack top)

- **Function Arguments**
  - Return address to the caller
  - Save old frame pointer (RBP)
  - Local variable storage
  - Buffer storage for temp data
  - Saved registers by the callee - RBX, RSI, RDI
Stack Frame – Sample Images from Internet

Corresponding C code:

```c
int function(int p1, int p2, int p3)
{
    int A, B, C;
    ...
}
```
## Memory Layout and Call Stack Example

<table>
<thead>
<tr>
<th>Entity</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Differences</th>
<th>Print Order</th>
<th>Memory Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where argv[0] points to</td>
<td>0x7fffc3bb6f</td>
<td>140737232747375</td>
<td>503</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Address of argv[0]</td>
<td>0x7fffc3bb78</td>
<td>140737232746872</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Address of heap2</td>
<td>0x7fffc3b880</td>
<td>140737232746624</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Address of heap1</td>
<td>0x7fffc3b878</td>
<td>140737232746616</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Address of localMain</td>
<td>0x7fffc3b874</td>
<td>140737232746612</td>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Address of argc</td>
<td>0x7fffc3b86c</td>
<td>140737232746604</td>
<td>12</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Address of argv</td>
<td>0x7fffc3b860</td>
<td>14073723274592</td>
<td>28</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Address of local1</td>
<td>0x7fffc3b844</td>
<td>140737232746564</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Address of parameter1</td>
<td>0x7fffc3b83c</td>
<td>140737232745566</td>
<td>40</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Address of local2</td>
<td>0x7fffc3b814</td>
<td>140737232746516</td>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Address of parameter2</td>
<td>0x7fffc3b80c</td>
<td>140737232740508</td>
<td>40</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Address of local3</td>
<td>0x7fffc3b7e4</td>
<td>140737232746468</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Address of parameter3</td>
<td>0x7fffc3b7dc</td>
<td>140737208965036</td>
<td>140737208965036</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entity</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Differences</th>
<th>Print Order</th>
<th>Memory Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where heap2 points to</td>
<td>0x16ae030</td>
<td>23781424</td>
<td>32</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Where heap1 points to</td>
<td>0x16ae010</td>
<td>23781392</td>
<td>32</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>Address of global1</td>
<td>0x602068</td>
<td>6299752</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>Address of staticLocalMain</td>
<td>0x602064</td>
<td>6299748</td>
<td>4</td>
<td>6</td>
<td>17</td>
</tr>
<tr>
<td>Address of staticLocal3</td>
<td>0x602060</td>
<td>6299744</td>
<td>4</td>
<td>27</td>
<td>18</td>
</tr>
<tr>
<td>Address of staticLocal2</td>
<td>0x60205c</td>
<td>6299740</td>
<td>4</td>
<td>24</td>
<td>19</td>
</tr>
<tr>
<td>Address of staticLocal1</td>
<td>0x602058</td>
<td>6299736</td>
<td>4</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Address of staticGlobal2</td>
<td>0x602054</td>
<td>6299732</td>
<td>4</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>Address of staticGlobal1</td>
<td>0x602050</td>
<td>6299728</td>
<td>4</td>
<td>13</td>
<td>22</td>
</tr>
<tr>
<td>Address of global2</td>
<td>0x60204c</td>
<td>6299724</td>
<td>2103519</td>
<td>12</td>
<td>23</td>
</tr>
<tr>
<td>Address of main</td>
<td>0x400576d</td>
<td>4196205</td>
<td>129</td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>Address of function3</td>
<td>0x4005ec</td>
<td>4196076</td>
<td>139</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>Address of function2</td>
<td>0x400561</td>
<td>4195937</td>
<td>139</td>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>Address of function1</td>
<td>0x400566d</td>
<td>4195798</td>
<td>4195798</td>
<td>15</td>
<td>27</td>
</tr>
</tbody>
</table>

---

**Linux Prompt> size a.out**

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>3549</td>
<td>568</td>
<td>40</td>
<td>4157</td>
<td>103d</td>
<td>a.out</td>
</tr>
</tbody>
</table>

---

Call by Value (Pass by Value)

- C only supports call-by-value mechanism for passing data to functions
- Some other languages also support call-by-reference (aliasing)
- Classical example:
  - Swapping values (not possible)
  - If programmer needs to swap:
    - Pass addresses (pointers), indirection
    - Or, use external variables

```c
#include <stdio.h>

int main(void)
{
    int i = 1, j = 2;
    void exchange (int, int);

    printf("main : i = %d j = %d\n", i, j);
    exchange(i, j);
    printf("main : i = %d j = %d\n", i, j);
    return 0;
}

void exchange(int i, int j)
{
    int t;
    t = i, i = j, j = t;
    printf("exchange: i = %d j = %d\n", i, j);
}
```
Function Prototypes (Declarations)

• Syntax: \[ \text{returnType functionName ( listOfParameterTypes ) ;} \]
• Parameter names are not necessary
  • If used, they can also be different from the names used in definition
• The arguments are converted to the declared types of the parameters
  • As if by assignment
  • No need for explicit type casting
• Function \text{definition} also serves as a \text{declaration} for the rest of the file
• An undeclared function will be assumed to have the return type of \text{int}
  • Suggestion: Always declare functions before using them
  • If compiler is told the prototype, it can help programmer catch type mismatches
Block Structure

• Syntax is similar to the body of a function
  • Variable definitions/declarations and statements within curly braces
  • \{ variable definitions/declarations statements \}

• Blocks can be nested
• Scope of a variable defined in a block is until the end of the block
• Block variable names _shadow_ identically named variables in outer blocks
• Technically, blocks do not act as stack frames
  • They allow to define new variables (coming after some statements)
  • They also provide _scope_ for the variables defined within
  • From a programmer’s view, you may assume that they act as a stack frame
External Variables

• External variables
  • Defined outside of any function
  • Available to all functions below in the same file
  • If defined before any function definition: sometimes called “global variable”
    • Available to all function in the file

• Usages
  • Passing data between functions
  • A function that needs to return more than 1 values can use global variables

• Disadvantages
  • Hides relationship between different parts of a program (hard to understand)
  • Changes in one part of the program may affect some other part in unexpected ways
  • Using external variables in a function destroys generality of the function
Scope and Lifetime of Entities

- **Scope**: Part of the program where a name can be used/accessed
  - **External entities** (non-static global variables and non-static functions)
    - From the point of their **definition/declaration** until the end of the file
    - From the point of their **declaration** in a block, until the end of the block
    - Note: static external entities are NOT visible outside of the file they are defined in
  - **Function parameters**
    - Inside the function
  - **Local variables and static local variables** (in **functions and/or blocks**)
    - Inside the function/block

- **Lifetime**: The duration that the entity is kept in memory
  - **External entities and all static entities**: Equal to the lifetime of the **program**
  - **Function parameters and local variables**: Equal to the lifetime of the **stack frame**
  - **Dynamically allocated data**: Until they are **freed explicitly** (or **program termination**)

Ceng 140 - Section 2
Selim Temizer - Lecture Notes
Storage Classes

• A variable belongs to one of two storage classes: **Automatic**, **Static**

• Do not confuse **static storage class** with the keyword **static** in C language

• **Storage class** is about **lifetime** of the variable

• **Automatic** variables:
  • Allocated upon entry to a code segment
  • Deallocated upon exit from that segment
  • They can only be defined within a block
  • Syntax: `auto type variableName;`
  • If storage class specifier (**auto**) is not used: variables in blocks taken to be auto
  • Can also be initialized when defined: `auto type variableName = expression;`
    • Expression is evaluated **each time** the block is entered
    • Function parameters, function calls, etc. can also be used in the expression (can be dynamic)
  • If not explicitly initialized, initial value of an automatic variable is **undefined**
Storage Classes

• **Static** variables:
  • Allocated at the beginning of the program execution
  • Storage remains allocated until the program terminates
  • *External variables* are always **static** (don’t confuse with static keyword!)
  • Syntax for static variables within blocks: `static type variableName ;`
  • Can also be **initialized** when defined: `static type variableName = expression ;`
    • Expression is evaluated just for the first time (and once) when the block is entered
    • Expression can only be a constant expression (should be compile-time-evaluatable)
  • If not explicitly initialized, initial value of a static variable is zero
  • Values of static variables are retained across calls to the function they are defined in
    • Lifetime = Lifetime of program
    • Scope = To the end of the block from the point they are defined
Storage Classes - Example

```c
#include <stdio.h>
int main(void)
{
    int i;
    void incr(void);

    for (i = 0; i < 3; i++) incr();
    return 0;
}

void incr(void)
{
    int auto_i = 0;
    static int static_i = 0;

    printf("auto = %d static = %d\n",
           auto_i++, static_i++);
}
```

- Output:

  auto = 0  static = 0
  auto = 0  static = 1
  auto = 0  static = 2
Separate Compilation and Data Abstraction

• Separate Compilation
  • Large programs are usually divided into logically related functions, called modules
  • Modules usually reside in separate files
  • Each file can be compiled, tested, maintained, etc. individually

• Abstract Data Objects
  • Equivalent to the concept of classes in object-oriented languages
  • Implementation details in a module/file can be hidden from the outside world
  • Only selected functions (interface) can be made available to the other files
  • Local changes in the module will not affect other files (other parts of application)
extern Declaration

• Remember difference between definition and declaration:
  • Definition allocates storage for the entity, also serves as a declaration
  • Declaration just lets the compiler know about the type and name of entity for the rest of the scope

• Every entity should be defined only once, somewhere
• Only external (top level) entities can be declared
  • Local variables cannot be declared, they can only be defined
• An external entity can be declared many times in same/different scopes
• For functions, use of extern keyword in declaration is optional
• External variables can be initialized only at definition (not declaration)
• Related declarations are usually grouped in header files for easy use
  • Other files just #include the header files, and then get linked with the implementations
Information Hiding

• Function and external variable definitions, prefixed with static keyword can only be accessed in the file they are defined
  • Their scope is limited from the point of their definition to the end of the file
  • Other files cannot access them (they are invisible to other files)

• Remember that:
  • static keyword applied to an external variable limits its scope to its file
  • static keyword applied to a local variable makes its lifetime equal to the lifetime of the application
Recursion

• A function may also call itself (same mechanism as calling any function)
  • This can be direct: call occurs inside the function itself
  • Or indirect: function calls other function, … , last function calls first function

Recursion in computer science is a technique where solution to a problem depends on
solution(s) to smaller instance(s) of the same problem

This approach can be applied to many types of problems, and it is one of the central ideas of computer science (used in Dynamic Programming, some AI algorithms)

Recursion - Example

```c
int factorial(int n)
{
    if (n == 0)
        return 1;    /* termination condition */
    else
        return n * factorial(n-1); /* recurse */
}

int factorial(int n)
{
    return n == 0 ? 1 : n * factorial(n-1);
}

int factorial(int n)
{
    if (n > 0)
        return n * factorial(n-1);
    else if (n == 0)
        return 1;
    else
    {
        printf("no factorial for negative values\n");
        return -1;
    }
}
```
Recursion – Notes from CENG 433

( Quick coverage of Recursion presentation from CENG 443 )
Recursion vs. Iteration

• In theory, any recursive function can be transformed into an iterative function
  • Recursive solutions are generally more understandable
  • Iterative solutions are generally faster
    • They eliminate function calls
    • They may eliminate duplicate computations

```c
int fib(int n)
{
    return n == 0 || n == 1 ?
        1 : fib(n-1) + fib(n-2);
}
```
Chapter 6 - Arrays

- Basics of Arrays
  - Array Definition/Declaration
  - Accessing Array Elements
  - Array Initialization

- Arrays as Function Arguments
  - Passing Array Elements as Arguments
  - Passing Arrays as Arguments
Basics of Arrays

• C language provides a capability similar to subscripting in mathematics
  • We can process collections of related data items easily
  • There is no need to invent new variable names for each item
  • We can refer to each item by the same group name and a subscript (index)

• Example
  • In TicTacToe (HW1), we created separate variables for board cells: c0, c1, … , c8
  • In DnDW (HW1), we created separate variables for each room: r0, r1, … , r9
  • What would happen if we had 10000 cells/rooms in the game?
    • With arrays we can define/create all of the cells easily: char cells[10000] ;
  • Arrays provide us much practical ways to work with a set of related data items
    • With array notation we can access any cell/room by using an index: rooms[395] = EMPTY ;
    • Also, we can create loops to iterate over all cells/rooms instead of long lists of statements:
      for ( i = 0 ; i < 10000 ; i++ ) { rooms[i] = EMPTY ; }
Definitions and Array Basics

• **Array**: An ordered finite collection of data items (same type each)
• **Elements**: Individual data items in an array

• **Array indices (subscripts)** start at 0 (zero)!
  • Example: room[0] is the first element, room[9] is the 10\textsuperscript{th} element

• **Properties of an array:**
  • **Type**: data type of its elements
  • **Location** (in memory): location of its first element
  • **Length**: number of elements in the array
  • **Size**: length of the array $\times$ size of an element (in Java, “size” refers to length)

• An array’s name is the address of the (1\textsuperscript{st} byte of the) first element of the array

• Array elements are placed **contiguously** in the memory
  • There is no gap and/or padding between any two consecutive array elements
Examples

• One-dimensional (single-subscripted, linear) array (vector): elements
• Two-dimensional (double-subscripted) array (matrix): rows, columns
• Three-dimensional array: rows, columns, levels
• ... (C language allows any number of dimensions)
• Note: In computer’s memory, all data items are stored like a vector
  • See example code: VectorAsMatrix.c
Array Definition

• Syntax:  `type arrayName [ expr₁ ] [ expr₂ ] ... [ exprₙ ] ;`

• Expressions should be compile-time computable and integral-valued
  • We can use constants and/or symbolic-constants in the expressions
  • Example: `int midtermGrades [ CENG_STUDENTS + ERASMUS_STUDENTS ] ;`

• Arrays are just like regular variables
  • Arrays can be local or global
  • Arrays can be defined as auto or static
  • Global arrays can be declared using extern keyword
    • When declaring, you may omit the first dimension, but all other dimensions are necessary in order to calculate the correct position of an array entity
    • Example: `extern int testScores [ ] [ NUMBER_OF_TESTS ] ;`
Accessing Array Elements

• arrayName[ index ] is equivalent to *( arrayName + index )
  • Basically, the box (of one of the elements) is calculated and returned to us
  • After that point, we can use this box just like any regular variable:
    Read the value in it, write some value in it, use it in an expression, etc.
    Example: matrix[2][3] = --matrix[0][0] + matrix[1][5] ;
  • See example code: ArraysAndAddresses.c
  • Multiple dimensions are handled similarly
    • Example: array[ index_1 ][ index_2 ] is equivalent to *( *( array + index_1 ) + index_2 )

• index can be any integral-valued expression

• C does NOT check the index to lie within the array’s memory region
  • We need to be very careful to NOT access outside of array’s boundaries
Copying Arrays into Other Arrays

• We cannot assign arrays to other arrays directly

• We need to copy element-by-element using a loop, or we need to use a library function

• Later in the course, we will also see another way to copy arrays (put array in a struct, and assign structs)

```c
int from[5] = { 1, 2, 3, 4, 5 } ;
int to[5] ;

to = from ; /* Illegal */
to[] = from[] ; /* Illegal */

for ( i = 0 ; i < 5 ; i++ ) /* OK */
{
    to[i] = from[i] ;
}

#include <string.h>

memcpy( to, from, 5 * sizeof(int) ) ; /* OK */
```
Array Initialization

• Syntax: List of initializers, enclosed in braces, separated by commas


• Multi-dimensional arrays
  • Inner braces are optional

```c
int test_score[5][4] = {
    { 95, 98, 97, 96 },
    { 79, 89, 79, 85 },
    { 99, 98, 99, 99 },
    { 90, 89, 83, 86 },
    { 75, 72, 79, 69 }
};
```
Array Initialization – Special Rules

• If number of initializers < number of elements → remaining elements are set to 0 (zero)
  • Useful trick for local arrays: int array[1000] = {0} ;

```cpp
int score[5] = { 41, 97, 91 };
int test_score[5][4] =
  {
    { 0, 98 },
    { 79, 89, 79 },
    { 0, 0, 0, 99 },
    { 90 },
  };
int score[5] = { 41, 97, 91, 0, 0 };
int test_score[5][4] =
  {
    { 0, 98, 0, 0 },
    { 79, 89, 79, 0 },
    { 0, 0, 0, 99 },
    { 90, 0, 0, 0 },
    { 0, 0, 0, 0 }
  };
```

• If number of initializers > number of elements → ERROR
Array Initialization – Special Rules

• If there are initializers, you can omit the array length
  • Compiler can count the number of initializers to determine length

```c
float sqrt[] = { 0, 1.0, 1.414, 1.732, 2.0 };
```

```c
float sqrt[5] = { 0, 1.0, 1.414, 1.732, 2.0 };
```

• Character arrays may also be initialized with string constants
  • Terminating ‘\0’ is also copied to the array

```c
char os [4] = "AIX";
```

```c
char os [4] = { 'A', 'I', 'X', '\0' };
```

```c
char computer[] = "sierra";
```

```c
char computer[7] = { 's', 'i', 'e', 'r', 'r', 'a', '\0' };
```
Array Initialization – Special Rules

It is also possible to initialize a character array with an explicit length specification by a string constant with exactly as many characters. The array does not receive the terminating ' \ 0' in that case. For example,

```c
char os [3] = "AIX";
```

is equivalent to

```c
char os[3] = { 'A', 'I', 'X' };
```

However, the string constant must not have more characters than the length of the character array being initialized. Thus, it is an error to initialize os as

```c
char os [3] = "ultrix";
```
Passing Array Elements as Arguments

- **Array elements** can be passed to functions just like regular variables
  - **Pass-by-value** mechanism (a copy of the value is passed to the function)
  - If type is different than parameter type, **conversion rules** apply as usual

```c
void printNumber ( long int x )
{
    printf( "%d\n" , x ) ;
}

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

    for ( i = 0 ; i < 5 ; i++ )
    {
        printNumber( numbers[i] ) ;
    }
}
```
Passing Arrays as Arguments

• Passing array’s name to a function = passing array’s starting address
  • Pass-by-value mechanism (a copy of the address is passed to the function)
  • In function prototype, indicating the first dimension is optional

```c
void add5 ( int array[3] )
{
    int i ;

    for ( i = 0 ; i < 3 ; i++ )
    {
        array[i] += 5 ;
    }
}

void main ( void )
{
    int numbers[3] = { 1 , 2 , 3 } ;
    add5( numbers ) ;
}
```

```c
void add5 ( int array[], int length )
{
    int i ;

    for ( i = 0 ; i < length ; i++ )
    {
        array[i] += 5 ;
    }
}

void main ( void )
{
    int num1[4] = { 1 , 2 , 3 , 4 } ;
    int num2[2] = { 8 , 7 } ;
    add5( num1 , 4 ) ; add5( num2 , 2 ) ;
}
```
Passing Arrays as Arguments

When declaring a multi-dimensional array as a parameter, it is necessary to specify the lengths of all but the first dimension. Thus, the prototype of a function to compute the cube root of every element of a two-dimensional array can be written as

```c
void matrix_cuberooroot(double xx[10][5]);
```

or as

```c
void matrix_cuberooroot(double xx[][5], int rows);
```

but not as

```c
void matrix_cuberooroot(double xx[][][],
                         int rows, int cols);
```
Array Practice – Self Study Suggestions

• Great and classical examples in *textbook* (pages 189 – 202), you really SHOULD study them

• It would be educative to rewrite *TicTacToe* and *DnDW* using arrays
  • Code will be much shorter and much easier to understand

• Coding *vector/matrix operations* (from linear algebra) would be very good practice, engineers always need linear algebra libraries

• Building *grid/maze/board games* would be fun, and would provide lots of array (especially 2D array) practice

• Coding various *searching* and *sorting* algorithms would provide good practice and will also help you in future courses (like CENG 213, 315)
Chapter 7 - Pointers

• Basics of Pointers
• Functions and Pointers
• Arrays and Pointers
• Strings and Pointers
• Multi-Dimensional Arrays and Pointers
• Pointer Arrays
• Pointers to Functions
• Dynamic Memory Management
Why Pointers?

• Pointers are used in almost every C program
• Sometimes they provide the only way to express a computation
• They lead to more efficient and compact code
• They enable us to emulate call-by-reference mechanism
• They help us to deal concisely and effectively with arrays
• They enable us to represent complex data structures (like linked-lists)
• They enable us to work with dynamically allocated memory
Basics

• The **address** of a variable/function **may be stored** in another variable

• **Pointer** (**address**): The address of an object (variable, function, etc.)

• **Pointer Variable** (**address variable**): A variable that holds an address
Address and Dereferencing Operators

• & : “address” or “address of” operator
  • Unary
  • Can only be applied to an L-value
    • Correct examples: &i, &main (i is a variable, main is a function)
    • Incorrect examples: &10, ’C’, &(x + 3)
  • If the operand is of type “T”, the result is of type “pointer to T”
    • Example: float f = 1.0f; float * fp = &f; (fp is a pointer to float)

• * : “dereferencing” or “indirection” or “value at the address” operator
  • Unary
  • Can only be applied to a pointer
    • Examples: *(&i), *(&main), *(int *) 100 (i is a variable, main is a function)
  • If the operand is of type “pointer to T”, the result is of type “T”
    • Example: float f = 1.0f; float * fp = &f; float value = *fp (value is a float)
    • *fp can occur anywhere f can (lvalue, rvalue, etc. ; they are semantically equivalent)
Defining a Pointer Variable

- **Pointer Variable**
  - Similar to regular variables, pointer variables have a **value** and a **type**
  - **Value** is the address stored in this variable
  - **Type** information helps the compiler use the address value properly
    - Needs to know how this address is to be interpreted (as a char, int, unsigned, float, ... ?)

- **Syntax:** `type *identifier = optional initializer ;`
  - `identifier` is a "pointer to type"
  - Semantics: the address stored in `identifier` is to be interpreted as the address of a `type` variable/box/object
  - Another interpretation: `*identifier` is an object of the stated `type`

- For each type of object (variable, function, struct, union, etc.) in C, a corresponding type of pointer can be declared

- If you need a generic pointer, use `void` as type: `void * vp ;`
  - Void pointers should not/cannot be dereferenced without a type cast
Examples

```c
int i = 1, j, *ip;
ip = &i;
j = *ip;
*ip = 0;
```

Note that the two statements

```c
ip = &i;
j = *ip;
```

are equivalent to the single assignment

```c
j = *(&i);
```

or to the assignment

```c
j = i;
```

The expression *ip, where ip is a pointer to integer i, can occur in any expression in any context where i can. Thus,

```c
j = *ip + 10;
```

is equivalent to

```c
j = i + 10;
```

and assigns 10 more than i to j;

```c
k = ++(*ip);
```

is equivalent to

```c
k = ++i;
```

and increments the value of i and assigns the incremented value to k;

```c
x = sqrt((double) *ip);
```

is equivalent to

```c
x = sqrt((double) i);
```

and casts i to double before it is passed to sqrt, and assigns the result to x; and

```c
printf("%d\n", *ip);
```

is equivalent to

```c
printf("%d\n", i);
```

and prints the current value of i.
NULL Pointer

• **Pointers are NOT equivalent to integers**
  • They are not interchangeable
    • Exception: The constant 0 (zero) can be assigned to a pointer of any type
  • Size of a pointer may be equal to (depending on the architecture)
    • Size of `unsigned long int` (usually 4 bytes, for 32-bit architectures)
    • Size of `unsigned long long int` (usually 8 bytes, for 64-bit architectures)

• A pointer value of 0 (zero) is known as the **NULL pointer**
• `stdio.h` contains the definition of NULL

```c
#include <stdio.h>
float *fp = NULL;
```
```c
int main(void)
{
    int i, j = 1;
    int *jpl, *jp2 = &j; /* jp2 points to j */

    jpl = jp2; /* jpl also points to j */
    i = *jpl; /* i gets the value of j */
    *jp2 = *jpl + i; /* i is added to j */

    printf("i = %d, j = %d, *jpl = %d, *jp2 = %d\n", i, j, *jpl, *jp2);
    return 0;
}
```
• Only addition (+) and subtraction (-) allowed (++, --, +=, -= also OK)

• Assume \( p \), \( p1 \) and \( p2 \) are pointers, \( n \) is an integral-valued expression
  • \( p + n \) : result is a pointer to the object that lies \( n \) objects after \(*p\)
    • Calculation: \( p + n \times \text{sizeof}(\ast p) \)
  • \( p – n \) : result is a pointer to the object that lies \( n \) objects before \(*p\)
    • Calculation: \( p - n \times \text{sizeof}(\ast p) \)
  • \( p2 – p1 \) : result is the number of objects that can fit in between the two
    • \( p1 \) and \( p2 \) should be of the same type
    • Result is a signed integral value
    • Type of result is \texttt{ptrdiff_t} (defined in \texttt{stddef.h}), which is implementation dependent
    • Result is undefined if \( p1 \) and \( p2 \) do not point to objects in the same array (or not aligned)
Precedence of & and * Operators

The unary address & and dereferencing * operators have equal precedence, which is the same as that of the other unary operators, and they associate from right to left. You have to be careful when you mix * with ++ or – in a pointer expression. Thus, if cp is a pointer to char, then

```c
*++cp
```

is interpreted as

```c
*(++cp)
```

and it first increments cp and then fetches the character it points to, but the expression

```c
*cp++
```

is interpreted as

```c
*(cp++)
```

and it first fetches the character cp points to and then increments cp.

• Common C idioms (important): Textbook, Section 7.1.6 (page 214)
Pointer Comparison

• `==`, `!=` can compare
  • Pointers of the same type
  • A void pointer and any other pointer
  • NULL and any pointer
  • Pointers are equal if they point to the same object or if both are NULL
    • Basically and simply, just compare the values of the pointer variables

• `<`, `<=`, `>`, `>=` can compare
  • Pointers of the same type
  • Comparison result is portable only if
    • Objects pointed to lie within the same array, or
    • Addresses are aligned as if they point to objects in the same array

• Remember: NULL is defined to be 0 (zero)

```c
if (ip == NULL) printf("null pointer, eh?\n");
can equivalently be written as
if (! ip) printf("null pointer, eh?\n");
```
Example

```c
int arraySum ( int array [], int length )
{
    int sum = 0          ;
    int * ap = & array[0] ;

    while ( ap < & array[length] )
        sum += *ap++;  /* Add *ap to sum, and then increment ap */

    return sum ;
}

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

    printf( "Sum of array entries = %d\n" , arraySum( a, 4 ) ) ;
    return 0 ;
}
```
Pointer Conversion

• Pointers can be explicitly cast to other pointer types; (type *)

```c
double d = 3.14 ;
int   * ip ;
double * dp = &d ;
ip = (int *) dp ;
printf( "%d , %d\n" , *ip , *(ip + 1) ) ;
dp = (double *) ip ;
printf( "%lf\n" , *dp ) ;
```

• Be careful about possible addressing exceptions (textbook, page 216)
Functions and Pointers

• Functions can take and/or return pointers to any data type

• Remember: C only supports call/pass by value mechanism
  • This also applies to pointers, there are no exceptions

  ```c
  double *maxp(double *xp, double *yp)
  {
      return *xp >= *yp ? xp : yp;
  }
  ```

• Dangerous case: returning address of a non-static local variable
  • The local variable will deactivate with the stack frame when the function ends
  • Correct way: may return address of static local variable, or make use of heap

  ```c
  double u = 1, v = 2, *mp;
  the statement
  mp = maxp(&u, &v);
  makes mp point to v
  ```
Arrays and Pointers

• Array name is an address (**almost** equivalent to a pointer)
  • **Important**: Array name is **not** a variable (no storage box), it is an **R-value**
  • A pointer variable can be used both as an **R-value** and an **L-value**

• Array names and pointers can **almost** be used interchangeably
  • We can subscript pointers, like arrays, for example: `ip[4]`
  • We can do pointer arithmetic with array names, for example: `*(a+2)`

```
char *cp, c[MAX]; int i;

<table>
<thead>
<tr>
<th>Array Notation</th>
<th>Pointer Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;c[0]</td>
<td>c</td>
</tr>
<tr>
<td>c [i]</td>
<td>*(c+i)</td>
</tr>
<tr>
<td>&amp;c[i]</td>
<td>c+i</td>
</tr>
<tr>
<td>cp[i]</td>
<td>*(cp+i)</td>
</tr>
</tbody>
</table>
```
Arrays as Function Arguments

• Passing “array of T” to a function, similar to, passing “pointer to T”

```c
int max(int *a, int length)
{
    int i, maxv;
    for (i = 1, maxv = a[0]; i < length; i++)
        if (a[i] > maxv) maxv = a[i];
    return maxv;
}
```

• Brainstorming examples: `printf( “Hello” + 1 );` `scanf( “%d” + 1 , &k );`
Strings and Pointers

• C Language does not have a built-in string data type

• A string in C is a null-terminated array of characters
  • `char robot[5] = { 'r' , '2' , 'd' , '2' , '\0' } ;`
  • `char robot[5] = "r2d2" ;  /* shortcut, same as above, NO string constant; initializer */`
  • `char robot[] = "r2d2" ;  /* same as above */`
  • `char robot[4] = "r2d2" ;  /* Still no string constant, but does not have '\0' at end */`
  • `char * robot = "r2d2" ;  /* DIFFERENT, points at a string constant, read-only */`

• A string constant is an “array of char”

• String constants are usually stored in read-only memory segments

```c
int strlen(char str[])
{
    int i = 0;
    while (str[i] != '\0') i++;
    return i;
}

int strlen(char *str)
{
    char *first = str;
    while (*str != '\0') str++;
    return str - first;
}

void strcpy(char *to, char *from)
{
    while(*to = *from) to++, from++;
}
```
## Library Functions for Processing Strings

- Prototypes are in `string.h`
- `const` means the argument will not be modified (more in Chapter 12)
- `size_t` is an unsigned integral type defined in `stddef.h`
- List and explanations: textbook, page 229-230

### Code Examples:

```
char si[MAX], s2[MAX];

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%d&quot;, strlen(&quot;cord&quot;));</code></td>
<td>4</td>
</tr>
<tr>
<td><code>printf(&quot;%s&quot;, strcpy(sl, &quot;string&quot;));</code></td>
<td>string</td>
</tr>
<tr>
<td><code>printf(&quot;%s&quot;, strncpy(s2, &quot;endomorph&quot;, 4));</code></td>
<td>endo</td>
</tr>
<tr>
<td><code>printf(&quot;%s&quot;, strcat(si, s2));</code></td>
<td>stringendo</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, strcmp(si, s2));</code></td>
<td>1</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;, strcmp(sl+6, s2, 4));</code></td>
<td>0</td>
</tr>
<tr>
<td><code>printf(&quot;%s&quot;, strchr(si, 'n'));</code></td>
<td>ngendo</td>
</tr>
<tr>
<td><code>printf(&quot;%s&quot;, strrchr(si, 'n'));</code></td>
<td>ndo</td>
</tr>
</tbody>
</table>
```
Multi-Dimensional Arrays and Pointers

• Some compilers generate more efficient code with pointers
• All arrays are stored as one-dimensional in memory
• Array name is a pointer to the first row
• Array operations can be implemented as pointer operations, or vice versa, as usual:
  • matrix[i][j] is equivalent to *( *(matrix + i) + j )

```c
int matrix[4][2]; /* matrix is a pointer to first row */
    /* Each row is an array of 2 char elements */

int (* rowPtr)[2]; /* rowPtr is a pointer to an array of 2 chars */
    /* Note: parentheses are necessary (precedence) */

rowPtr = matrix + 2; /* rowPtr points to third row of the matrix */

(* rowPtr)[1] = 999; /* Equivalent to *( (* rowPtr) + 1 ) = 999 */
```
Examples

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>is a pointer to the first row;</td>
</tr>
<tr>
<td>matrix+i</td>
<td>is a pointer to the i\textsuperscript{th} row;</td>
</tr>
<tr>
<td>*(matrix+i)</td>
<td>is the i\textsuperscript{th} row which is converted into a pointer to the first element in the i\textsuperscript{th} row;</td>
</tr>
<tr>
<td>*(matrix+i)+j</td>
<td>is a pointer to the j\textsuperscript{th} element in the i\textsuperscript{th} row; and</td>
</tr>
<tr>
<td><em>((</em>(matrix+i) + j)</td>
<td>is matrix[i][j], the j\textsuperscript{th} element in the i\textsuperscript{th} row.</td>
</tr>
</tbody>
</table>

```c
int colsum(int (*matrix)[MAXCOLS],
           int rows, int column)
{
    int (*rptr)[MAXCOLS] = matrix;
    int i, sum;

    for (i = 0, sum = 0; i < rows; i++)
        sum += (*rptr++)[column];
    return sum;
}
```
Pointer Arrays

• An ordinary array, elements of which are pointers

```c
char *day[7] = {
    "monday", "tuesday", "Wednesday",
    "thursday", "friday", "Saturday", "sunday"
};

char *dayname(int n) {
    return n >= 0 && n <= 6 ? day[n] : NULL;
}
```
Command-Line Arguments

- main function can also take arguments (usually named argc and argv)
  - argc : argument count
  - argv : argument vector (the first string is the name of the program itself)
- If an argument contains whitespace, use " " :
- All arguments are always stored as character strings
  - use atoi, atof functions (stdlib.h) to convert to numbers

```c
int main(int argc, char **argv)
{
    ...
}
```

```
printargs vox populi vox Dei
```

```
char **argv
```
Pointers to Pointers

• A pointer variable can contain the address of another pointer
• There is no limit on the number of levels of indirection

```c
int i = 1;
int *p;
int **q;
p = &i;
q = &p;
```

```c
#include <stdio.h>

int main ( int argc , char ** argv )
{
    while ( --argc ) printf( "\s" , *++argv ) ;
    printf( "\n" ) ;
    return 0 ;
}
```

### Diagram

- **q**: Pointer to an integer
- **p**: Pointer to an integer
- `i`: Integer

**Diagram Explanation**

- `int ***p;` means
  - `***p` is an integer,
  - `**p` is a pointer to an integer,
  - `*p` is a pointer to a pointer to an integer, and
  - `p` is a pointer to a pointer to a pointer to an integer.
Pointers to Functions

• A function’s name is the address of its machine code in TEXT segment

• We can define pointer variables to point to any function
  • Example:  int (* fp) ( int paramName1 , int paramName2 ) ;
  • Parameter names are optional (just to give more info to the reader)
  • Parentheses are required (otherwise meaning is different due to precedence)
  • fp is a “pointer to a function that takes two integers and returns an integer”

• Examples:

\begin{verbatim}
int (*fp)(int, int) , gcd(int, int) ;
fp = gcd ;
result = (*fp)(42, 56) ;
\end{verbatim}

\begin{verbatim}
void (*gp)(void) , initialize(void) ;
gp = initialize ;
(*gp)() ;
\end{verbatim}
Dynamic Memory Allocation

- In many situations, we don’t know the amount of the data that we need to process (data may come from user at run-time)
- We may try to predict maximum size of data and arrange storage accordingly, but this approach usually wastes memory, or sometimes our predictions may fall short of actual data size
- C language allows us to allocate/deallocate storage dynamically
  - Functions for allocation: `malloc`, `calloc`
  - Function for changing size of allocated storage: `realloc`
  - Function for deallocation (releasing unused memory): `free`
  - All four functions declared in `stdlib.h`
  - Useful operator frequently needed: `sizeof`
sizeof Operator

- **sizeof** is a unary operator

- **Syntax:**
  - `sizeof (typename)` → size, in bytes, of an object of the given type
  - `sizeof expression` → analyzes expression at compile time to determine its type
    → yields same result as if applied to expression’s type
    → does NOT execute expression (side effects do NOT occur)

- **Value returned is of type** `size_t` (an unsigned integral type, in `stddef.h`)

- If operand is a **n-element array of type** `T`, result is `n × sizeof(T)`
  - `int a[10] ; sizeof( a )` → 20 or 40 (depending on compiler)
  - `sizeof( “computer” )` → 9 (string constant is a null-terminated array of chars)
The `sizeof` operator does not cause any of the usual type conversions in determining the type of the expression. For example, when applied to an array name, `sizeof` does not cause the array name to be converted to a pointer. However, if the expression contains operators that do perform usual type conversions, those conversions are taken into account in determining its type. For example, if

```c
char c;
```

then `sizeof (c) is the same as sizeof (char),
but `sizeof (c+0) is the same as sizeof (int).

because the type of the expression `c+0`, after the usual type conversion, is `int`. 
Dynamic Memory Management Functions

```c
void *malloc (size_t size);
```

The function `malloc` allocates storage for an object whose size is specified by `size`. It returns a pointer to the allocated storage and `NULL` if it is not possible to allocate the storage requested. The allocated storage is not initialized in any way.

```c
void *calloc (size_t nobj, size_t size);
```

The function `calloc` allocates the storage for an array of `nobj` objects, each of size `size`. It returns a pointer to the allocated storage and `NULL` if it is not possible to allocate the storage requested. The allocated storage is initialized to zeros.
Dynamic Memory Management Functions

```c
void *realloc (void *p, size_t size);

The function `realloc` changes the size of the object pointed to by `p` to `size`. It returns a pointer to the new storage and `NULL` if it is not possible to resize the object, in which case the object (*p) remains unchanged. The new size may be larger or smaller than the original size. If the new size is larger, the original contents are preserved and the remaining space is uninitialized; if smaller, the contents are unchanged up to the new size.

The function `realloc` behaves like `malloc` for the specified size if `p` is a `NULL` pointer.
```

```c
void free(void *p) ;

The function `free` deallocates the storage pointed to by `p`, where `p` is a pointer to the storage previously allocated by `malloc`, `calloc`, or `realloc`. If `p` is a null pointer, `free` does nothing.
```
Dynamic Memory Management Functions

The behavior of the functions `realloc` and `free` is undefined, if \( p \) does not match a pointer earlier returned by a call to `malloc`, `calloc`, or `realloc`, or if the storage has been deallocated by a call to `realloc` or `free`.

- `malloc`, `calloc`, `realloc` return address (void *) of the allocated space, or NULL if unsuccessful (if there is not enough free memory).
- Return value can be safely type-casted to any desired pointer type.
- Order and contiguity of storage allocated by successive calls to `malloc`, `calloc`, `realloc` is not specified.
Chapter 8 – Structures and Unions

• Basics of Structures
  • Struct variables, initialization, accessing members, assignment, size, nested structs, pointers to structs

• Structures and Functions
  • Scope of struct definition, struct function arguments, struct function values

• Structures and Arrays
  • Arrays of structs, structs containing arrays, arrays of structs containing arrays

• Structures Containing Pointers
  • Self-referential structs

• Unions
Structure (Struct) Basics

• Arrays group data items of same type into a single object, similarly structs group any related data items (might be of different types)

• A structure is a collection of logically related data items grouped under a single name, called a structure tag

• Struct members (or components, or fields) can be of different types

• Struct definition syntax:

  ```
  struct tag
  {
    variable declarations
  } ;
  ```

• Defining a struct defines a new type, it is NOT a variable definition
  • No storage space is allocated when we create a new data type
Struct Variables

• Variables of a specific struct type can be defined in 2 ways
  • Right after struct definition:
  • Later on, by referring to the struct tag:

```
struct tag
{
  variable declarations
} var1, var2;
```

• Struct tag is optional, but if omitted, we can only use 1st way above

• Each struct
• Definition
• Introduces
• a new data type

```
struct { char c; int i; } a; /* a,b,c,d are all */
struct { char c; int i; } b; /* of different types */
struct tag1 { char c; int i; } c;
struct tag2 { char c; int i; } d;
struct tag2 { char c; int i; } e; /* Redefinition Error */
struct tag2 e;
```

```
a = b; /* Gives error */
b = c; /* Gives error */
c = d; /* Gives error */
d = e; /* OK */
```
Struct Initialization and Member Access

- When we define a struct variable, we can also initialize it immediately

```c
struct date
{
    int day, month, year;
} independence = {15, 8, 1947};
```

- If there are fewer initializers, remaining members are initialized to 0
- If there are more initializers → Error!
- Struct members can be accessed by **dot (structure member) operator**
  - Syntax: `structVariable.memberName`
  - Dot operator has same precedence as `()`, `[]`, `-`
  - Dot operator is left-associative

```c
struct date newyear = {1, 1};
```

```c
struct date man_on_moon;
man_on_moon.day = 20;
man_on_moon.month = 7;
man_on_moon.year = 1969;
```
Struct Assignment

• Struct variables of the same type can be assigned to each other
  • This is similar to assigning a variable to another of the same type
  • All contents (values of all members, including arrays) are copied
  • Compared to alternatives below, this way of assignment is more natural
  • This way and also alternatives below do a shallow copy, but NOT deep copy!
    • Be careful when the struct contains one or more pointer member variables
    • For deep copying, we may need to create a function to deep copy a given struct variable

• Alternatives
  • Member-wise assignment
  • Memcpy

• Check out sample code on COW page: StructAssignment.c
Size of a Structure

• We can use sizeof operator to learn the size of a struct
• Size of a struct might be greater than the sum of sizes of its members
  • Due to alignment requirements, members might have gaps between

```c
struct porous
{
  char c ;
  int i ;
};

int size = sizeof( struct porous ) ; /* Most likely 8, rather than 5 */
```
Nested Structures

- Structures can be nested (a struct might be embedded in another)
  - We use repeated dot operator to access inner members
- There is no limit on the depth of nesting
- A struct CANNOT be nested within itself
  - However, a struct can contain a pointer to the same struct type

```c
struct project
{
  int number;
  struct date start;
  struct date finish;
  float budget;
};

struct company
{
  struct projects government;
  struct projects industrial;
  struct company parent; /* illegal */
};
```
Nested Structures – Initialization

- We can also initialize when defining a variable of a nested struct type

```c
struct stockindex
{
    float high, low, close;
    struct date weekending;
}
dowjones = {2520.79, 2381.19, 2520.79, {19, 10, 1990}};
```

The inner pair of braces is optional and certainly not necessary when all the initializers are present. Thus, we can have

```c
struct stockindex sp500 =
    (312.48, 296.41, 312.48, 19, 10, 1990);
```
Pointers to Structures

- The same as pointers to regular variables (keeps the starting address)
  - Used in data structures like linked lists, trees

- -> : structure pointer operator (arrow operator)
  - Syntax: `pointerName -> memberName`

```c
struct student
{
    char name[10] ;
    int id ;
    char courseNames[5][10] ;
} ;

struct student s1 = { "Selim" , 1234 , { "Ceng 140" , "Ceng 443" } } ;
struct student * p1 = &s1 ;

s1 .id = 5678 ;
(*p1).id = 5678 ; /* Same as above, parentheses are necessary */
p1->id = 5678 ; /* Same as above */
```
Structures and Functions

• Scope of a struct type definition:
  • Identical to discussions about variable names (in Chapter 5)
    • Can be local to a function (defined in a function), or
    • Can be external (defined outside of any function)
  • Struct definitions are usually collected in header file(s)
    • Header file(s) can be included in modules which need those struct definitions

• Structures as function arguments
  • Struct members can be passed to functions
  • Structs can be passed to functions
  • Pointers to structs can be passed to functions