CSC1110 Introduction to Computing

C Programming

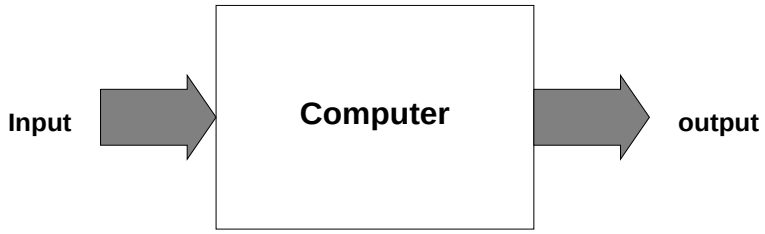
Prepared by Sau-Ming LAU.

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Contents

Lecture 1

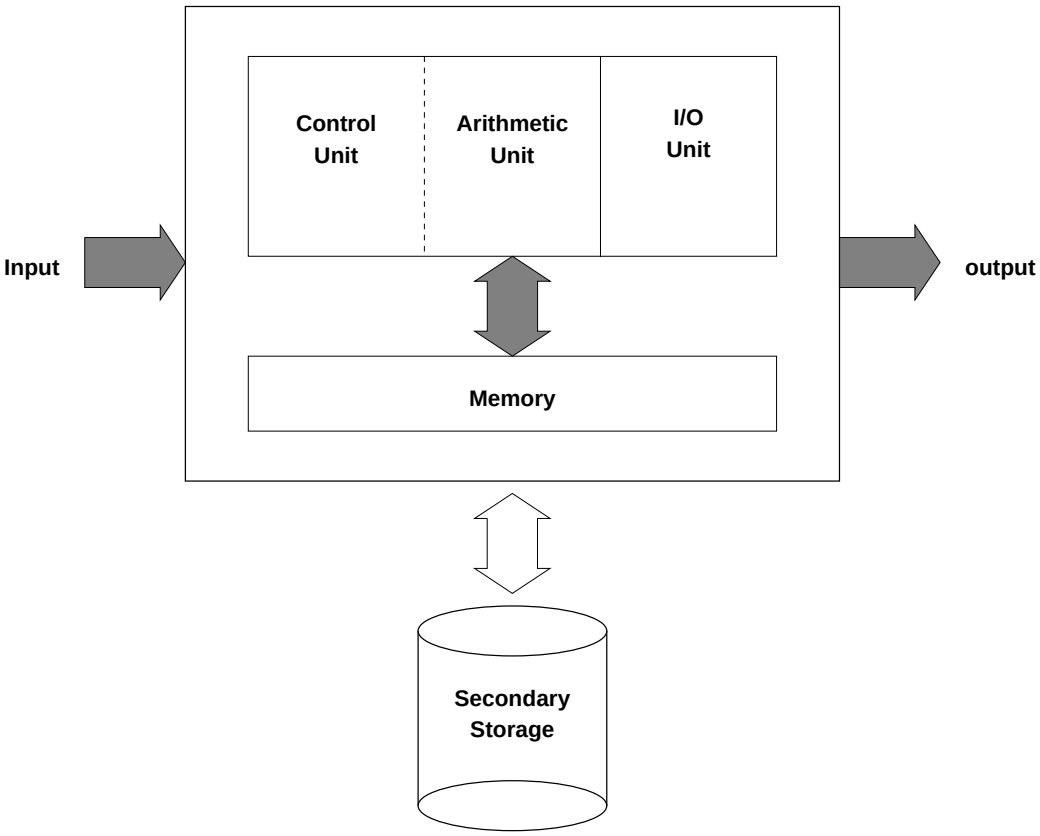
Introduction



What is a computer?

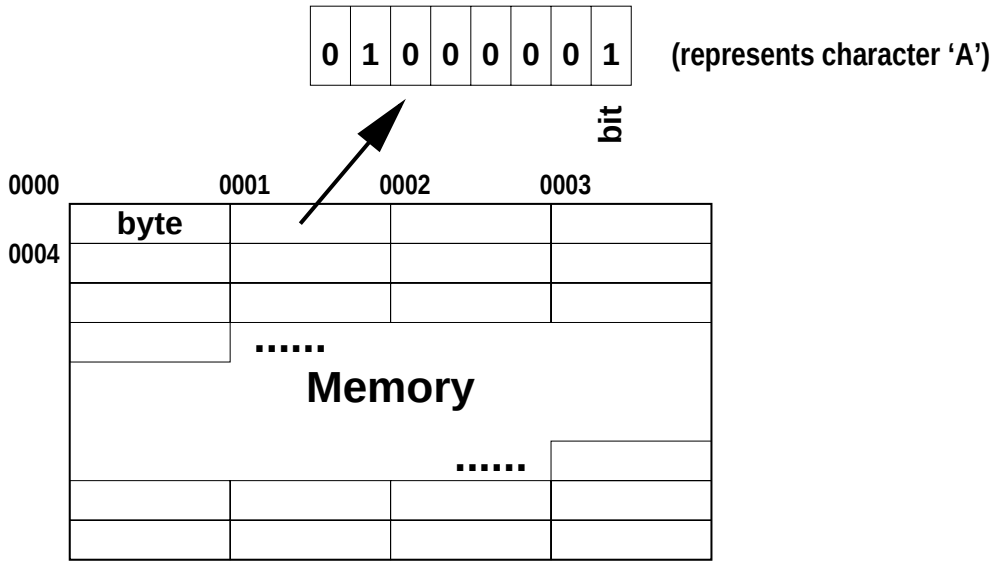
- A computer accepts external *input*, *processes* the data, and produces certain *output*.
- An *algorithm* is a "recipe" for processing information — a series of steps that transform input to output.

1.1 Basic Computer Architecture



Key Elements in a Computer

- **Input Unit**:
  - Obtains information (data and programs) from various input devices, such as keyboard and disk drive.
- **Output Unit**:
  - Places processed information onto various output devices, such as screen or printer.
- **Memory Unit**:
  - Retains information entered from the input unit.
  - Stores programs.
  - Stores intermediate results during processing.
- **Arithmetic and Logic Unit** (ALU):
  - Performs actual calculations and comparisons.
- **Control Unit**:
  - Supervises and coordinates the entire operations of a computer.
- **Secondary Storage**:
  - A device, such as tapes or disks, for storing programs or data that are not actively being used by other units.



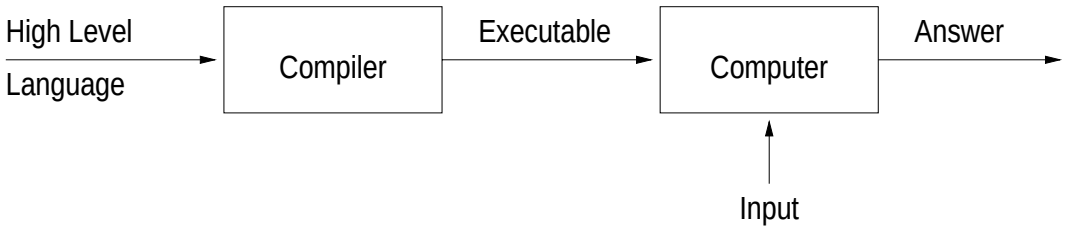
Memory Organization — Bits & Bytes

- In the lowest level, computers understand only 0's and 1's.
- Computer memory stores a sequence of 0's and 1's.
- Each such storage unit is called a *bit*.
- Eight bits are grouped together to form a larger storage unit called a *byte*.
- Each byte has a unique *address* for identification.
- Computer memory can thus be viewed as a series of bytes.

1.2 Computer Programming

Computer Programming

- The job of a programmer is to design a *complete* and *precise* set of *instructions* for a computer to carry out a particular problem-solving task.
- A programming language is a medium for programmers to express instructions to the computer.
- Do we need to write our programs in 0's and 1's?!



High Level Programming Languages

- Fortran, Cobol, Pascal, Modula-2, Ada, C, Smalltalk, C++, Prolog, Lisp, Perl, Java, Visual Basic, ...
- Compilation: High level languages require the use of a *compiler* to translate into machine executable codes.

□ **End.**

Lecture 2

An Overview of the C Language

History of the C Language

- C is a *general purpose* programming language.
- Small and relatively easy to learn. (?)
- Designed by Dennis Ritchie of Bell Labs in 1972.
- C Standards:
  - Traditional Unix C
  - "*The C Programming Language*" by Kernighan and Ritchie, 1978.
  - ANSI (American National Standards Institute)

2.1 Your First C Program

Program over_1.c

```
/* The traditional first program in honor of
   Dennis Ritchie who invented C at Bell Labs
   in 1972 */

#include <stdio.h>

int main(void)
{
   printf("Hello, world!\n");
   return(0);
}
```

```
Hello, world!
```

2.2 Your Second C Program

Program over_2.c

```
#include <stdio.h>

int main(void)
{
   printf("Hello, world!\n");
   printf("Hello, universe!\n");
   return(0);
}
```

```
Hello, world!
Hello, universe!
```

Remember!

- We must tell the computer all the steps involved in a precise way.

Exercise

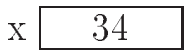
Write a program that displays the following two sentences:

```
Read it:
   She sells sea shells by the seashore.
```

2.3 Variables, Expressions, and Assignments

What is a Variable?

- Storage cells in the main memory.
- A variable can be viewed as a *box* for holding some values.



- Depending on the type of data it is storing, a variable can occupy one or more bytes.

- Any sensible program maintains a number of variables.
- A program can be regarded as a "series" of program *statements* for modifying the values of variables.

Program over_3.c

```
#include <stdio.h>

int main(void)
{
   int   side, perimeter, area;

   side = 3;
   perimeter = 4 * side;
   area = side * side;

   printf("Side     : %d\n", side);
   printf("Perimeter: %d\n", perimeter);
   printf("Area     : %d\n", area);
   return(0);
}
```

```
Side     : 3
Perimeter: 12
Area     : 9
```

```
#include <stdio.h>

int main(void)
{
   int   side, perimeter, area;

   side = 3;
   perimeter = 4 * side;
   area = side * side;

   printf("Side     : %d\n", side);
   printf("Perimeter: %d\n", perimeter);
   printf("Area     : %d\n", area);
   return(0);
}
```

Variables

- `side`, `perimeter`, and `area` are called variables.
- The *type* of each variable must be specified. For example, `int` is the integer data type.
- A variable name consists of a sequence of letters, digits and underscores.
- A variable name should be meaningful.
- *Variable declaration* — Create a variable by giving it a name and specifying its type.

```
#include <stdio.h>

int main(void)
{
   int   side, perimeter, area;

   side = 3;
   perimeter = 4 * side;
   area = side * side;

   printf("Side     : %d\n", side);
   printf("Perimeter: %d\n", perimeter);
   printf("Area     : %d\n", area);
   return(0);
}
```

```
int    side, perimeter, area;
```

(*Variable declarations*)

?	?	?
side	perimeter	area

```
side = 3;
```

(*Assignment statement*)

3	?	?

```
perimeter = 4 * side;
```

(*Expression* on the R.H.S. *evaluates* to 12.)

3	12	?

```
area = side * side;
```

(R.H.S. expression evaluates to 9.)

3	12	9

```
#include <stdio.h>

int main(void)
{
   int   side, perimeter, area;

   side = 3;
   perimeter = 4 * side;
   area = side * side;

   printf("Side     : %d\n", side);
   printf("Perimeter: %d\n", perimeter);
   printf("Area     : %d\n", area);
   return(0);
}
```

Output to the Screen

`printf("Side     :  %d\n", side);`

- This time, `printf()` needs two *arguments*.
- `"Side     :  %d\n"`
  - Called the *format string*.
  - The *format specifier* `%d` specifies that the value of the corresponding expression is to be printed in the format of a *d*ecimal integer.
- `side`

  The expression whose value is to be supplied to the format string.

```
#include <stdio.h>

int main(void)
{
   int   side, perimeter, area;

   side = 3;
   perimeter = 4 * side;
   area = side * side;

   printf("Side     : %d\n", side);
   printf("Perimeter: %d\n", perimeter);
   printf("Area     : %d\n", area);
   return(0);
}
```

The general form of a simple C program

```
# preprocessing directives

int main(void)
{
    variable declarations

    statement_1;
    statement_2;
    statement_3;
    statement_4;
    ...
    return(0);
}
```

□ **End.**

Lecture 3

Basic C Syntax

What is Language Syntax?

- C, as a language, has a set of rules for putting together words and punctuations to make correct programs.
- These rules are the *syntax* of the language.
- For a program to be compiled successfully, it must be *syntactically correct*.
- C compilers will fail to compile a syntactically incorrect program, no matter how trivial the error is.

A Program with Typos

```
#include <stdio.h>

int main(void)
{
     int    side, perimeter, area,

     side = 3;
     perimeter = 4 * side;
     area = side * side;

     printf("Side    :  %d\n", side);
     printf("Perimeter  %d\n", perimeter);
     printf("Area    :  %d\n", area);

     return(0);
}
```

During Compilation

```
square.c:  In function ‘main’:
square.c:7:  redeclaration of ‘side’
square.c:5:  ‘side’ previously declared here
```

3.1 Lexical Elements

Program basic_1.c

```
/* Read in two integers and print their sum. */

#include <stdio.h>

int main(void)
{
   int   a, b, sum;

   printf("Input two integers:  ");
   scanf("%d%d", &a, &b);
   sum = a + b;
   printf("%d + %d = %d\n", a, b, sum);
   return (0);
}
```

Tokens in the C Language

- The compiler views a program as a series of *tokens*.
- Identifying and categorizing tokens in a program allow the compiler to analyze the syntax of the program.
- In ANSI C, there are six kinds of tokens.

Token Type	**Example**
Keyword	`int, return, void`
Identifier	`main, a, b, sum, printf, scanf`
Constant	`0`
String constant	`"Input two integers:  "`
Operator	`(), =, +, &`
Punctuator	`, ; { }`

3.1.1 Keywords

What is a Keyword?

- Keywords have a strict meaning.
- Cannot be redefined or used in other ways, i.e. *reserved*.
- Compared to other major high level programming languages, C has a small number of keywords — 32 only.

ANSI C Keywords

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

3.1.2 Identifiers

Identifiers Naming Rules

- Identifiers are used to give unique names to various objects (such as a variable) in a program.
- An identifier is a token that is composed of a sequence of letters, digits, and the underscore (_).
- The first letter of an identifier must be a letter or an underscore.
- Example identifiers:
  `inches, radius, _id, student_name, main, printf, scanf`
- Invalid identifiers:
  `#num, 101_south, -plus`
- Lower- and upper-case letters are treated as distinct, i.e. case-sensitive:
  `my_name` is different from `My_name`.
- Choose identifiers that have *mnemonic* significance, for example,
  `tax = price * tax_rate;`
- Identifiers starting with an underscore (_) are not recommended, for example, `_job`.
- Identifiers such as `scanf` and `printf` have already been defined in the *standard library* and should not be redefined.
- `main` is a special identifier to signify where program execution should start.

Exercise

Which of the following are not valid identifiers and why?

`3id`	valid?	invalid?
`o_no_o_no`	valid?	invalid?
`00_go`	valid?	invalid?
`start*it`	valid?	invalid?
`__yes`	valid?	invalid?
`1_i_am`	valid?	invalid?
`one_i_aren't`	valid?	invalid?
`me_to-2`	valid?	invalid?
`main`	valid?	invalid?
`xYshouldI`	valid?	invalid?
`int`	valid?	invalid?

3.1.3 Constants

Categories of Constants

- *Decimal integer constants*:

  `0` `17`

- *Floating constants*:

  `1.0` `3.14`

- *Character constants*:

  `'a'` `'b'` `'\n'` `'\t'`

- String constants (next page)
- Enumeration constants (discuss later)

3.1.4 String Constants

What is a String?

- A string constant is a sequence of characters enclosed in a pair of double quote ", for example,

  `"C is easy to learn."`

- String constants are different from character constants — `"a"` is different from `'a'`.

Program basic_2.c

```
#include <stdio.h>

int main(void)
{
        printf ("This is a string.\n");
        printf ("\n");
        printf ("a string with \"double quotes\".");
        printf ("\n");
        printf ("a single backslash \\ is in this string.");
        printf ("\n");
        printf ("a = b + c;");
        printf ("\n");
        return (0);
}
```

```
This is a string.
a string with "double quotes".
a single backslash \ is in this string.
a = b + c;
```

Be Careful!!!

Character sequences that would have meaning if outside a string constant are just a sequence of characters when surrounded by double quotes.

3.1.5 Operators and Punctuators

Categories of Operators

- *Arithmetic operators*:

```
+ - * /
```

- *Binary operators*:

```
a + b
17 * 3
x = y
```

- *Unary operators*:

```
x++
&sum
```

- Operators can be used to *delineate* identifiers, for example,

```
sum=a+b;
```

- But we typically put *white space* around binary operators to improve *readability*.

```
sum = a + b;
```

Punctuators

Examples of punctuators are:
parentheses `()`, braces `{ }` , commas, and semicolons.

3.2 More on Operators

3.2.1 Operator Precedence and Associativity

Basic Idea

- Operators have *precedence* and *associativity* that determine precisely how expressions are evaluated.
- `1 + 2 * 3`
  The operator `*` has *higher precedence* than `+`, causing the multiplication to be performed first.
- `(1 + 2) * 3`
  Parenthesis can be used to change the order in which operations are performed.
- `1 + 2 - 3 + 4 - 5`

  is equivalent to

  `(((1 + 2) - 3) + 4) - 5`.

  Because `+` and `-` have the same precedence, the expression is evaluated using the associativity rule "*left to right*."

Operator precedence and associativity	
Operator	**Associativity**
() ++ (postfix) −− (postfix)	left to right
+ (unary) − (unary) ++ (prefix) −− (prefix)	right to left
* / %	left to right
+ −	left to right
= += −= *= /= etc.	right to left

How to use the table?

- All the operators on a given line, such as `*`, `/`, `%`, have equal precedence with respect to each other.
- Operators on a given line have higher precedence than all the operators that occur on the lines below.
- When evaluating expressions involving operators on the same level of precedence, refer to the associativity.
- `- a * b - c` is equivalent to `((- a) * b) - c`

3.2.2 The Increment and Decrement Operators

The Increment Operator

- The *increment operator* `++`.
- The `++` operator can be applied to variables, but not to constants or ordinary expressions, for example,

  ```
  i++ /* valid */
  ```

  ```
  777++ /* invalid */
  ```

- The `++` operator increases the value of the variable that it applies to by one.
- Example,

  ```
  value = 10;
  value++;
  printf ("%d\n", value);
  ```

  This program fragment causes 11 to be printed.

PreFix and PostFix Forms

- The `++` operator can occur in either *prefix* or *postfix* position, with different results.
  - Each of the expression `++i` and `i++` has a value.
  - The prefix expression `++i` causes the stored value of `i` to be incremented (by one) first, then the expression evaluates to the newly stored value of `i`.
  - The postfix expression `i++` evaluates to the current value of `i` first, then the stored value of `i` is incremented (by one).
- Consider the example:

```
int a, b, c=0;

a = ++c;
b = c++;
printf ("%d %d %d\n", a, b, ++c);
```

  This program fragment causes `1 1 3` to be printed.

The Decrement Operator

- The *decrement operator* `--`.
- The `--` operator is similar to `++`, except that the associated variable is decremented by one.

Exercise

Evaluate the expressions in the table below. Refer to the *Operator Precedence and Associativity Table* shown on Page 28.

Declarations and Initializations		
`int a=1, b=2, c=3, d=4;`		
Expression	**Equivalent Expression**	**Value**
`a * b / c`	`(a * b) / c`	0
`a * b % c + 1`		
`++ a * b - c --`		
`7 - - b * ++ d`		

3.2.3 Miscellaneous Assignment Operators

The Assignment Expression

`variable = right_side_expression`

- Low precedence, right-to-left associativity
- The value of the `right_side_expression` is assigned to `variable`.
- AND the value becomes *the value of this assignment expression* as a whole. Refer to the example below.

Program basic_3.c

```
#include <stdio.h>

int main(void)
{
    int a=3, b=4, sum;

    printf ("Value = %d\n", sum=a+b);
    printf ("Sum = %d\n", sum);
    return (0);
}
```

```
Value = 7
Sum = 7
```

Multiple Assignment

- *Multiple assignment*, for example,

  ```
  a = b = c = 0;
  ```

- First `c` is assigned the value `0`, and the expression `c = 0` has value `0`. Then `b` is assigned the value `0`, and the expression `b = (c = 0)` have value `0`. Finally `a` is assigned the value `0`, and the expression `a = (b = (c = 0))` has value `0`.

Assignment Operators — Short Form

- `k = k + 2` is equivalent to `k += 2`.
- The *semantics* of

  ```
  variable = variable op (expression)
  ```

  is equivalent to

  ```
  variable op= expression
  ```

Assignment operators						
=	+=	−=	*=	/=	%=	etc.

Be Careful!!

Note that the following statement

```
j *= k + 3;
```

is equivalent to

```
j = j * (k + 3);
```

rather than

```
j = j * k + 3;
```

3.3 Symbolic Constants

Program basic_4.c

```
#include <stdio.h>
#define PI 3.14

int main(void)
{
   float  radius=1.5;

   printf ("Radius=%.1f\n", radius);
   printf ("Area  =%.2f\n", radius*radius*PI);
   return (0);
}
```

```
Radius=1.5
Area  =7.07
```

What are Symbolic Constants?

- `PI` is called a *Symbolic Constant*.
- The preprocessor changes all occurrences of the identifier `PI` to 3.14.
- Thus,

  `printf ("Area =%.2f\n", radius*radius*PI);`

  becomes

  `printf ("Area =%.2f\n", radius*radius*3.14);`

3.4 Comments

What are Comments?

- Comments are used as a *documentation* aid.
- The compiler ignores the comments.
- The aim of documentation is to explain clearly how the program works and how it is to be used.
- Comments should be written *simultaneously* when you write the program.

```
/* a comment */

/*** another comment ***/

/*****/

/*
 * A comment can be written in this fashion
 * to set it off from the surrounding code.
 */

/*****************************
 * If you wish, you can      *
 * put comments in a box.    *
 *****************************/

/*
     /* This is not allowed */
*/
```

3.5 The use of `printf()`

Basic Usage of `printf()`

- `printf()` is used for printing *formatted output*.
- Two arguments:
  - *format string*
  - *a list of arguments*.
- The format string defines how many arguments are needed and how they are to be formatted.
- Each argument is specified by a *format specifier* — a `%` and a *conversion character*, such as `%d`.
- The data type of the arguments should match the corresponding specifier in the format string.

```
printf ("The data: %s %d %f %c\n", "one", 2, 3.33, 'G');
```

```
The data: one 2 3.330000 G
```

`printf()` conversion	
Character	**How the corresponding argument is printed**
c	as a character
d	as a decimal integer
e	as a floating-point number in scientific notation
f	as a floating-point number
g	in the e-format or f-format, whichever is shorter
s	as a string

Controlling Field-Widths in `printf()`

- When an argument is printed, the place where it is printed is called its *field*.
- The number of characters in a field is called the *field width*.
- The field width can be specified as an integer between the `%` and the conversion character.
- Refer to the example below.

```
printf ("%c%3c%7d\n", 'A', 'B', 9);
```

```
A  B      9
----------- This line is not produced by the above statement.
```

Controlling Precisions in `printf()` — `%f`

- For floating values, we can control the *precision*, as well as the field width.
- The precision is the number of digits to the right of the decimal point.
- In a format `%m.nf`, the field width is specified by `m`, and the precision is specified by `n`.
- Refer to the example below.

```
printf ("Some numbers: %.1f %.2f %.3f\n", 1.0, 2.0, 3.0);
printf ("More numbers:%7.1f%7.2f%7.3f\n", 4.0, 5.0, 6.0);
```

```
Some numbers: 1.0 2.00 3.000
More numbers:    4.0   5.00  6.000
```

The Use of Expressions in `printf()`

- An expression can also be used as an argument in `printf()`.
- The expression is first evaluated and the resulting value is then matched with the corresponding format specifier in the format string.

Program basic_5.c

```
#include <stdio.h>

int main(void)
{
   float   x, y;

   x = 1.0;
   y = 2.0;
   printf("The sum of %.1f and %.1f is %.3f!\n", x, y, x + y);
   return (0);
}
```

```
The sum of 1.0 and 2.0 is 3.000!
```

3.6 Variable Initialization

Program basic_6.c

```
#include <stdio.h>

int main(void)
{
   float  radius;

   radius=1.5;
   printf ("Radius=%.1f\n", radius);
   printf ("Area  =%.2f\n", radius*radius*3.14);
   return (0);
}
```

```
Radius=1.5
Area  =7.07
```

Program basic_7.c

```
#include <stdio.h>

int main(void)
{
   float  radius=1.5;

   printf ("Radius=%.1f\n", radius);
   printf ("Area  =%.2f\n", radius*radius*3.14);
   return 0;
}
```

Variable Initialization

- Compare the program above to the program on the previous page.
- When a variable is declared, it may be *initialized* to a particular value.
- For example,

  ```
  float    radius=1.5;
  ```

- is equivalent to

  ```
  float radius;
  radius=1.5;
  ```

3.7 The use of `scanf()`

```
#include <stdio.h>

int main(void)
{
   float  radius=1.5;

   printf ("Radius=%.1f\n", radius);
   printf ("Area  =%.2f\n", radius*radius*3.14);
   return (0);
}
```

- What if the radius is .....?

The use of `scanf()`

- Analogous to `printf()`, but is used for input rather than output.
- The format string defines how many arguments are needed and how the data in the *input stream* are to be interpreted.
- Corresponding to each format specifier in the format string is the *memory address* of the variable going to store the input data.
- Example below,

```
scanf ("%d", &num);
```

- The format specifier `%d` causes input characters to be interpreted as a decimal integer.
- The value of the decimal integer is stored in the variable `num`.

`scanf()` conversion

Character	How characters in the input stream are converted
c	character
d	decimal integer
f	floating-point number (`float`)
lf	floating-point number (`double`)
Lf	floating-point number (`long double`)
s	string

Program basic_8.c

```
#include <stdio.h>

int main(void)
{
   float  radius;

   printf ("Input radius? ");
   scanf ("%f", &radius);
   printf ("Area = %.2f\n", radius*radius*3.14);
   return (0);
}
```

```
Input radius? 1.5
Area = 7.07
```

```
Input radius? 3.4
Area = 36.30
```

Exercise

Below is part of a program that begins by asking the user to input three integers. Complete the program so that when the user executes it and types in 2, 3, and 7, the screen looks like:

```
Input three integers:
1st?  2
2nd?  3
3rd?  7
Sum of the integers = 12
```

```
#include <stdio.h>

int main(void)
{
   int a, b, c;

   printf ("Input three integers: \n");
   printf ("1st? ");
   scanf("%d", &a);
   ...

   return (0);
}
```

□ **End.**

Lecture 4

Flow of Control — I

Categories of Flow Controls

- *Relational, Equality, Logical Operators* are provided to facilitate flow controls.
- *Sequential* — Statements in a program are executed one after another.
- *Selection* — A choice of alternative actions (`if`, `if-else`, and `switch`).
- *Repetition* — Repeat certain actions (`while`, `for`, and `do`).

NOTE

Repetition will be covered in Lecture 5 (Flow of Control — II).

4.1 Relational Operators and Expressions

Relational Operators

< > <= >=

- All *relational operators* are binary operators, taking two operands.

Relational Expressions

- A *relational expression* is an expression that involves the use of one or more relational operators.
- Relational expressions yield either *true* or *false*.
- "True" is represented by integer 1.
- "False" is represented by integer 0.

Examples

Relational Expressions		Expression Value	
$7 < 9$		true	1
if a is 5,	$a > 3$	true	1
if a is 2,	$a > 3$	false	0
if b is 40 and c is 6,	$b <= (9 * 3 + c)$	false	0

Declarations and Initializations
`int i=1, j=2, k=3;`
`double x=5.5, y=7.7;`

Expression	**Equivalent Expression**	**Value**
`i < j - k`	`i < (j - k)`	0
`- i + 5 * j >= k + 1`		
`x - y <= j - k - 1`		
`x + k + 7 < y / k`		

Operator precedence and associativity	
Operator	**Associativity**
() ++ (postfix) −− (postfix)	left to right
+ (unary) − (unary) ++ (prefix) −− (prefix) !	right to left
* / %	left to right
+ −	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += − = * = /= etc.	right to left
, (comma operator)	left to right

4.2 Equality Operators and Expressions

Equality Operators / Expressions

==	!=
(Equal-to)	(Not-Equal-to)

- The *equality operators* are binary operators, taking two operands.
- A *equality expression* yields either 1 (true) or 0 (false).
- Examples,

```
9 == 10
c == 'A'
'A' != 'B'
k != -2
area != (length * length)
```

Declarations and Initializations

```
int     i=1, j=2, k=3;
```

Expression	Equivalent Expression	Value
`i == j`	`j == i`	0
`i != j`		
`i + j + k == - 2 * - k`		

4.3 Logical Operators and Expressions

Logical Operators / Expressions

!	&&	||
(NOT)	(AND)	(OR)

- Operator ! is unary, taking only one operand.
- Operators && and || are binary, taking two operands.
- *Logical expressions* yield either 1 (true) or 0 (false).

Expression as an Operand to Logical Expressions

Note carefully that when an expression is used as an operand to a logical expression,

- if the operand expression has the value *zero*, it will be regarded as *false* when evaluating the logical expression.
- if the operand expression has a *non-zero* value, it will be regarded as *true* when evaluating the logical expression.

4.3.1 The NOT (!) Operator

Examples		
NOT **Expressions**		**Value**
!9		0
!0		1
if a is 3,	$!a$	0
if x is 2,	$!(x + 7.7)$	0
if $a < b$ is false and $c < d$ is true,	$!(a < b \mid\mid c < d)$	0

Questions

What is the value of the logical expression below?

```
!!5
```

And what will be printed by the following statement?

```
printf("%d", !!5);
```

Declarations and Initializations
`int i=7, j=7;`
`double x=0.0, y=999.9;`

Expression	**Equivalent Expression**	**Value**
`! (i - j) + 1`	`(! (i - j)) + 1`	2
`! i - j + 1`		
`! ! (x + 3.3)`		
`! x * ! ! y`		

4.3.2 The AND (&&) and OR (||) Operators

Declarations and Initializations
`int i=3, j=3, k=3;`
`double x=0.0, y=2.3;`

Expression	**Equivalent Expression**	**Value**
`i && j && k`	`(i && j) && k`	1
`x || i && j - 3`		
`i < j && x < y`		
`i < j || x < y`		

Textbook Page 93 - bottom table - wrong heading - should be "Values of binary logical expressions"

4.4 Short-Circuit Evaluations

What is Short-Circuit Evaluations?

- Evaluation of expressions containing && and || stops as soon as the outcome (true or false) is known.
- `expr1 && expr2`

  If `expr1` evaluates to false (zero), the evaluation of `expr2` will not occur.
- `expr1 || expr2`

  If `expr1` evaluates to true (non-zero), the evaluation of `expr2` will not occur.

```
int   i, j;

i=2 && (j=2);
printf ("%d %d\n", i, j);     /* 1 2 is printed */

(i=0) && (j=3);
printf ("%d %d\n", i, j);     /* 0 2 is printed */

i=0 || (j=4);
printf ("%d %d\n", i, j);     /* 1 4 is printed */

(i=2) || (j=5);
printf ("%d %d\n", i, j);     /* 2 4 is printed */
```

Note carefully the effect of parenthesis in the above expressions! Refer to the operator precedence table if needed.

Relational, Equality, and Logical Operators		
Relational operators	less than	<
	greater than	>
	less than or equal to	<=
	greater than or equal to	>=
Equality operators	equal to	==
	not equal to	!=
Logical operators	(unary) negation	!
	logical and	&&
	logical or	||

Operator precedence and associativity	
Operator	**Associativity**
() ++ (postfix) −− (postfix)	left to right
+ (unary) − (unary) ++ (prefix) −− (prefix) !	right to left
* / %	left to right
+ −	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += − = * = /= etc.	right to left
, (comma operator)	left to right

4.5 The Compound Statement

What is a Compound Statement?

- A series of variable declarations and statements surrounded by a pair of braces { } .
- The group of declarations and statements will then be regarded as a single logical unit by the compiler.

Example

```
int main(void)
{
     int x, y;

     scanf ("%d", &x);
     scanf ("%d", &y);

     {
          int temp;

          temp = x;
          x = y;
          y = temp;
     }
}
```

Another Example

- ```
  if (...)
        one_statement;
  ```

- ```
  if (...)  {
        one_statement;
        another_statement;
    }
  ```

4.6 Selection — The `if` Statement

Syntax and Semantics

```
if (expr)
     statement;

next_statement;
```

- If `expr` is true (nonzero), then `statement` is executed.
- Otherwise `statement` is skipped, and control passes to the `next_statement`.

Example One

```
score = 80;

if (score >= 50)
     printf ("Congratulations!\n");

printf ("Your score is %d.\n", score);
```

```
Congratulations!
Your score is 80.
```

Example Two

```
score = 40;

if (score >= 50)
     printf ("Congratulations!\n");

printf ("Your score is %d.\n", score);
```

```
Your score is 40.
```

if and Compound Statements

```
if (score >= 50) {
     grade = 'P';
     printf ("Congratulations!\n");
}

printf ("Your score is %d\n.", score);
```

4.7 Selection — The `if-else` Statement

Syntax and Semantics

```
if (expr)
     statement_1;
else
     statement_2;

next_statement;
```

- If `expr` is true (nonzero), then `statement_1` is executed and `statement_2` is skipped.
- Otherwise `statement_1` is skipped and `statement_2` is executed.
- In both cases, control then passes to `next_statement`.

Example

```
if (score >= 50) {
     grade = 'P';
     printf ("Congratulations!\n");
} else
     grade = 'F';

printf ("Your score is %d.\n", score);
printf ("Your grade is %c.\n", grade);
```

When `score` is 80

```
Congratulations!
Your score is 80.
Your grade is P.
```

When `score` is 40

```
Your score is 40.
Your grade is F.
```

4.7.1 The "Dangling `else`" Problem

Nested-`if` Statement

- An `if` statement can be used as the statement part of another `if` statement.

```
if (score >= 50)
     if (score >= 85)
          printf ("Congratulations!\n");
printf ("Your score is %d.\n", score);
```

Exercise

What will be printed by the above program fragment if `score` is

(a) `40`

(b) `70`

(c) `90` ?

The Dangling-`else` Problem

- An `if-else` statement can also be used as the statement part of another `if` statement.
- However, there are two *possible* interpretations:
- **Case I**

```
if (score >= 50)
     if (score >= 85)
          printf ("Congratulations!\n");
     else
          printf ("You got a pass.\n");
```

- **Case II**

```
if (score >= 50)
     if (score >= 85)
          printf ("Congratulations!\n");
else
     printf ("You got a pass.\n");
```

`if-else` Pairing Rule

- An `else` statement attaches to the nearest `if` that has not been paired with an `else`.
- So Case I has a better *indentation* for revealing the semantics of the program fragment.

Example

The program below finds and prints the minimum among the three values input by the user.

Program flow1_1.c

```
#include <stdio.h>

int main(void)
{
   int   x, y, z, min;

   printf("Input three integers:  ");
   scanf("%d%d%d", &x, &y, &z);

   if (x < y)
      min = x;
   else
      min = y;

   if (z < min)
      min = z;

   printf("The minimum value is %d.\n", min);

   return (0);
}
```

4.7.2 Deeply Nested `if-else` Statements

```
if (score >= 85)
     grade = 'A';
else
     if (score >= 80)
          grade = 'B';
     else
          if (score >= 70)
               grade = 'C';
          else
               if (score >= 60)
                    grade = 'D';
               else
                    if (score >= 50)
                         grade = 'E';
                    else
                         grade = 'F';
```

Alternative Style

```
if (score >= 85)
     grade = 'A';
else if (score >= 80)
     grade = 'B';
else if (score >= 70)
     grade = 'C';
else if (score >= 60)
     grade = 'D';
else if (score >= 50)
     grade = 'E';
else
     grade = 'F';
```

4.8 Selection — The `switch` Statement

Syntax and Semantics

```
switch (expression) {
     case constant_expr_1:
          statement_1;
          break;
     case constant_expr_2:
          statement_2;
          break;

     ....

     case constant_expr_n:
          statement_n;
          break;

     default:
          statement_d;
}

next_statement;
```

Execution Steps

- Evaluate `expression`, whose result must be a simple data type (such as `int`, `char`).
- Compare the result with each `constant_expr_?` in turn until a matching is found.
- Say `constant_expr_2` is matched.
  - Execute `statement_2`, `statement_3`, ..., `statement_n`, `statement_d`.
  - However, whenever a `break` is reached, jump to `next_statement` immediately.
- If no `constant_expr_?` matches,
  - If `default` present, execute `statement_d`.
  - Else execute `next_statement`.

Program flow1_2.c

```
#include <stdio.h>

int main(void)
{
     int     n;

     printf ("Enter a number: ");
     scanf ("%d", &n);

     switch (n) {
        case 1:
             printf("One\n");
             break;
        case 2:
             printf("Two\n");
             break;
        case 3:
             printf("Three\n");
             break;
        case 4:
             printf("Four\n");
             break;
        case 5:
             printf("Five\n");
             break;
        default:
             printf("Invalid number!\n");
     }

     return(0);
}
```

```
Enter a number: 2
Two
```

```
Enter a number: 5
Five
```

```
Enter a number: 10
Invalid number!
```

Program flow1_3.c

```
#include <stdio.h>

int main(void)
{
     int     n;

     printf ("Enter a number: ");
     scanf ("%d", &n);

     switch (n) {
        case 1:
             printf("One\n");
        case 2:
             printf("Two\n");
        case 3:
             printf("Three\n");
        case 4:
             printf("Four\n");
             break;
        case 5:
             printf("Five\n");
        default:
             printf("Invalid number!\n");
     }

     return(0);
}
```

```
Enter a number: 2
Two
Three
Four
```

```
Enter a number: 5
Five
Invalid number!
```

```
Enter a number: 10
Invalid number!
```

Program flow1_4.c

```
#include <stdio.h>
#define BEEP printf("\a\n")

int main(void)
{
     int     num;

     printf ("Enter a number between 1-5 ? ");
     scanf ("%d", &num);

     switch (num) {
        case (1):
             BEEP;
             break;
        case (2):
             BEEP; BEEP;
             break;
        case (3):
             BEEP; BEEP; BEEP;
             break;
        case (4):
             BEEP; BEEP; BEEP; BEEP;
             break;
        case (5):
             BEEP; BEEP; BEEP; BEEP; BEEP;
             break;
        default:
             printf ("1-5 please!\n");
     }

     return (0);
}
```

Exercise

Modify the above program so that only five `BEEP`s are used while the behavior of the program is maintained.

4.9 Selection — The Conditional Operator ?:

Syntax and Semantics

```
expr1 ? expr2 : expr3
```

- First, `expr1` is evaluated.
- If true, then `expr2` is evaluated and the value of `expr2` is taken as the value of the whole expression.
- Else, `expr3` is evaluated and the value of `expr3` is taken as the value of the whole expression.
- Similar to the `if-else` statement. But any difference?

- Example,

```
printf("min=%d", (x < y) ? (min = x) : (min = y));
```

- Example,

```
(y < z) ? printf("y is smaller.") : printf("z is smaller");
```

□ **End.**

Lecture 5

Flow of Control — II

Categories of Flow Controls

- *Relational, Equality, Logical Operators* are provided to facilitate flow controls.
- *Sequential* — Statements in a program are executed one after another.
- *Selection* — A choice of alternative actions (`if`, `if-else`, and `switch`).
- *Repetition* — Repeat certain actions (`while`, `for`, and `do`).

NOTE

The first three topics have already been covered in Lecture 4 (Flow of Control — I).

5.1 Repetition — The `while` Statement

Syntax and Semantics

```
while (expr)
     statement;

next_statement;
```

- As long as (i.e. `while`) `expr` is *true*, repeatedly execute `statement`.
- When `expr` evaluates to *false*, execute `next_statement`.

```
int i = 1, sum = 0;

while (i <= 4) {
     sum += i;
     i++;
}
printf ("%d\n", sum);
```

	Before		Condition	After	
	`i`	`sum`	`i <= 4`	`i`	`sum`
1st	1	0	True	2	1
2nd	2	1	True	3	3
3th	3	3	True	4	6
4th	4	6	True	5	10
5th	5	10	False	/	/

Program flow2_1.c

```
#include <stdio.h>

int main(void)
{
     int     cnt=0, n, max, x;

     printf ("How many numbers do you wish to enter? ");
     scanf ("%d", &n);
     printf ("\nEnter %d decimal numbers:\n", n);
     scanf ("%d", &x);
     max = x;
     cnt++;

     while (cnt < n) {
          scanf ("%d", &x);
          if (max < x)
               max = x;
          cnt++;
     }

     printf ("\nMaximum value: %d\n", max);
     return (0);
}
```

```
How many numbers do you wish to enter? 4

Enter 4 decimal numbers:
4 3 99 20

Maximum value: 99
```

Program flow2_2.c

```
#include <stdio.h>

int main(void)
{
     int     data, sum=0;

     scanf("%d", &data);

     while (data>=0) {
          sum += data;
          scanf("%d", &data);
     }

     printf ("The sum is %d\n", sum);

     return (0);
}
```

```
10
20
30
-1
The sum is 60
```

Infinite Loop Using `while`

```
while (1)
     statement;

next_statement;
```

- Stop the program in the operating system level, for example, `Ctrl-C` in DOS.
- There is a `break` statement for terminating the loop (discussed later).
- Use carefully!

5.2 Repetition — The `do-while` Statement

Syntax and Semantics

```
do
     statement;
while (expr);

next_statement;
```

- Similar to `while`.
- But `statement` executed at least once because `expr` is evaluated at bottom.

Program flow2_3.c

```
#include <stdio.h>

int main(void)
{
     int     cnt=0, max=0, n, x;

     printf ("How many numbers do you wish to enter? ");
     scanf ("%d", &n);
     printf ("\nEnter %d decimal numbers:\n", n);

     do {
          scanf ("%d", &x);
          if (max < x)
               max = x;
          cnt++;
     } while (cnt < n);

     printf ("\nMaximum value: %d\n", max);
     return (0);
}
```

5.3 Repetition — The `for` Statement

Syntax and Semantics

```
for (expr1; expr2; expr3)
    statement;

next_statement;
```

- `expr1` — initialization
  `expr2` — condition
  `expr3` — (increment)
- Execution Steps:
  1. `expr1` is evaluated.
  2. `expr2` is evaluated.
     - if *true*,
       (a) `statement` is executed.
       (b) `expr3` is executed.
       (c) goto step (2) again.
     - if *false*, `next_statement` is executed.

`while` Equivalent to the `for` Statement

```
expr1;

while (expr2) {
    statement;
    expr3;
}

next_statement;
```

Program flow2_4.c

```
#include <stdio.h>

int main(void)
{
     int     count, i;

     printf("Count? ");
     scanf("%d", &count);
     printf("\n");

     for (i=0; i<count; i++)
          printf("%d\n", count-i);

     printf("Go!\n");
     return (0);
}
```

```
Count? 5

5
4
3
2
1
Go!
```

Program flow2_5.c

```
#include <stdio.h>

int main(void)
{
     int     i, n, factorial=1;

     printf ("n ? ");
     scanf ("%d", &n);

     for (i=1; i <= n; i++)
       factorial *= i;

     printf ("The factorial of %d is %d\n.", n, factorial);

     return (0);
}
```

```
n ? 4
The factorial of 4 is 24.
```

The Empty Statement and `for`

```
for (expr1; expr2; expr3)
     statement;

next_statement;
```

- `expr1` and/or `expr2` and/or `expr3` can be missing. However, the `;` is still needed at the proper position.
- `;` is called the *empty statement*.
- Useful when a statement is needed *syntactically* but no action is required *semantically*.
- Example,

```
i=1;
sum=0;

for ( ; i<=10; i++)
     sum += i;
```

Infinite Loop Using `for`

```
for (;;) {
     statement;
}
```

Nested-`for`

A `for` loop can be nested inside another `for` loop.

Program flow2_6.c

```
#include <stdio.h>

int main()
{
     int i, j, row, column;

     printf("Row    = ? ");
     scanf("%d", &row);
     printf("Column = ? ");
     scanf("%d", &column);

     for (i=1; i<=row; i++) {
          for (j=1; j<=column; j++)
               printf("*");
          printf("\n");
     }

     return (0);
}
```

```
Row    = ? 3
Column = ? 5
*****
*****
*****
```

Program flow2_7.c

```
#include <stdio.h>

int main(void)
{
     int  i, j, size;

     printf("Size? ");
     scanf("%d", &size);

     for (i=1; i<=size; i++) {
          for (j=1; j<=(size-i); j++)
               printf(" ");
          for (j=1; j<=i; j++)
               printf("*");
          printf("\n");
     }

     return (0);
}
```

```
Size? 5
    *
   **
  ***
 ****
*****
```

Exercise

Study and execute the following program.

Program flow2_8.c

```
#include <stdio.h>
#define   N   7

int main(void)
{
     int   cnt = 0, i, j, k;

     for (i = 0; i <= N; ++i)
          for (j = 0; j <= N; ++j)
               for (k = 0; k <= N; ++k)
                    if (i + j + k == N) {
                         ++cnt;
                         printf("%3d%3d%3d\n", i, j, k);
                    }

     printf("\nCount: %d\n", cnt);
     return (0);
}
```

The Comma Operator and `for`

`expr1 , expr2`

- Lowest precedence of all operators.
- Left-to-right associativity.
- Value of `expr2` taken as value of the whole expression.
- Example, `a = 0 , b = 1;`
- Example,

```
for (i=1, factorial=1; i<=n; i++)
     factorial *= i;
```

5.4 Controlling Repetition

`break` and `while`

Causes an exit from the innermost enclosing loop.

Program flow2_9.c

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

int main(void)
{
     int     x;

     while (1) {
          scanf("%d", &x);
          if (x <= 0)
               break;
          printf ("square root = %1.2f\n", sqrt(x));
     }
     printf ("Bye!\n");
     return (0);
}
```

```
10
square root = 3.16
16
square root = 4.00
0
Bye!
```

`sqrt(x)` evaluates to $\sqrt{x}$. The use of `sqrt()` requires `#include <math.h>`.

break and do-while

Causes an exit from the innermost enclosing loop.

Program flow2_10.c

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

int main(void)
{
     int     x;

     do {
          scanf("%d", &x);
          if (x <= 0)
               break;
          printf ("square root = %1.2f\n", sqrt(x));
     } while (1);

     printf ("Bye!\n");
     return (0);
}
```

```
10
square root = 3.16
16
square root = 4.00
0
Bye!
```

`break` and `for`

Causes an exit from the innermost enclosing loop. Will `expr3` be executed before leaving the `for` loop?

Program flow2_11.c

```
#include <stdio.h>

int main(void)
{
     int     i, x;

     for (i=0; i<10; i++) {
          printf("i = %d\t", i);
          printf("x = ? ");
          scanf("%d", &x);
          if (x==0)
               break;
     }
     printf("After the loop, i = %d\n", i);
     printf ("Bye!\n");
     return (0);
}
```

```
i = 0   x = ? 10
i = 1   x = ? 20
i = 2   x = ? 4
i = 3   x = ? 5
i = 4   x = ? 0
After the loop, i = 4
Bye!
```

`continue`

Causes the current iteration of a loop to stop, and begins the next iteration.

Program flow2_12.c

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

int main(void)
{
     int     data, sum=0, k;

     for (k=0; k<MAX; k++) {
          scanf ("%d", &data);
          if (data <= 0)
               continue;
          sum += data;
     }
     printf ("Sum of positive values is %d\n.", sum);
     return (0);
}
```

```
10
20
-1
90
-5
Sum of positive values is 120.
```

`continue` as applied to `while` and `do-while`? Leave to you as exercises!

□ **End.**

Lecture 6

Fundamental Data Types

The Importance of Data Types

- Variable Declarations:
  - Reserve an appropriate amount of memory.
  - Ensure correct operations on variables.
- Expressions:
  - Combinations of constants, variables, and function calls.
  - Have *value* and *type*.

Fundamental data types: Long Form		
`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

- Abbreviations:
  - `signed int` ⇔ `int`
  - `short int` ⇔ `short`
  - `long int` ⇔ `long`
  - `unsigned int` ⇔ `unsigned`

Fundamental data types: Abbreviated		
`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

Fundamental data types grouped by functionality			
Integral types	`char`	`signed char`	`unsigned char`
	`short`	`int`	`long`
	`unsigned short`	`unsigned`	`unsigned long`
Floating types	`float`	`double`	`long double`
Arithmetic types	*Integral types* + *Floating types*		

6.1 The Data Type `char`

What is a `char`?

- One of the fundamental data types in C.
- Representing a single character.
- Enclosed inside `' '`.
- Examples are `'A'`, `'d'`, `'9'`, `'+'`, `'='`, ...

Internal Representation of a `char`

- A `char` can be thought of as an integer value **(not the `int` data type)**.
- For example,
  - `'A'` is stored as integer value 65.
  - `'+'` is stored as integer value 43.

What integer value represents what character?

The ASCII table (shown on the next page) defines the standard mappings between characters to their corresponding integer values.

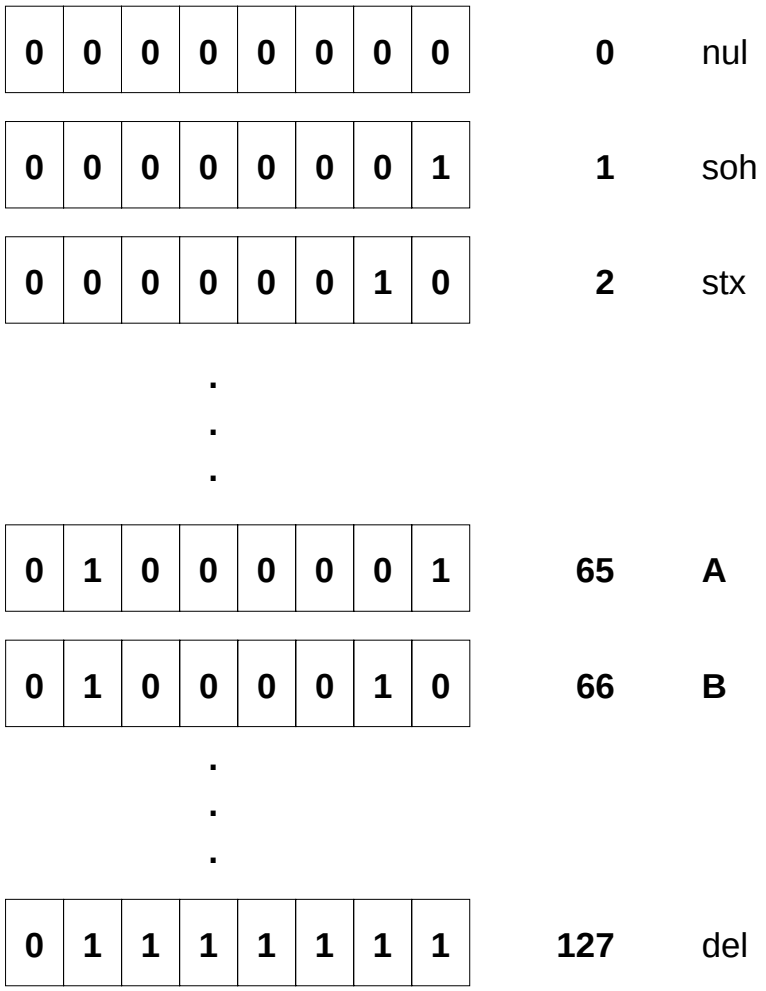
The 7-bit ASCII Table

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	NL	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	$	%	&	’
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	∧	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	∼	DEL		

- ASCII stands for “*American Standard Code for Information Interchange.*”
- Some are printable characters: ’A’, ’=’, ’:’, ….
- Some are non-printable characters: newline (NL), bell (BEL), tab (HT, VT), ….

How does a byte store a `char`?

- Each character is stored in one byte.
- One byte $\longrightarrow$ 8 bits $\longrightarrow$ 2^8 possibilities
- 256 distinct values can be represented.
- The range is 0 – 255.



Some character constants and their integer ASCII values					
Lowercase ASCII value	'a'	'b'	'c'	···	'z'
	97	98	99	···	112
Uppercase ASCII value	'A'	'B'	'C'	···	'Z'
	65	66	67	···	90
Digit ASCII value	'0'	'1'	'2'	···	'9'
	48	49	50	···	57
Other ASCII value	'&'	'*'	'+'	···	
	38	42	43	···	

Nonprinting or hard-to-print characters		
Name of character	Written in C	Integer value
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
formfeed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11

- Escape character (\)
- Escape sequence (e.g. \n)

Some Examples

`printf ("%c", 'a');`	`a`
`printf ("%d", 'a');`	`97`
`printf ("%c", 97);`	`a`
`printf ("H\tello");`	`H  ello`
`printf ("\007");`	*bell!*
`printf ("\"Hello!\"");`	`"Hello!"`

6.2 The Data Type `int`

The `int` Data Type

- ..., -3, -2, -1, 0, +1, +2, +3, ...
- Computers can store only a finite portion of natural numbers. That is, upper and lower limits exist.
- Why?
- An `int` is stored in 2 bytes, 4 bytes, or even larger.

2-byte `int`

- 2 bytes $\Leftrightarrow$ 16 bits
- 2^{16} distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

  $-2^{15},\ -2^{15}+1,\ \ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots,\ 2^{15}-1$

- That is,

  $-32768,\ -32767,\ \ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots,\ 32767$

4-byte `int`

- 4 bytes $\Leftrightarrow$ 32 bits
- 2^{32} distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

  $$-2^{31},\ -2^{31}+1,\ \ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots,\ 2^{31}-1$$

- That is,

  $$-2147483648,\ \ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ \ldots,\ 2147483647$$

- Approximately,

  $$(\approx -2\ billion),\ \ldots,\ (\approx 2\ billion)$$

Exercise

Find out the range of values that can be stored if a certain computer uses 8 bytes to store one single integer value.

Finding out the size of an integer

`sizeof(int)`

- Can be used to find out the number of bytes used for storing an `int`.
- Example below.

Program data_1.c

```
#include <stdio.h>

int main(void)
{
     printf("size of int   = %d\n", sizeof(int));
     return (0);
}
```

```
size of int   = 4
```

Program data_2.c

```
#include <stdio.h>
#define BIG 2000000000

int main(void)
{
     int    a, b=BIG, c=BIG;

     printf("Size of int = %d bits\n", sizeof(int)*8);

     a = b + c;
     printf ("a=%d, b=%d, c=%d\n", a, b, c);
     return(0);
}
```

```
Size of int = 32 bits
a=-294967296, b=2000000000, c=2000000000
```

The Overflow Problem

In the above program,

- $b + c > 2^{31} - 1 \approx 4$ billions
- An 32-bit integer variable cannot hold this large value.
- Variable a *overflows*.

How to solve the overflow problem?

`int` variation — `short int`

- **May** take less number of bytes than `int`.
- **May** thus store a smaller integer value.
- For example,
  - Suppose your computer uses 4-byte `int`, ranging from -2147483648 to +2147483647.
  - A `short int` may occupy 2 bytes, ranging from -32768 to +32767.
  - Why?

`int` variation — `long int`

- **May** take more number of bytes than `int`;
- **May** thus store a larger integer value.
- For example,
  - Suppose your computer uses 4-byte `int`, ranging from -2147483648 to +2147483647.
  - A `long int` may occupy 8 bytes, ranging from (-2^{63}) to $(+2^{63} - 1)$.
  - Why?

`int` variation — `unsigned int`

- Take same number of bytes as `int`.
- But stores non-negative integers only.
- Range: $0 \sim 2^{int\_length} - 1$
- 2-byte `unsigned int`:
  - 0 to $2^{16} - 1$
  - 0 to 65535.
- 4-byte `unsigned int`:
  - 0 to $2^{32} - 1$
  - 0 to 4294967295.

Note

Note that combinations of `short` and `long` with `unsigned` are possible, leading to:

- `unsigned short int`
- `unsigned long int`

Exercise

Find out the range of values that can be represented for

- 2-byte `unsigned short int`
- 8-byte `unsigned long int`

Program data_3.c

```
#include <stdio.h>
#define BIG 2000000000

int main(void)
{
     unsigned int     a, b=BIG, c=BIG;

     printf("Size of int = %d bits\n", sizeof(int)*8);

     a = b + c;
     printf ("a=%u, b=%d, c=%d\n", a, b, c);
     return(0);
}
```

```
Size of int = 32 bits
a=4000000000, b=2000000000, c=2000000000
```

Using `unsigned int`

- Note the use of `unsigned` in declaring variable `a`.
- Note the use of `%u` in the `printf`.
- Overflow is avoided. Why?

6.3 The Floating Types

What is a Floating-Point value?

- Store real values such as 0.1, -2.4, 3.14159.
- Exponential notation:

$$\begin{aligned} 1.234567e5 &= 1.234567 \times 10^5 \\ &= 123456.7 \end{aligned}$$

$$\begin{aligned} 1.234567e\text{-}3 &= 1.234567 \times 10^{-3} \\ &= 0.001234567 \end{aligned}$$

Precision and Range

- Precision — number of significant decimal places.
- Range — largest and smallest values that can be stored.
- Example, `float` on *a particular machine*:

  Precision: 6 significant figures

  Range: 10^{-38} to 10^{+38}

  $$0.d_1d_2d_3d_4d_5d_6 \times 10^n, \qquad -38 \leq n \leq +38$$

`float` variation — `double`

- Example, `double` on a *particular machine*:

  Precision: 15 significant figures

  Range: 10^{-308} to 10^{+308}

  $$0.d_1d_2\ldots d_{15} \times 10^n, \qquad -308 \leq n \leq +308$$

- Example, assuming precision as 15 significant figures

  $$x = 123.45123451234512345;$$

  results in

  $$0.123451234512345 \times 10^3$$

Important

- Not all real numbers are representable.
- Floating arithmetic operations need not be exact.
- For very large computations, rounding errors may accumulate and become significant.

6.4 The `sizeof` Operator

The `sizeof` Operator

- Used to find the number of bytes needed to store a piece of data.
- General form,

```
sizeof(data_type_or_name)
```

- `data_type_or_name` can be a valid data type (such as `int`, `float`) or a variable name that has already been declared.

Program data_4.c

```
#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("Here are the sizes of some fundamental types:\n\n");
   printf("       char:%3d byte \n", sizeof(char));
   printf("      short:%3d bytes\n", sizeof(short));
   printf("        int:%3d bytes\n", sizeof(int));
   printf("       long:%3d bytes\n", sizeof(long));
   printf("   unsigned:%3d bytes\n", sizeof(unsigned));
   printf("      float:%3d bytes\n", sizeof(float));
   printf("     double:%3d bytes\n", sizeof(double));
   printf("long double:%3d bytes\n", sizeof(long double));
   printf("\n");

   return (0);
}
```

```
Here are the sizes of some fundamental types:

       char:  1 byte
      short:  2 bytes
        int:  4 bytes
       long:  4 bytes
   unsigned:  4 bytes
      float:  4 bytes
     double:  8 bytes
long double: 16 bytes
```

Execution results vary with different compilers.

6.5 Data Type Conversions

Arithmetic Conversions?

- An arithmetic expression has both a value and a type.
- For example, if both `x` and `y` have type `int`, the expression `x + y` also has type `int`.
- When an expression contains operands of different types, unification on data type is needed.
- This is achieved *by the compiler* using *arithmetic conversions*.
- Sometimes called *coercion*.

Declarations

```
char c;       short s;            int i;
unsigned u;   unsigned long ul;   float f;
double d;     long double ld;     long l;
```

Expression	Type	Expression	Type
`c - s / i`	`int`	`u * 7 - i`	`unsigned`
`u * 2.0 - i`	`double`	`f * 7 - i`	`float`
`c + 3`	`int`	`7 * s * ul`	`unsigned long`
`c + 5.0`	`double`	`ld + c`	`long double`
`d + s`	`double`	`u - ul`	`unsigned long`
`2 * i / l`	`long`	`u - l`	*system-dependent*

Casting

- Explicit type conversions *by the programmer.*
- Example,

```
int     i=9;
float   f;

f = (double) i;
```

  What is the data type of variable `i` after the casting operation?

- Examples,

```
(long) ('A' + 1.0)

x = (float) ((int) y + 1)

(double) (x = 77)
```

- Examples,

```
printf("%f\n", 14/3);          /* incorrect */
printf("%f\n", 14.0/3);        /* 4.666667 */
printf("%f\n", (float) 14/3); /* 4.666667 */
```

Suffixes

A suffix can be appended to a numerical constant to specify its type explicitly.

Suffix	Type	Example
`u` or `U`	`unsigned`	37U
`l` or `L`	`long`	37L
`ul` or `UL`	`unsigned long`	37UL

Suffix	Type	Example
`f` or `F`	`float`	3.7F
`l` or `L`	`long double`	3.7L

□ **End.**

Lecture 7

Functions

What are functions?

- A simple C function is a **named** block of statements.
- C functions are used to solve small problems.
- Small functions are combined into other larger functions and ultimately used in `main()`.

7.1 Function Invocation

Program func_1.c

```
#include <stdio.h>

prn_message()
{
     printf("\tHave a nice day!\n");
}

int main(void)
{
     printf("In the main!\n");

     prn_message();

     printf("Back to main!\n");
     return (0);
}
```

```
In the main!
        Have a nice day!
Back to main!
```

To invoke a function

- Write the function name followed by parenthesis.
- Program control passes to the function.
- When the function reaches the closing brace ('}'), control is passed back to where it was called.

Program func_2.c

```
#include <stdio.h>

prn_message()
{
     printf("\tHave a nice day!\n");
}

int main(void)
{
     printf("There is a message for you,\n");
     prn_message();
     prn_message();
     prn_message();

     return (0);
}
```

```
There is a message for you,
     Have a nice day!
     Have a nice day!
     Have a nice day!
```

NOTE

The same function can be called repeatedly, just like nothing has happened before.

7.2 Functions and Local Variables

Program func_3.c

```
#include <stdio.h>

func()
{
     int   two;

     two=2;
     printf("two is %d\n", two);
}

int main(void)
{
     int   one;

     one = 1;
     func();

     return (0);
}
```

NOTE

- A function can have its own variables declared.
- Such variables are called **local variables**.

Program func_4.c

```
#include <stdio.h>

func()
{
     int   two=2;

     printf("%d\n", two);
}

int main(void)
{
     func();
     printf("%d\n", two);     /* compilation error */
     return (0);
}
```

NOTE

- Local variables are only visible within the function in which they are declared. Thus, `two` is visible only inside `func()`.
- When the program is compiled, an *error message* similar to the following appears:

  ```
  test.c:  In function ‘main’:
  test.c:13:  ‘two’ undeclared (first use in
  this function)
  ```

- The *scope* of a variable is the portion of the program that the variable is visible and thus accessible.
- What is the scope of variable `two`?

Program func_5.c

```
#include <stdio.h>

func()
{
     int local, x, y;

     local = 2;
     y = 3;
     x = local + y;

     printf("\t x = %d\n", x);
     printf("\t local in func = %d\n", local);
}

int main(void)
{
     int local;

     local = 1;
     printf("local in main = %d\n", local);

     func();
     printf("local in main = %d\n", local);
}
```

```
local in main = 1
      x = 5
      local in func = 2
local in main = 1
```

7.3 Function and Parameters Passing

Function Parameters

The function caller can provide one or more data values to the function.

7.3.1 Single Parameter Passing

Program func_6.c

```
#include <stdio.h>

prn_message(int k)
{
     int   i;

     printf("Here is the message:\n");
     for (i = 0; i < k; i++)
          printf("   Have a nice day!\n");
}

int main(void)
{
     int   n;

     printf("There is a message for you.\n");
     printf("How many times do you want to see it? ");
     scanf("%d", &n);
     prn_message(n);
     return (0);
}
```

```
There is a message for you.
How many times do you want to see it? 3
Here is the message:
   Have a nice day!
   Have a nice day!
   Have a nice day!
```

Explanation

- Function `prn_message()` needs a single integer for performing its work.
- In the calling environment (i.e. `main()`), `n` is used as an **actual parameter**.
- In function `prn_message()`, **formal parameter** `k` receives the value of the actual parameter from the calling environment.

7.3.2 Multiple Parameters Passing

Program func_7.c

```
#include <stdio.h>

print_min(int x, int y)
{
     if (x < y)
          printf ("min = %d\n", x);
     else
          printf ("min = %d\n", y);
}

int main(void)
{
     int  a, b;

     printf("Input a? ");
     scanf("%d", &a);

     printf("Input b? ");
     scanf("%d", &b);

     print_min(a, b);
     return (0);
}
```

```
Input a? 80
Input b? 35
min = 35
```

Explanation

- Function `print_min()` takes two integer parameters.
- In the calling environment (i.e. `main()`), variables `a` and `b` are used as parameters.
- In the function `print_min()`, parameters `x` and `y` receive the values of the parameters from the calling environment.
- The type and number of parameters in the calling environment and in the function must be matched.

7.4 The return Statement

Program func_8.c

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

func()
{
     float     num;

     do {
          printf("Number = ? ");
          scanf("%f", &num);
          if (num <= 0)
               return;
          else
               printf("SQRT(%.2f) = %.2f\n", num, sqrt(num));
     } while (1);
}

int main(void)
{
     func();
     printf("\nThanks for using this program!\n");
     return (0);
}
```

```
c:\> run
Number = ? 10
SQRT(10.00) = 3.16
Number = ? 81
SQRT(81.00) = 9.00
Number = ? 0

Thanks for using this program!
c:\>
```

NOTE

- When the `return` statement is reached, a function terminates immediately.
- A function may have more than one `return` statement.
- If there is no `return`, the function terminates when the execution reaches its closing brace ('}').

7.5 Function Type

Program func_9.c

```
#include <stdio.h>

void prn_message()
{
     printf("\tHave a nice day!\n");
}

int main(void)
{
     int n;

     printf("There is a message for you,\n");
     prn_message();
     return (0);
}
```

What is function type?

- Each function has a type, corresponding to the data value that it returns, if any.
- If a function returns no value, its type should be `void`.
- We declare a function's type before its name. For example,

```
int get_input()
{
     ...
}
```

7.6 Function with Return Value

Program func_10.c

```
#include <stdio.h>

int min(int a, int b)
{
     if (a < b)
          return a;
     else
          return b;
}

int main(void)
{
     int   j, k, m;

     printf("Input two integers:  ");
     scanf("%d%d", &j, &k);
     m = min(j, k);
     printf("\nThe minimum is %d.\n",  m);
     return (0);
}
```

```
Input two integers:  99 3

The minimum is 3.
```

Explanation

- Function `min()` returns an integer.
- To return an integer value, the `return` statement is used.

```
return expression;
```

  where `expression` is the data value to be returned.

- Since `min()` returns an integer, its type is `int`.
- The *return value* of the function can be used by the calling environment (`main()` in the example).
- Note that the following two formats are equivalent:

```
return expression;

return (expression);
```

Summary on the `return` Statement

- When a `return` is encountered, program control is passed back to where it was called.
- The first form:

```
return;
```

  - Used when the function returns no value.
  - The type of the function must therefore be `void`.

- The second form:

```
return expression;
```

  - Used when the function returns a value.
  - The value of the `expression` is returned.
  - The type of the `expression` must be the same as the type of the function.
  - The type of the function must therefore not be `void`.
  - Conversely, when the type of a function is not `void`, there must be at least one `return expression` statement with the correct `expression` data type.

- When there is no `return` statement in a function, control is passed back when the closing brace (`'}'`) of the function is encountered.
- What type should this function be?

```
void fcn()
{
     ...
}
```

```
void fcn()
{
     ...
     return;
}
```

```
void fcn()
{
     ...
     return x;
}
```

```
int fcn()
{
     ...
}
```

```
int fcn()
{
     ...
     return;
}
```

```
int fcn()
{
     int i;
     ...
     return i;
}
```

```
int fcn()
{
     float f;
     ...
     return f;
}
```

```
int fcn()
{
     int   i;
     float f;
     ...
     if (...)
         return i;
     else
         return f;
}
```

Summary on Function Definition

```
type function_name (parameter_type_list)
{
     variable declarations;

     statements;
}
```

- The `type` of the function is the type of the value returned by the function.
- `function_name` is an identifier and is used when the function is to be called.
- The `parameter_type_list` describes the number and types of the formal parameters passed to the function when it's called.
- The formal parameters can be used inside the function just like variables.
- Both `type` and `parameter_type_list` can be `void`.

7.7 Function Prototype

```
#include <stdio.h>

int sum(int a, int b)
{
     ......
     avg = average(x, y);
     ......
}

float average(int a, int b)
{
     ......
}

void prn_value()
{
     ......
}

int main(void)
{
     ......
}
```

```
#include <stdio.h>

float average(int a, int b)
{
     ......
}

int sum(int a, int b)
{
     ......
     avg = average(x, y);
     ......
}

void prn_value()
{
     ......
}

int main(void)
{
     ......
}
```

NOTE

- Any problem with the order shown on the left?
- Solution is shown on the right.
- But consider the following situation.

```
float average(int a, int b)
{
     ......
     total = sum(x, y);
     ......
}

int sum(int a, int b)
{
     ......
     avg = average(x, y);
     ......
}
```

Solution — Function Prototypes

```
int sum(int a, int b);
```

- Tells the compiler:
  - The number and type of parameters that are to be passed to the function.
  - The type of the value that is to be returned by the function.

  before the function is actually defined.
- Parameter names (identifiers) can be omitted.

```
int sum(int , int);
```

A Program with Proper Function Prototypes

```
#include <stdio.h>

int sum(int, int);
float average(int, int);
void prn_value();

int main(void)
{
     ......
}

int sum(int a, int b)
{
     ......
}

float average(int a, int b)
{
     ......
}

void prn_value()
{
     ......
}
```

7.8 Use of Random Numbers

Program func_11.c

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

int main(void)
{
     int  i;

     for (i=0; i<10; i++)
          printf("%d\n", rand());

     return 0;
}
```

```
c:\1110> 7_12.exe
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

```
c:\1110> 7_12.exe
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

Program func_12.c

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

int main(void)
{
     int  i, seed;

     printf("Seed = ? ");
     scanf("%d", &seed);

     srand(seed);

     for (i=0; i<5; i++)
          printf("%d\n", rand());

     return 0;
}
```

```
C:\1110> 7_13.exe
Seed = ? 5
18655
8457
10616
31877
10193
```

```
C:\1110> 7_13.exe
Seed = ? 10
4543
28214
11245
8870
16887
```

Program func_13.c

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

int main (void)
{
     int     seed, rank, suit;

     seed=time(NULL);
     srand(seed);

     rank = rand() % 4;
     switch (rank) {
          case 0: printf("Spade ");
                  break;
          case 1: printf("Heart ");
                  break;
          case 2: printf("Club ");
                  break;
          case 3: printf("Diamond ");
                  break;
     }

     suit = rand() % 13 + 1;
     if (suit <= 10)
          printf ("%d\n", suit);
     else
          switch (suit) {
               case 11: printf("J\n");
                        break;
               case 12: printf("Q\n");
                        break;
               case 13: printf("K\n");
                        break;
          }

     return 0;
}
```

Program func_14.c

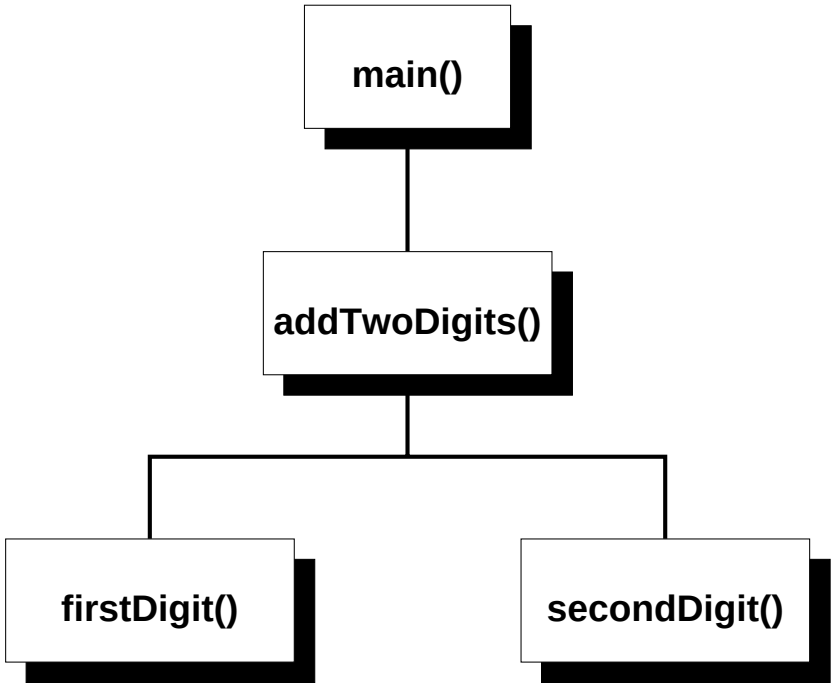
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void prn_random (int);

int main (void)
{
     int     n;

     printf("How many random fractions? ");
     scanf("%d", &n);
     prn_random(n);
     return(0);
}

void prn_random(int k)
{
     int     i;

     srand(time(NULL));
     for (i=1; i<=k; i++)
          printf("%.2f\n", (float) rand()/RAND_MAX);
}
```

```
How many random fractions? 4
0.51
0.18
0.31
0.53
```

7.9 Functions and Structured Programming

Top-Down Design

In top-down design, a program is divided into a main module and its related modules. Each module is in turn divided into submodules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.



Program func_15.c

```
#include <stdio.h>

int addTwoDigits(int);
int firstDigit(int);
int secondDigit(int);

int main(void)
{
     int  number;
     int  sum;

     printf("Enter an integer: ");
     scanf("%d", &number);

     sum = addTwoDigits(number);
     printf("Sum of last two digits is: %d\n", sum);

     return 0;
}

int addTwoDigits(int number)
{
     int  result;

     result = firstDigit(number) + secondDigit(number);
     return result;
}

int firstDigit(int num)
{
     return (num % 10);
}
```

```
int secondDigit(int num)
{
     int  result;

     result = (num / 10) % 10;
     return result;
}
```

□ **End.**

Lecture 8

More on Variables — Scope and Storage

8.1 Variable Scope Rules

What are Scope Rules?

- The **scope** of an identifier is defined to be the part of program in which the identifier is known or accessible.
- **Scope rules** depend on the notion of **blocks**.
- In C, a block is a compound statement, which can contain declarations of its own.
- *The basic scope rule of C is that identifiers are accessible within the block in which they are declared.*

Program more_1.c

```
#include <stdio.h>

int main(void)
{
     int  a=1;                          /* outer block */

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

     {                                  /* inner block starts */
          int  b=2;

          printf ("\t %d\n", b);
     }                                  /* inner block ends */

     printf ("%d\n", a);                /* outer block */

     return 0;
}
```

```
1
        2
1
```

NOTE

- Variable a is declared in the outer block (main).
- Variable b is declared in the inner block.

Program more_2.c

```
#include <stdio.h>

int main(void)
{
     int  a=1;

     printf ("%d\n", a);              /* valid */

     {
          int  b=2;

          printf ("\t %d\n", a);      /* valid */
          printf ("\t %d\n", b);      /* valid */
     }

     printf ("%d\n", a);              /* valid */
     printf ("%d\n", b);              /* invalid */

     return 0;
}
```

NOTE

- Since the outer block encloses the inner block and variable a is declared in the outer block, the scope of variable a also covers the inner block.
- Variable b is declared in the inner loop. Its scope is equivalent to the inner block only. It is therefore not accessible outside the inner block.

Program more_3.c

```
#include <stdio.h>

int main(void)
{
     int  a=1;                          /* outer block */

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

     {                                  /* inner block starts */
          int  a=2, b=3;

          a++;
          printf ("\t %d\n", a);
          printf ("\t %d\n", b);
     }                                  /* inner block ends */

     printf ("%d\n", a);
     return 0;
}
```

Masking

- What will be printed?
- Variable `a` is declared in the outer block (`main`).
- **Another** variable with the same name `a` is declared in the inner block.
- The outer block `a` is *hidden* from the inner block and cannot be accessed from within the inner block.

Program more_4.c

```
#include <stdio.h>
int  universe;
void fcn();

int main(void)
{
     int  m;

     universe = 10;      /* valid */
     m = 33;             /* valid */
     f = 99;             /* invalid */
     return 0;
}

void fcn()
{
     int  f;

     universe = 20;      /* valid */
     m = 33;             /* invalid */
     f = 99;             /* valid */
}
```

Global *versus* Local Variables

- The declaration of `universe` appears outside any function. Its scope covers the whole program file. We say that `universe` is a **global** variable.
- The scope of `m` covers only the `main` function where it is declared. We say that `m` is a **local** variable inside `main()`. Similarly, `f` is a local variable inside `fcn()`.

8.2 Storage Class

Storage Class

- Every variable in C has three attributes:
  - Name
  - Type
  - Storage Class
- Four storage classes

```
auto    register    static    extern
```

- In the following format,

```
[Storage Class] type name;
```

- Example,

```
void fcn()
{
    register int x;
}
```

8.2.1 The Storage Class `auto`

```
int main(void)
{
     int    a, b, c;
     float  f;

     ...
}
```

```
int main(void)
{
     auto int   a, b, c;
     auto float f;

     ...
}
```

`auto`

- Variables declared within function bodies are by default `auto`matic.
- Thus, automatic is the most common of the four storage class.
- The keyword `auto` is seldom used.

Creation / Destruction of `auto` variables

- When the block is entered, memory is allocated for the automatic variables (Creation).
- When the block is exited, the memory set aside for the automatic variables are released (Destruction). Thus the values of these variables are lost.
- If the block is re-entered, the whole process repeats. But the values of the variables are **unknown**. Why?

8.2.2 The Storage Class `register`

```
int main()
{
     register int i;

     for (i=0; i<1000; i++) {
          ...
     }
     ...
}
```

`register`

- Advices the computer that the associated variables should be stored in CPU registers if possible.
- If not, the storage class defaults to automatic and the variable is stored inside the main memory.
- Attempts to improve program execution speed because accessing a register is much faster than accessing a memory location.

8.2.3 The Storage Class `static`

Program more_5.c

```
#include <stdio.h>

void fcn(void);

int main(void)
{
     int  i;

     for (i=0; i<5; i++)
          fcn();
     return 0;
}

void fcn(void)
{
     static int     static_var = 0;
     int            auto_var = 0;

     printf("%d \t %d\n", auto_var, static_var);
     auto_var++;
     static_var++;
}
```

```
0       0
0       1
0       2
0       3
0       4
```

`static`

- Allow a local variable to retain its previous value when the function is reentered.

8.2.4 The Storage Class `extern`

`extern`

- In the development of large scale software, it is not uncommon to spread the source code across several files.
- External variables can be shared among functions in different files.
- This can be achieved using the `extern` keyword prepended to a variable declaration.

```
extern type var_name;
```

  - Inform the compiler to look for the declaration of `var_name` elsewhere, either later within the same file or in another file.
  - Supply the type information of `var_name`.
  - **No new variable is created.**

File: `main.c`

```
#include <stdio.h>

extern int sum;
extern void summation(void);

void main(void)
{
     summation();
     printf("Sum = %d\n", sum);

     return 0;
}
```

File: `sum.c`

```
#include <stdio.h>

int sum;

void summation(void)
{
     int a, b;

     scanf("%d %d", &a, &b);
     sum = a + b;
}
```

□ **End.**

Lecture 9

Array Processing

What is an array?

An array is a fixed-size, sequenced collection of elements of the same data type.

- One single name.
- Indexible.
- Stored contiguously .

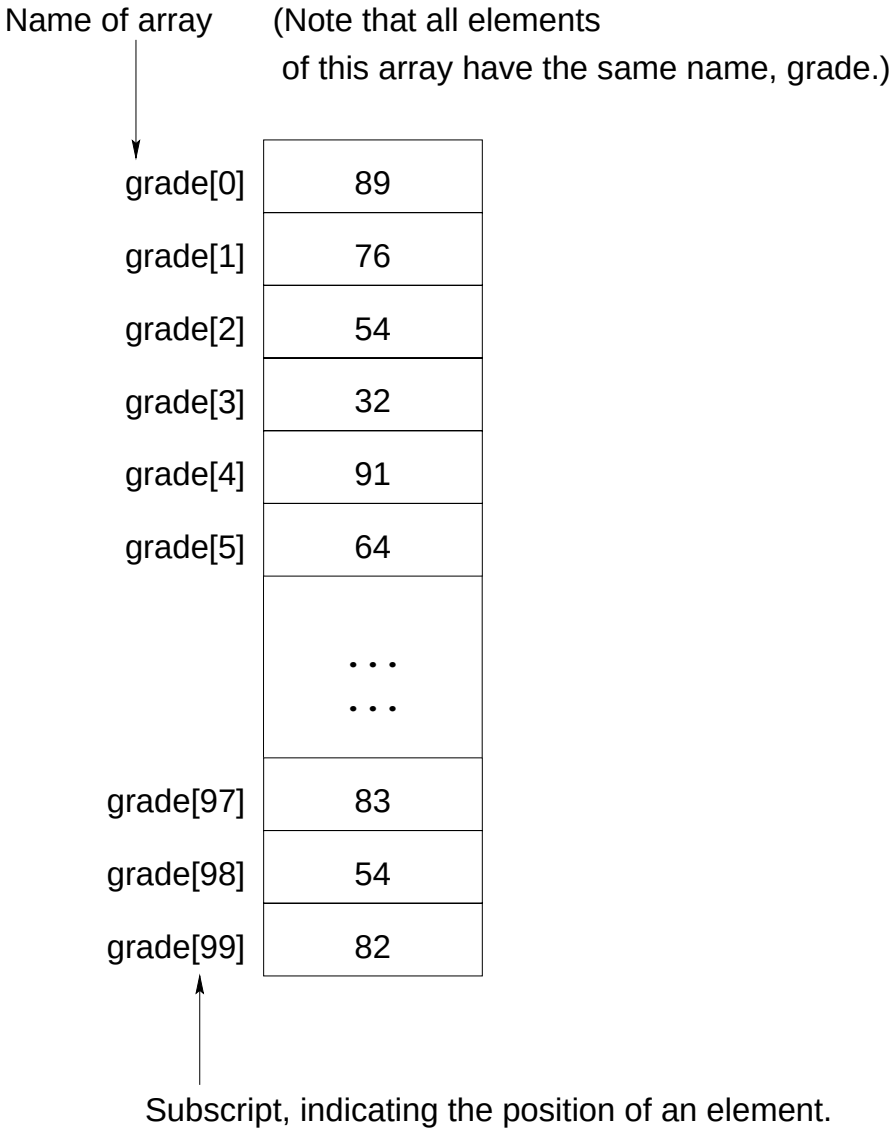
Program array_1.c

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

int main(void)
{
     int grade1, grade2, grade3;

     printf("Student 1? ");
     scanf("%d", &grade1);
     printf("Student 2? ");
     scanf("%d", &grade2);
     printf("Student 3? ");
     scanf("%d", &grade3);

     printf("Average = %.2f\n", (grade1+grade2+grade3)/3.0);
     return 0;
}
```

NOTE

- The program works if there are only three students.
- What if the class size is 100?
- Declare 100 variables?

9.1 One-Dimensional Array



One-Dimensional Array

- The array `grade` can hold 100 distinct integer values simultaneously.
- The array elements are referred to as `grade[0]`, `grade[1]`, `grade[2]`, ..., `grade[99]`.
- The integral value inside `[ ]` is called a *subscript* or *index*.
- Note carefully array subscript starts with 0 in C.
- All the elements are of the same type (`int`).
- Questions:
  - How can an array be declared?
  - How can the elements in an array be processed?

Program array_2.c

```
#include <stdio.h>

int main(void)
{
     int  list[4];
     int  i, sum=0;

     for (i=0; i<4; i++) {
          printf("number = ? ");
          scanf("%d", &list[i]);
     }

     for (i=0; i<4; i++)
          sum += list[i];

     printf("\nsum = %d\n", sum);

     return 0;
}
```

```
number = ? 1
number = ? 2
number = ? 3
number = ? 4

Sum = 10
```

9.1.1 Array Bounds

Program array_3.c

```
#include <stdio.h>

int main(void)
{
     int  list[4];
     int  i, sum=0;

     for (i=0; i<4; i++) {
          printf("number = ? ");
          scanf("%d", &list[i]);
     }

     for (i=0; i<=4; i++)
          sum += list[i];

     printf("\nsum = %d\n", sum);

     return 0;
}
```

```
number = ? 1
number = ? 2
number = ? 3
number = ? 4

sum = 42
```

Question

What is the difference between the previous two programs?

Array Bounds

- The value of an array subscript must be in the range 0 to 3 in the above example.
- An array subscript value outside this range will cause a run-time error — **array out of bound**.
- The effect of the error is unpredictable.
- It is the programmer's responsibility (not the compiler's) to ensure correct value of array subscripts.

9.2 Two-Dimensional Array

	col 0	col 1	col 2	col 3	col 4
row 0	`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[0][3]`	`a[0][4]`
row 1	`a[1][0]`	`a[1][1]`	`a[1][2]`	`a[1][3]`	`a[1][4]`
row 2	`a[2][0]`	`a[2][1]`	`a[2][2]`	`a[2][3]`	`a[2][4]`

Two-Dimensional Arrays

```
int a[3][5];
```

- The first dimension denotes the row-index, which starts with 0.
- The second dimension denotes the column-index, which also starts with 0.
- In processing array elements of a two-dimensional array, each dimension requires a single `for` loop.
- Therefore, a two-level nested-`for` loop is necessary.

Program array_4.c

```
#include <stdio.h>

int main(void)
{
   int   a[3][5], i, j, sum=0;

   /* fill the array */
   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         a[i][j] = i + j;

   /* print and sum array values */
   for (i = 0; i < 3; ++i) {
      for (j = 0; j < 5; ++j) {
         printf("a[%d][%d] = %d   ", i, j, a[i][j]);
         sum += a[i][j];
      }
      printf("\n");
   }

   printf("\nsum = %d\n\n", sum);

   return 0;
}
```

```
a[0][0] = 0   a[0][1] = 1   a[0][2] = 2   a[0][3] = 3   a[0][4] = 4
a[1][0] = 1   a[1][1] = 2   a[1][2] = 3   a[1][3] = 4   a[1][4] = 5
a[2][0] = 2   a[2][1] = 3   a[2][2] = 4   a[2][3] = 5   a[2][4] = 6

sum = 45
```

9.3 Multi-Dimensional Array

Array dimensions	
Declarations of arrays	**Remarks**
`int a[100];`	a one-dimensional array
`int b[2][7];`	a two-dimensional array
`int c[5][3][2];`	a three-dimensional array

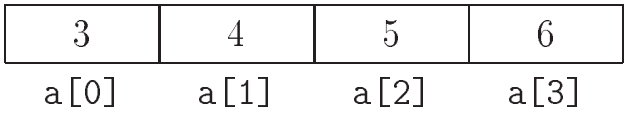
```
#include <stdio.h>

int main()
{
        int     a[7][9][2];
        int     i, j, k, sum=0;

        ...
        ...
        for (i=0; i<7; i++)
                for (j=0; j<9; j++)
                        for (k=0; k<2; k++)
                                sum += a[i][j][k];
        ...
        ...
}
```

9.4 Array Initializations

9.4.1 One Dimensional

Example One

```
int a[4] = { 3, 4, 5, 6 };
```



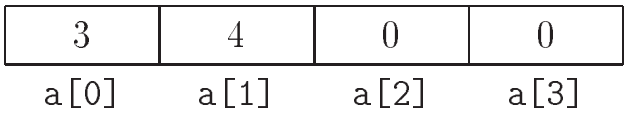
Example Two

```
int a[] = { 3, 4, 5, 6 };
```

- If an array is declared without a size, it is implicitly given the size of the number of *initializers*.

Example Three

```
int a[4] = { 3, 4 };
```



- When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero.

9.4.2 Two Dimensional

Example One

```
int a[2][3] = { 1, 2, 3, 4, 5, 6 };
```

	col 0	col 1	col 2
row 0	1	2	3
row 1	4	5	6

Example Two

```
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Example Three

```
int a[][3] = { {1, 2, 3}, {4, 5, 6} };
```

- If the first bracket pair is empty, then the compiler takes the size from the number of inner brace pairs.

9.4.3 Multi Dimensional

Example One

```
int a[2][2][3] = {
     { {1, 1, 0}, {2, 0, 0} },
     { {3, 0, 0}, {4, 4, 0} },
};
```

Example Two

```
int a[ ][2][3] = {
     { {1, 1, 0}, {2, 0, 0} },
     { {3, 0, 0}, {4, 4, 0} },
};
```

9.5 An Example — Bubble Sort

The Bubble Sort Algorithm

- Sorting means to re-arrange a list of items so that they appear in increasing (or decreasing) order.
- One of the most simple sorting algorithms is the bubble sort, which uses the Archimedes Principle: lighter materials should rise to rest on top of heavier materials.
- The bubble sort algorithm is given below:
  - The algorithm consists of two `for` loops, one nested inside the other.
  - In each pass of the outer loop, the next "lightest" element is "bubbled" to as far "top" as possible. The inner loop is responsible for the bubbling process.
  - The inner loop always starts from the "bottom" and tries to move "lighter" elements in the upward direction by comparing and possibly swapping adjacent elements.
  - The algorithm is guaranteed to sort `n` data values in at most `n - 1` outer iterations.

Program array_5.c

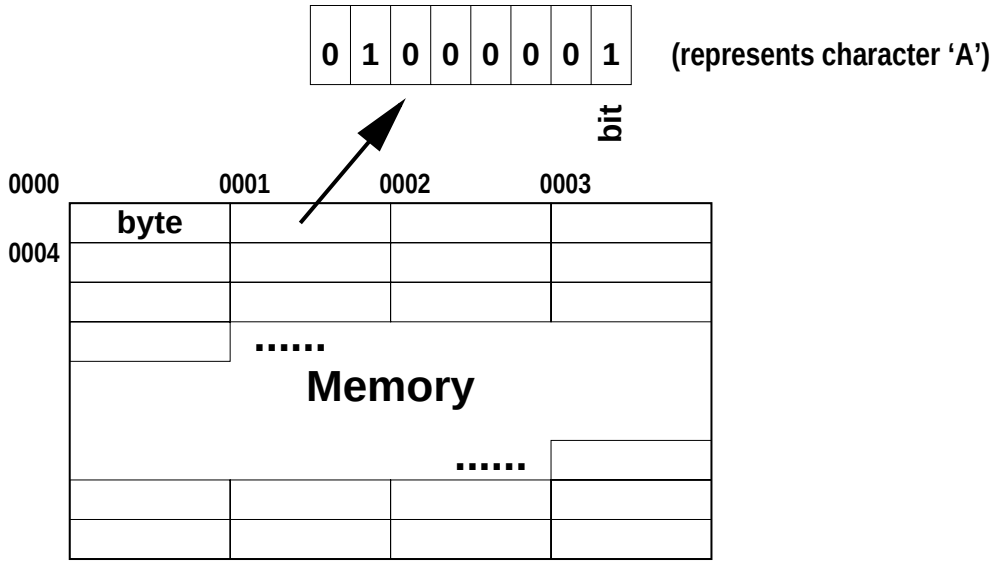
```
#include <stdio.h>
#define SIZE 6

int main(void)
{
     int     i, j, temp;
     int     list[SIZE] = { 22, 47, 31, 16, 9, 5 };


     for (i=0; i < SIZE-1; i++)
          for (j=SIZE-1; j>i; j--)
               if (list[j-1] > list[j]) {
                    temp = list[j-1];
                    list[j-1] = list[j];
                    list[j] = temp;
               }

     for (i=0; i<SIZE; i++)
          printf("%d ", list[i]);
     printf("\n");

     return 0;
}
```

```
Original list:   22 47 31 16 9 5

i=0
After j=5:       22 47 31 16 5 9
After j=4:       22 47 31 5 16 9
After j=3:       22 47 5 31 16 9
After j=2:       22 5 47 31 16 9
After j=1:       5 22 47 31 16 9

i=1
After j=5:       5 22 47 31 9 16
After j=4:       5 22 47 9 31 16
After j=3:       5 22 9 47 31 16
After j=2:       5 9 22 47 31 16

i=2
After j=5:       5 9 22 47 16 31
After j=4:       5 9 22 16 47 31
After j=3:       5 9 16 22 47 31

i=3
After j=5:       5 9 16 22 31 47
After j=4:       5 9 16 22 31 47

i=4
After j=5:       5 9 16 22 31 47
```

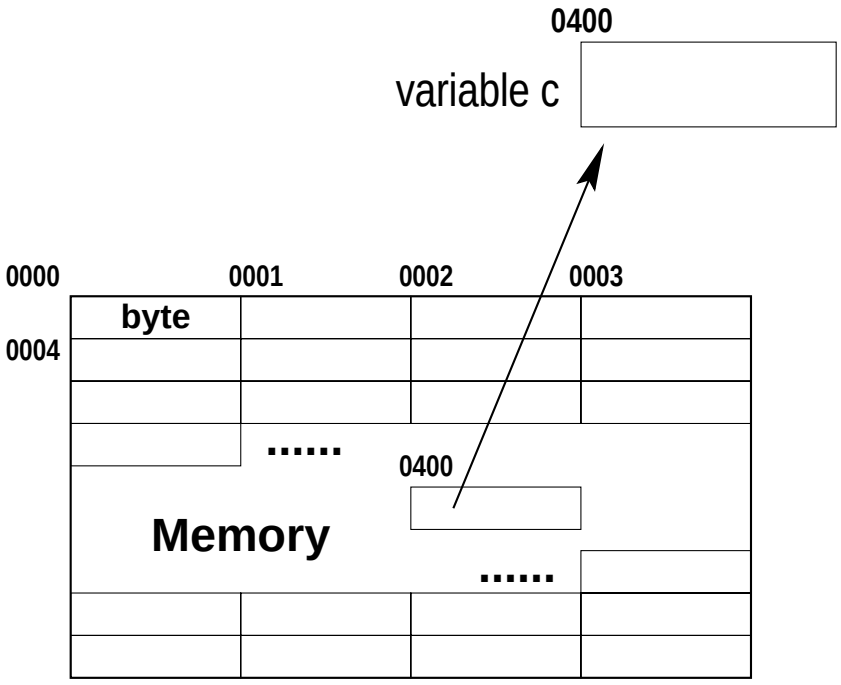
□ **End.**

Lecture 10

Addresses, Pointers, & Arrays Revisited



10.1 Memory Addresses and Pointers

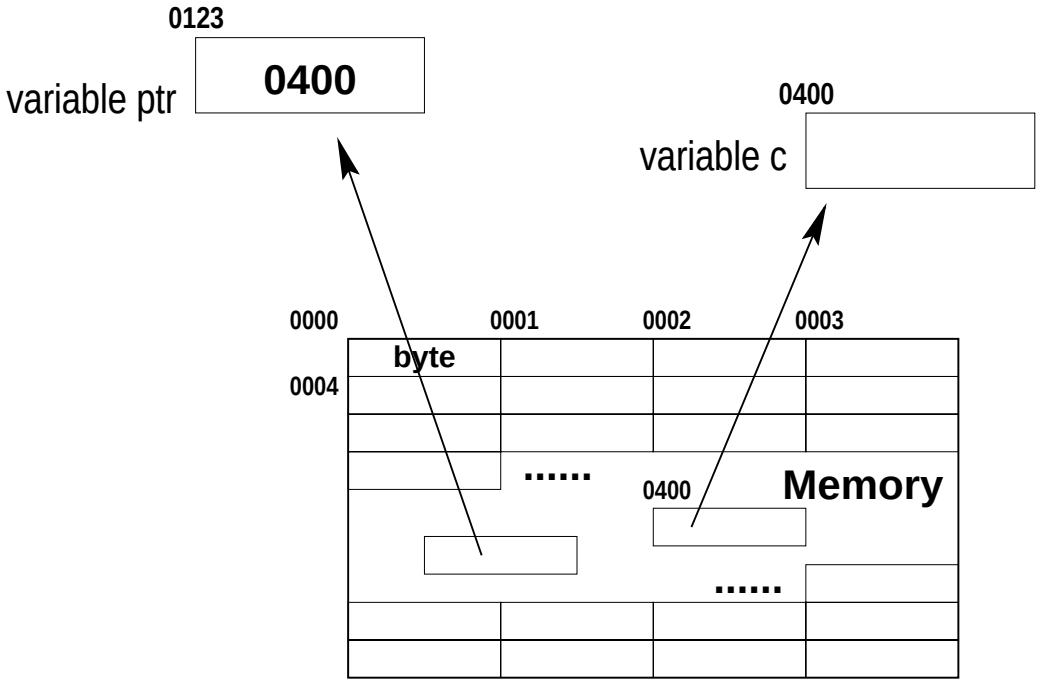


Variables and Memory Addresses

```
char c;

scanf("%c", &c);
```

- `&c` denotes the address of variable `c` in memory, that is, 0400.
- `scanf("%c", &c);`
  - causes the input character to be stored in memory address of `c` (i.e. 0400).
  - effectively stores the character into variable `c`.

10.1.1 Defining Pointers



Pointer to Variables

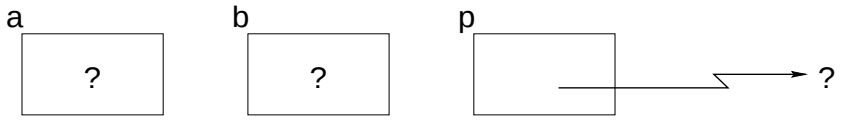
```
char      c;
char      *ptr;

ptr = &c;
```

- `char *ptr;`
  - Declares the variable `ptr`.
  - Variable name is `ptr`, NOT `*ptr`. The `*` denotes that `ptr` is a pointer.
  - The type of `ptr` is **pointer to char**, or simply **char pointer**.
  - Meaning that, the variable `ptr` stores the memory address of a `char` variable.
- `ptr = &c;`
  - `&c` returns the memory address where variable `c` is stored (i.e. 0400).
  - Stores the address 0400 into the variable `ptr` using the ordinary assignment operator `=`.

10.1.2 Addressing and Dereferencing

Step 1

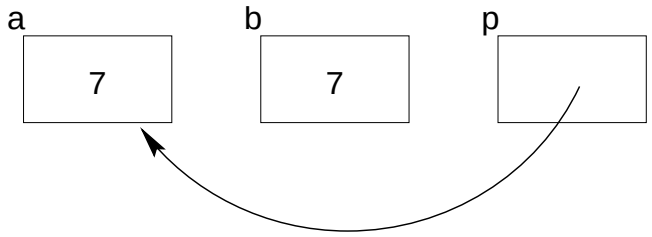
```
int     a, b, *p;
```



- Declare the two integer variables `a` and `b`, and the pointer variable `p`.

Step 2

```
a = b = 7;
p = &a;
```



- Stores the address of variable `a` into pointer `p`.

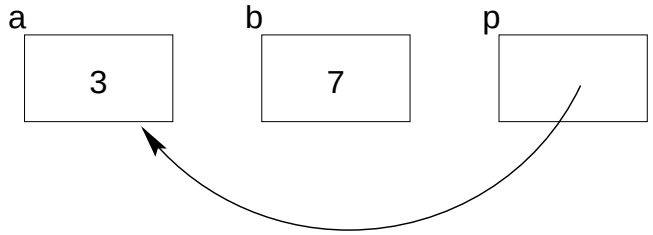
Step 3

- Now we can use pointer `p` to access the value of variable `a` by using the *indirection* (or called the *dereference*) operator `*`.
- `*p` refers to the value of the variable to which `p` points.
- Example,

```
printf("a = %d\n", *p);
```

Step 4

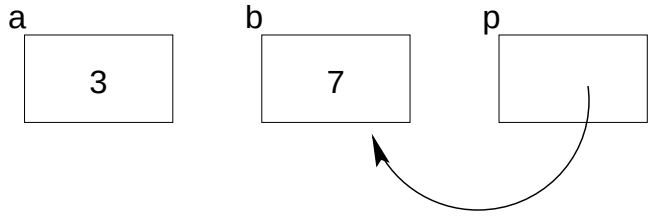
```
*p = 3;
printf ("a = %d\n", a);
```



- When `*p` appears on the L.H.S. of an assignment, it means the value on the R.H.S. is to be written onto the memory location to which `p` points.

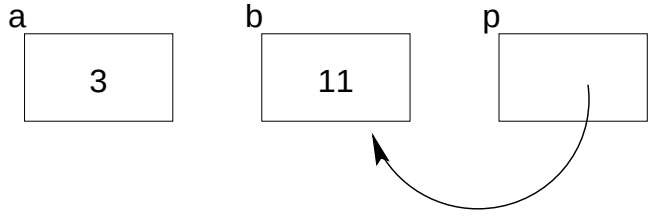
Step 5

```
p = &b;
```



Step 6

```
*p = 2 * *p - a;
printf("b = %d\n", b);
```



10.2 Pointer Assignments

Pointer Assignment

```
int    x, *ptr1, *ptr2;
char   c, *ptr3;

ptr1 = &x;
ptr2 = ptr1;        /* valid */
ptr3 = &c;
ptr2 = ptr3;        /* invalid */
```

- One pointer can be assigned to another only when they both have the same type.

The `void` Pointer

```
int    x, *xptr;
char   *cptr;
void   *vptr;

xptr=&x;
vptr = xptr;        /* valid */
cptr = vptr;        /* valid */
```

- However, pointer assignment is allowed when one of the operands is of type "pointer to void".
- We can think of `void *` as a generic pointer type.

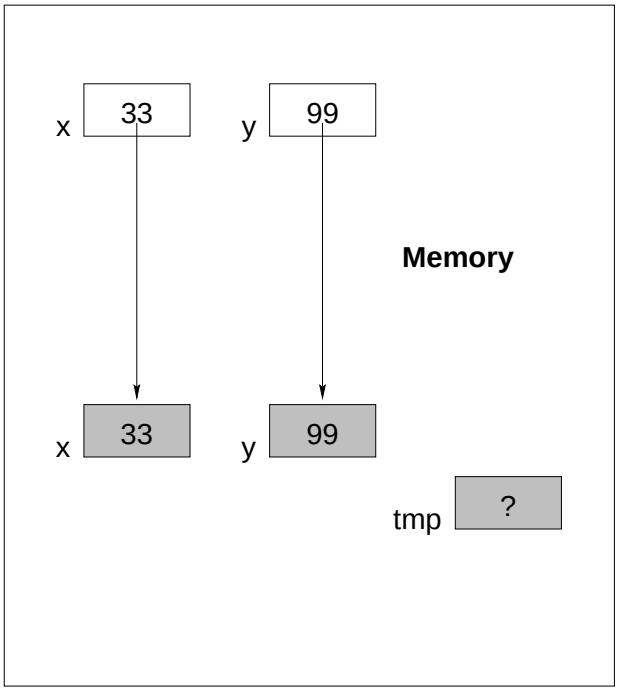
10.3 Call-by-Value

Program point_1.c

```
#include <stdio.h>
void wrong_swap (int, int);

int main(void)
{
     int  x=33,  y=99;

     printf ("x=%d, y=%d\n", x, y);
     wrong_swap(x, y);
     printf ("x=%d, y=%d\n", x, y);
     return 0;
}

void wrong_swap (int x, int y)
{
     int  tmp;

     printf ("\t x=%d, y=%d\n", x, y);
     tmp = x;
     x = y;
     y = tmp;
     printf ("\t x=%d, y=%d\n", x, y);
}
```

```
x=33, y=99
         x=33, y=99
         x=99, y=33
x=33, y=99
```

```
int main(void)
{
        int     x=33, y=99;

        ......
        wrong_swap(x, y);
        ......
}


void wrong_swap(int x,  int y)
{
        int     tmp;

        ......
        ......
}
```



Call-by-Value

- Whenever variables are passed as arguments to a function, their values are **copied** to the corresponding function parameters.
- The parameters have a lifespan identical to the lifespan of the function.
- That is, the parameters are created when the function is entered, and are destroyed (freed from memory) when the function returns.
- No matter how the value of the parameters changes, the variables in the calling environment (`main` in the example) are not changed.

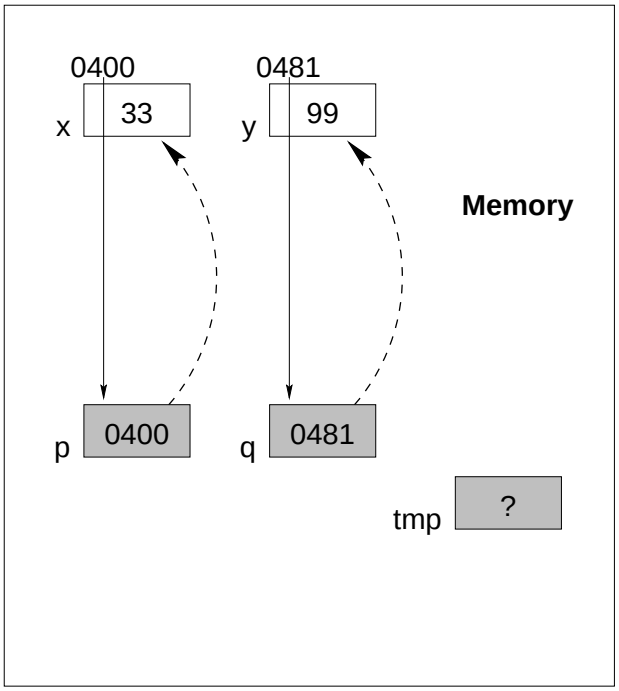
10.4 Call-by-Reference

Program point_2.c

```
#include <stdio.h>
void swap (int *, int *);

int main(void)
{
     int  x=33,  y=99;

     printf ("x=%d, y=%d\n", x, y);
     swap(&x, &y);
     printf ("x=%d, y=%d\n", x, y);
     return 0;
}

void swap (int *p, int *q)
{
     int  tmp;

     printf ("\t *p=%d, *q=%d\n", *p, *q);
     tmp = *p;
     *p = *q;
     *q = tmp;
     printf ("\t *p=%d, *q=%d\n", *p, *q);
}
```

```
x=33, y=99
         *p=33, *q=99
         *p=99, *q=33
x=99, y=33
```

```
int main(void)
{
        int     x=33, y=99;

        ......
        swap (&x, &y);
        ......
}


void swap(int *p, int *q)
{
        int     tmp;

        ......
        ......
}
```



Call-by-Reference

- Passing an address as an argument when the function is called.
- Declaring a formal parameter to be a pointer.
- Using the dereferenced pointer in the function body.

10.5 Pointer Arithmetic

Program point_3.c

```
#include <stdio.h>

int main(void)
{
        int     ivar, *iptr;
        char    cvar, *cptr;

        iptr = &ivar;
        cptr = &cvar;

        printf ("Original\n");
        printf ("iptr = %p\n", iptr);
        printf ("cptr = %p\n", cptr);

        printf ("\nIncremented\n");
        printf ("iptr = %p\n", ++iptr);
        printf ("cptr = %p\n", ++cptr);

        return 0;
}
```

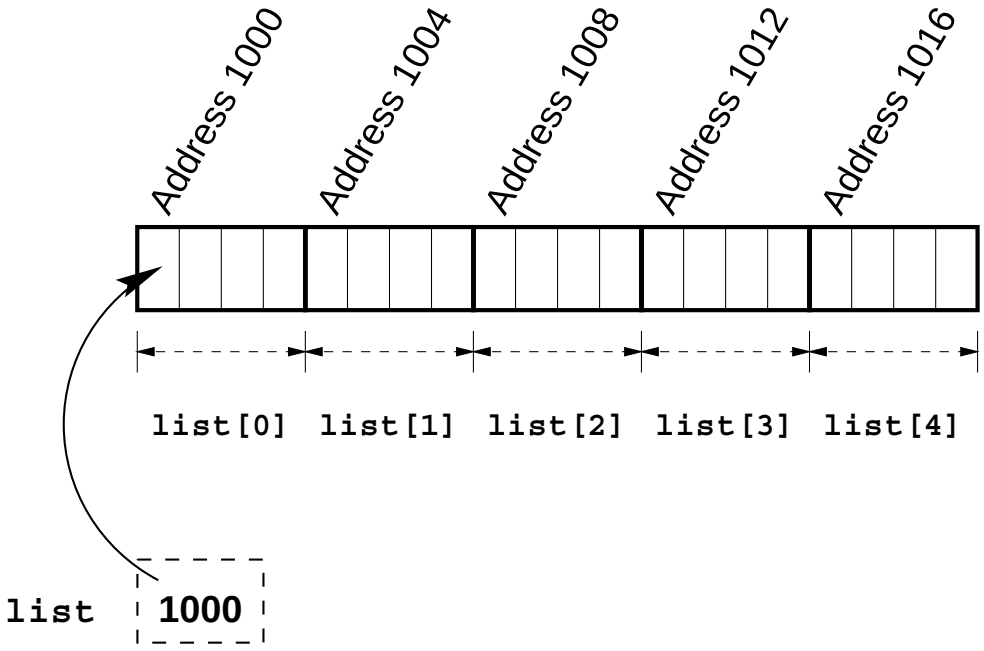
```
Original
iptr = effffc0c
cptr = effffc07

Incremented
iptr = effffc10
cptr = effffc08
```

Pointer Arithmetic

- When pointer `iptr` is incremented, its value is raised by 4, in the example.
- However, when pointer `cptr` is incremented, its value is raised only by 1.
- *If a pointer variable is incremented, its value is actually increased by* `sizeof(type)`*, where* `type` *is the data type that the pointer points to.*
- This allows the pointer to point to the next piece of data item in the memory meaningfully.
- Other arithmetic operators (`+`, `-`, `*`, ...) when applied to pointer variables behave similarly.

10.6 Arrays and Memory Addresses

10.6.1 One-Dimensional Arrays

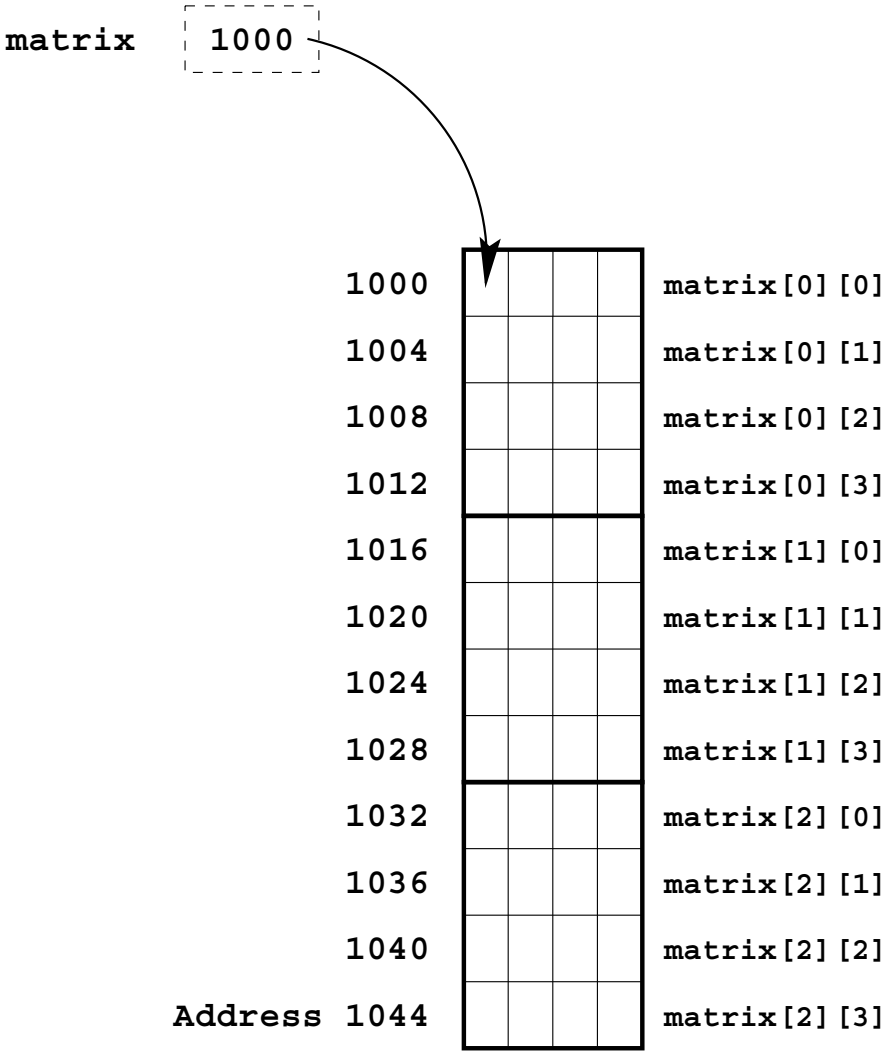


One-Dimensional Arrays

- Assuming 4-byte integers.
- Assuming that the array is stored at memory location 1000 onwards.
  - `list[0]` begins at 1000
  - `list[1]` begins at 1004
  - `list[2]` begins at 1008
  - and so on.
- *An array name without subscript denotes the address of the first element of the array.*
- In the example,
  - `list` refers to the address of `list[0]`, i.e. 1000.
  - This address is referred to as the *base address* of the array.
  - `list` is actually a constant address value since the location of the array will not be changed during program lifetime.
- Starting address of `list[i]` is given by

`list + sizeof(int)` $\times$ `i`

10.6.2 Two-Dimensional Arrays



Two-Dimensional Arrays

- Assuming 4-byte integers.
- Assuming that the array is stored at memory location 1000 onwards.
  - `matrix[0][0]` begins at 1000.
  - `matrix[0][1]` begins at 1004.
  - `matrix[0][2]` begins at 1008.
  - and so on.
- Note that the array is stored in the "row-major" discipline.
  - Row 0 starts at address 1000.
  - Row 1 starts at address 1016.
  - Row 2 starts at address 1032.
- An array name without subscript (`matrix`) denotes the address of the first element of the array (`matrix[0][0]`).
- For a 2-dimensional `int` array `matrix[ROW][COL]`, the starting address of `matrix[i][j]` is given by

  `matrix + sizeof(int)` $\times$ `(COL` $\times$ `i + j)`

10.7 Passing Arrays to Functions

10.7.1 One-Dimensional Arrays

Program point_4.c

```
#include <stdio.h>
#define SIZE 5

int fsum(int []);

int main(void)
{
     int  list[SIZE];
     int  i;

     for (i=0; i<SIZE; i++) {
          printf("number = ? ");
          scanf("%d", &list[i]);
     }

     printf("sum = %d\n", fsum(list));
     return 0;
}

int fsum (int array[])
{
     int  i, sum=0;

     for (i=0; i<SIZE; i++)
          sum += array[i];
     return sum;
}
```

Passing an 1-D Array to a Function

- In the calling environment `main()`:
  - The name of the array is used as the parameter.
  - The bracket pair `[]` must be omitted.
- In the called function `fsum()`:
  - The bracket pair `[]` is required to tell the compiler that the parameter is an array.
  - The array size can be omitted. In fact, even if the array size is given, it is ignored by the compiler!
  - Why?

Program point_5.c

```
#include <stdio.h>
#define SIZE 5

void times2(int []);

int main(void)
{
     int  list[SIZE];
     int  i;

     for (i=0; i<SIZE; i++) {
          printf("number = ? ");
          scanf("%d", &list[i]);
     }

     times2(list);

     printf("\n");
     for (i=0; i<SIZE; i++) {
          printf("list[%d] = %d\n", i, list[i]);
     }
     return 0;
}

void times2(int array[])
{
     int i;

     for (i=0; i<SIZE; i++)
          array[i] *= 2;
}
```

```
number = ? 1
number = ? 2
number = ? 3
number = ? 4
number = ? 5

list[0] = 2
list[1] = 4
list[2] = 6
list[3] = 8
list[4] = 10
```

The Call-By-Reference Mechanism

- In the called function `times2()`, elements in `array[]` are doubled.
- In the calling environment `main()`, after calling `times2()`, elements in `list[]` are also doubled!
- Why?
  - The array `list[]` is not copied to the function `times2()`.
  - Instead, `array[]` in `times2()` refers to the same array `list[]` in `main()`.
  - Thus, modifying any element in `array[]` causes modifications to be made on `list[]` actually.
- *Call-By-Reference*

10.7.2 Two-Dimensional Arrays

Program point_6.c

```
#include <stdio.h>
#define   ROW   3
#define   COL   5

int fsum(int [][COL]);

int main(void)
{
   int   matrix[ROW][COL], i, j;

   for (i = 0; i < ROW; ++i)
      for (j = 0; j < COL; ++j)
         scanf("%d", &matrix[i][j]);

   printf ("sum = %d\n", fsum(matrix));
   return 0;
}

int fsum(int array[][COL])
{
   int   i, j, sum=0;;

   for (i = 0; i < ROW; ++i)
      for (j = 0; j < COL; ++j)
         sum += array[i][j];

   return sum;
}
```

Passing a 2-D Array to a Function

- In the calling environment `main()`:
  - The name of the array is used as the parameter.
  - The two bracket pairs `[][]` are omitted.
- In the called function `fsum()`:
  - The two bracket pairs `[][]` are required to tell the compiler that the parameter is a 2-D array.
  - The first dimension size can be omitted, while the second dimension size must be specified, i.e., `array[][COL]`.
  - Why?
- *Passing a 2-D array to a function also employs the call-by-reference mechanism.*

10.7.3 Multi-Dimensional Arrays

Passing a Multi-D Array to a Function

- In the calling environment:
  - The name of the array is used as the parameter.
  - The bracket pairs `[][][]...` are omitted.
  - Example,

    ```
    fsum(array_3d);
    ```

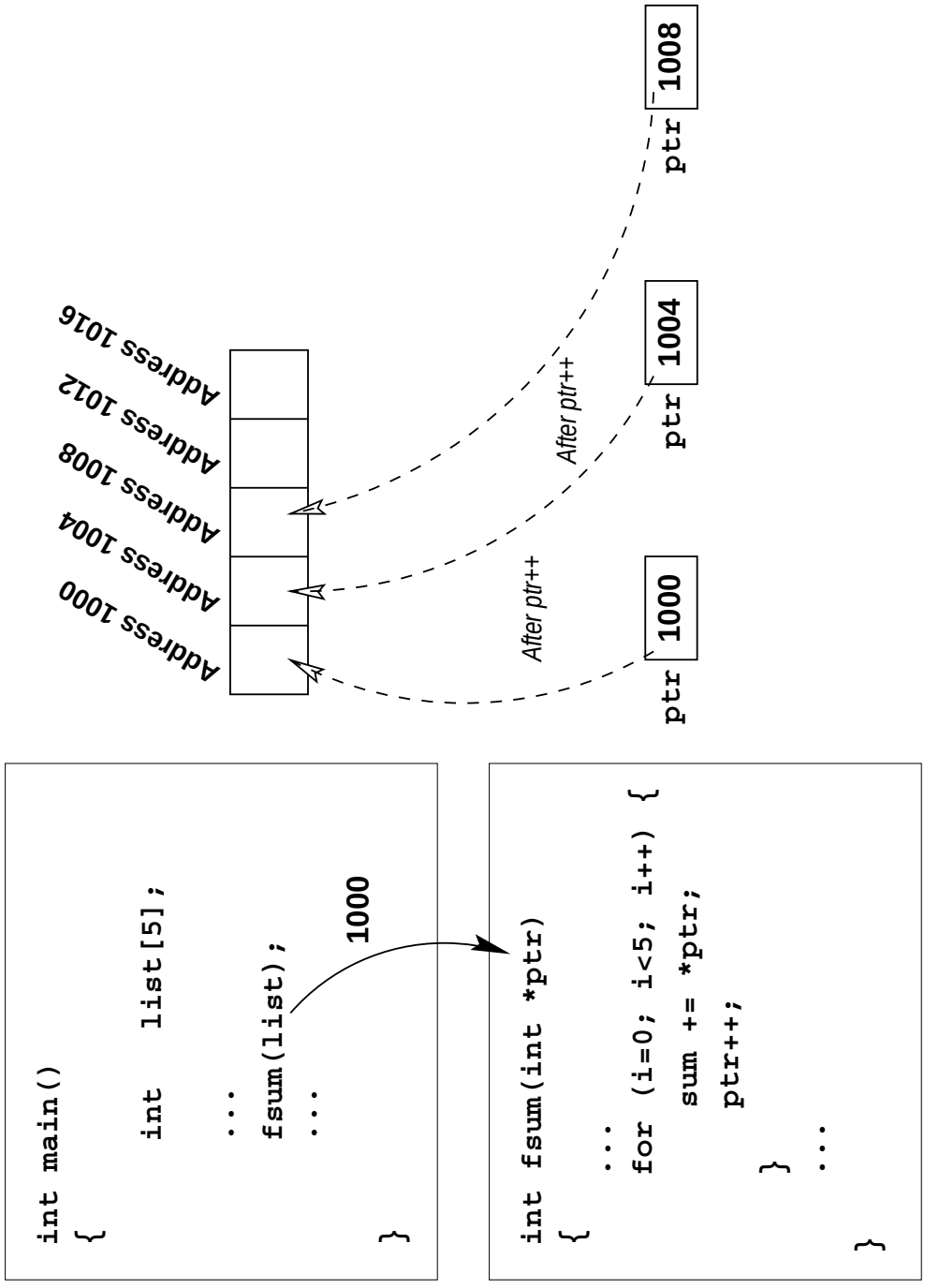
- In the called function:
  - The bracket pairs `[][][]...` are required to tell the compiler that the parameter is a multi-dimensional array.
  - The first dimension size can be omitted.
  - All the other dimension sizes must be specified.
  - Example,

    ```
    int fsum(int array_3d[][10][20])
    {
        ...
    }
    ```

10.7.4 An Alternative View

Program point_7.c

```
#include <stdio.h>
#define SIZE 5

int fsum(int *);

int main(void)
{
     int  list[SIZE];
     int  i;

     for (i=0; i<SIZE; i++) {
          printf("number = ? ");
          scanf("%d", &list[i]);
     }

     printf("sum = %d\n", fsum(list));
     return 0;
}

int fsum (int *ptr)
{
     int  i, sum=0;

     for (i=0; i<SIZE; i++) {
          sum += *ptr;
          ptr++;
     }
     return sum;
}
```


int main()
{
int list[5];
...
fsum(list);
...
}
1000
int fsum(int *ptr)
{
...
for (i=0; i<5; i++) {
sum += *ptr;
ptr++;
}
...
}
Address 1000
Address 1004
Address 1008
Address 1012
Address 1016
ptr
1000
After ptr++
ptr
1004
After ptr++
ptr
1008

Explanation

- In the calling environment `main()`:
  - The name of the array is used as the parameter.
  - It's actually the base address of the array to be passed (i.e. 1000 in the diagram).
  - No new array is created.
- Call-by-Reference!!
- In the called function `fsum()`:
  - `ptr` is an *integer pointer* receiving the address value passed as function parameter (i.e. 1000).
  - `ptr` thus pointing to the first array element of `list[]`.
  - `ptr` points to the next array element when it is being incremented (pointer arithmetic!).
- What's the difference between the two methods?

```
int sum (int list[]);

int sum (int *ptr);
```

10.7.5 Further Example

Program point_8.c

```
#include <stdio.h>
#define SIZE  4

int skip_sum(int *, int);

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

     while (1) {
          printf("Skip how many? ");
          scanf("%d", &n);
          if (n < 0)
               break;
          else
               printf("Sum = %d\n", skip_sum(array, n));
     }
     printf ("Bye!\n");
     return 0;
}

int skip_sum(int *ptr, int skip)
{
     int  i, sum=0;

     ptr = ptr + skip;
     for (i=1; i <= (SIZE-skip); i++, ptr++)
          sum += *ptr;
     return sum;
}
```

```
Skip how many? 1
Sum = 9
Skip how many? 2
Sum = 7
Skip how many? 3
Sum = 4
Skip how many? 4
Sum = 0
Skip how many? -1
Bye!
```

□ **End.**

Lecture 11

Structures and Unions

11.1 Structures

What are Structures?

- A structure is a collection of related elements, possibly of different types, having a single name.
- The elements in a structure can be of the same or different types. But all elements in a structure must be related.

11.1.1 Structure Declaration

Structure Type

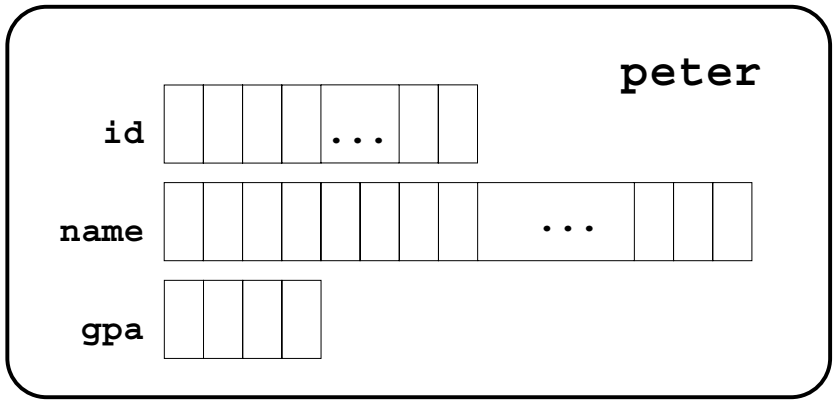
- A structure **type** can be defined using the keyword `struct`. For example,

```
struct student {
     char     id[9];
     char     name[26];
     float    gpa;
};
```

  This defines a new data type called `struct student`, which consists of three **members**.

- However, no variable storage has yet been allocated.
- Structure **variables** can be defined just like variables of simple data types. For example,

```
struct student    peter;
```



Global Structure Type

Usually a `struct` type is defined globally. For example,

```
#include <stdio.h>

struct student {
     char     id[9];
     char     name[26];
     float    gpa;
};

int main(void)
{
     struct student    peter;

     ...
}
```

More Examples

- Complex number

```
struct complex {
     double   real;
     double   imaginary;
};
```

- Employee record

```
struct employee {
     char     name[50];
     char     position;
     float    salary;
};
```

- A card

```
enum suits {CLUBS, DIAMONDS, HEARTS, SPADES};

struct card {
     enum suits   suit;
     int          rank;
};
```

11.1.2 Accessing `struct` Members

Program struc_1.c

```
#include <stdio.h>

struct date {
    int    day;
    int    month;
    int    year;
};

int main(void)
{
        struct  date    today;

        today.day = 25;
        today.month = 12;
        today.year = 1997;

        if (today.day==25 && today.month==12)
                printf("Merry X'mas!\n");
        return 0;
}
```

The Member Operator

Members of a `struct` variable can be accessed using the **member operator** (.).

```
struct_var.member
```

where `struct_var` is a `struct` variable and `member` is a defined component of the `struct`.

Program struc_2.c

```
#include <stdio.h>
struct date {
     int    day;
     int    month;
     int    year;
};

int main(void)
{
     struct date     today, dob;
     int             age;

     printf("Date of birth (dd mm yy)? ");
     scanf("%d %d %d", &dob.day, &dob.month, &dob.year);

     printf("Today (dd mm yy)? ");
     scanf("%d %d %d", &today.day, &today.month, &today.year);

     if (today.month > dob.month ||
         (today.month == dob.month && today.day >= dob.day))
             age = today.year - dob.year;
     else
             age = today.year - dob.year - 1;
     printf("Age = %d\n", age);
     return 0;
}
```

```
Date of birth (dd mm yy)? 27 04 87
Today (dd mm yy)? 21 11 97
Age = 10
```

11.1.3 Initializing Structures

Example One

- `struct` variables can be initialized as follows:

```
struct date {
     int      day;
     int      month;
     int      year;
};

struct date birthday = {5, 12, 91};
```

- The initial values should be constant values or constant expressions. That is, no variables should be involved in the initializer expressions.
- For automatic `struct` variables, if fewer initializers are listed than the number of members in the `struct`, values of the remaining members are *undefined*.

Example Two

```
struct employee {
     char     name[50];
     char     position;
     float    salary;
};

struct employee peter={"Peter Chen", 'P', 8900.5};
```

11.1.4 Array Members of Structures

Program struc_3.c

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

#define NAMEMAX    30
typedef enum {FEMALE, MALE} Gender;

struct person {
     char surname[NAMEMAX+1];
     char forename[NAMEMAX+1];
     Gender  sex;
     int  age;
};

int main(void)
{
     struct person    user;
     struct person    computer = { "Machine", "Computing",
                                   FEMALE, 50 };

     printf("Your surname? ");
     gets(user.surname);
     printf("Your forename? ");
     gets(user.forename);

     printf("Dear %c. %s\n", user.forename[0], user.surname);
     printf("I am %s.\n", computer.forname);

     return 0;
}
```

```
Your surname? Collins
Your forename? Phil
Dear P. Collins
I am Computing.
```

Entities	**Meaning**
`computer`	the whole structure
	`struct person`
`computer.surname`	the whole string member `surname`
	`char []`
	`Machine`
`computer.surname[0]`	1st character of string `computer.surname`
	`char`
	`'M'`
`computer.forename[3]`	16th character of `computer.forename`
	`char`
	`'p'`
`computer.age`	the member `age`
	`int`
	50

11.1.5 Alternative `struct` Declarations

Structure Variable

```
struct {
     char     id[9];
     char     name[26];
     float    gpa;
} peter;
```

- This defines one single `struct` variable `peter`.

Type-Defined Structure

```
typedef struct {
     char     id[9];
     char     name[26];
     float    gpa;
} STUDENT;

STUDENT peter;
```

- This defines the data type `STUDENT`, which is a `struct`.
- Note that

  ```
  STUDENT peter;
  ```

  is sufficient; rather than

  ```
  struct STUDENT peter;
  ```

Program struc_4.c

```
#include <stdio.h>

struct date {
     int    day;
     int    month;
     int    year;
};

typedef struct date date_t;

int main(void)
{
     date_t     today, dob;
     int        age;

     printf("Date of birth (dd mm yy)? ");
     scanf("%d %d %d", &dob.day, &dob.month, &dob.year);

     printf("Today (dd mm yy)? ");
     scanf("%d %d %d", &today.day, &today.month, &today.year);

     if (today.month > dob.month ||
         (today.month == dob.month && today.day >= dob.day))
             age = today.year - dob.year;
     else
             age = today.year - dob.year - 1;

     printf("Age = %d\n", age);
     return 0;
}
```

11.1.6 Alignment Requirement and `struct`

What is an Alignment Requirement?

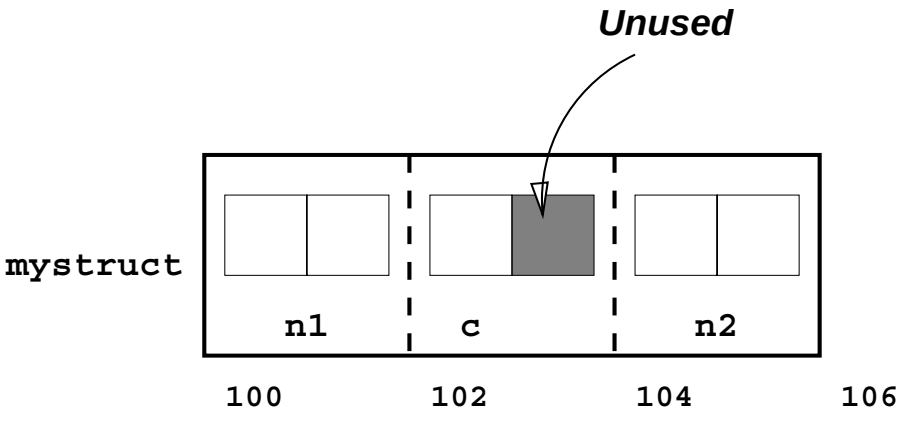
Many CPUs or operating systems require that multi-byte data items be placed at an address that is a multiple of 2 (or 4) for efficient data fetching.

Example

- Assuming that an integer occupies two bytes and a character occupies one byte.
- Assuming that the machine requires alignment on double byte boundary addresses.

```
struct alignment {
     int      n1;
     char     c;
     int      n2;
} mystruct;
```

- The above `struct` variable might be placed in memory depicted as follows.



11.1.7 Structure Assignments

Program struc_5.c

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

#define NAMEMAX   30

typedef enum {FEMALE, MALE} Gender;

struct person {
     char    surname[NAMEMAX+1];
     char    forename[NAMEMAX+1];
     Gender  sex;
     int     age;
};
typedef struct person person_t;

int main(void)
{
     person_t     computer1;
     person_t     computer2 = { "Machine", "Computing",
                                FEMALE, 50};

     computer1 = computer2;  /* struct assignment */

     printf("computer 1: %s %s\n",
                     computer1.forename, computer1.surname);
     printf("computer 2: %s %s\n",
                     computer2.forename, computer2.surname);
     return 0;
}
```

```
computer 1: Computing Machine
computer 2: Computing Machine
```

Structure Assignments

- The assignment operator '=' can be applied to `struct` variables.
- Individual member of the source `struct` is *copied* to the corresponding member of the destination `struct`.
- **This applies to array members**.

11.2 Structures and Functions

11.2.1 Structures as Function Parameters

Program struc_6.c

```
#include <stdio.h>

struct rational {
     unsigned int    num;
     unsigned int    den;
};

float r_Real(struct rational rat)
{
     return (float) rat.num / rat.den;
}

int main(void)
{
     struct rational rat={3, 4};
     float  f;

     f = r_Real(rat);
     printf("float=%.2f\n", f);

     return 0;
}
```

```
float=0.75
```

11.2.2 Structures as Function Return Values

Program struc_7.c

```
#include <stdio.h>

struct rational {
     unsigned int    num;
     unsigned int    den;
};

float r_Real(struct rational rat)
{
     return (float) rat.num / rat.den;
}

struct rational r_Create(unsigned int num, unsigned int den)
{
     struct rational  rat;
     unsigned int     gcd;

     gcd = hcf(num, den);    /* assuming existence of hcf() */
     rat.num = num / gcd;
     rat.den = den / gcd;
     return (rat);
}

int main(void)
{
     struct rational rat;
     float  f;

     rat = r_Create(6, 8);
     f = r_Real(rat);
     printf("float=%.2f\n", f);
     return 0;
}
```

11.2.3 Complete Example

Program struc_8.c

```
#include <stdio.h>

typedef struct {
     unsigned int    num;
     unsigned int    den;
} Rational;

float r_Real(Rational rat)
{
     return (float) rat.num / rat.den;
}

Rational r_Create(unsigned int num, unsigned int den)
{
     Rational       rat;
     unsigned int   gcd;

     gcd = hcf(num, den);
     rat.num = num / gcd;
     rat.den = den / gcd;
     return (rat);
}

Rational r_Add(Rational rat1, Rational rat2)
{
     unsigned int    num, den;

     num = rat1.num * rat2.den + rat2.num * rat1.den;
     den = rat1.den * rat2.den;
```

```
    return r_Create(num, den);
}

int main(void)
{
    Rational rat1, rat2;

    rat1 = r_Create(6, 8);
    rat2 = r_Create(1, 2);
    printf("float=%.2f\n", r_Real(r_Add(rat1, rat2)));

    return 0;
}
```

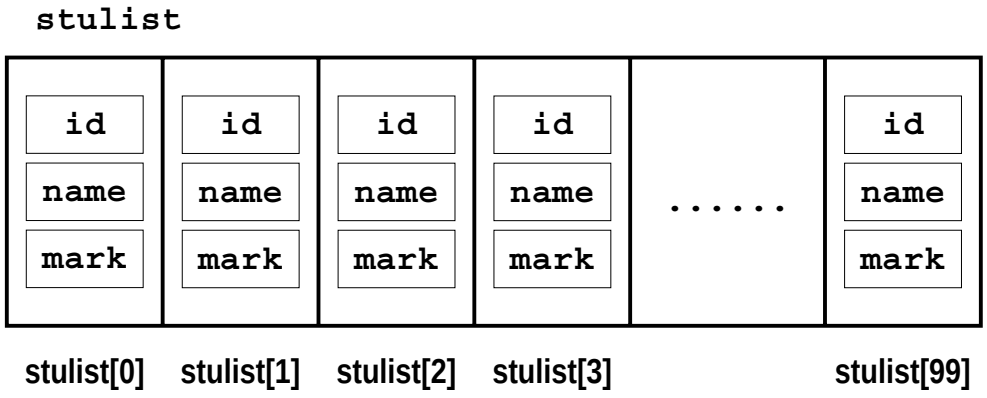
11.3 Miscellaneous Topics on Structures

11.3.1 Array of Structures

What is an Array of Structures?

- A structure may occur as an array element.
- Example:

```
struct student {
     char     id[9];
     char     name[30];
     int      mark;
};

int main(void)
{
     struct student   stulist[100];
...
}
```



Entities	**Meaning**
`stulist`	an array of `struct student`
`stulist[1]`	the 2nd element of `stulist` array
	a variable of type `struct student`
`stulist[i].name`	the name of the $(i+1)$th student in the `stulist` array
	a string
`stulist[i].name[j]`	the $(j+1)$th character of the name of the $(i+1)$th student in the `stulist` array
	a single character

11.3.2 Nested Structures

Program struc_9.c

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

struct date {
     int     day;
     int     month;
     int     year;
};

struct book {
     char            author[30];
     char            title[50];
     char            publisher[30];
     int             edition;
     struct date     date_of_pub;
};

int main(void)
{
     struct book     abook, booklist[100];

     abook.date_of_pub.day = 10;
     abook.date_of_pub.month = 2;
     abook.date_of_pub.year = 1997;

     strcpy(booklist[10].author, "Al Kelly");
     strcpy(booklist[10].title, "C By Dissection");
     strcpy(booklist[10].publisher, "Addison-Wesley");
     booklist[10].edition = 3;
     booklist[10].date_of_pub.day = 1;
```

```
    booklist[10].date_of_pub.month = 2;
    booklist[10].date_of_pub.year = 1996;

    return 0;
}
```

11.3.3 Pointer to Structures

Program struc_10.c

```
#include <stdio.h>

struct date {
     int    day;
     int    month;
     int    year;
};

int main(void)
{
     struct date   today;
     struct date   *ptrDate;

     /*
     today.day = 25;
     today.month = 12;
     today.year = 1997;
     */

     ptrDate=&today;

     (*ptrDate).day = 25;
     (*ptrDate).month = 12;
     (*ptrDate).year = 1997;

     if (today.day==25 && today.month==12)
             printf("Merry X'mas!\n");

     return 0;
}
```

Program struc_11.c

```
#include <stdio.h>

struct date {
     int    day;
     int    month;
     int    year;
};

int main(void)
{
     struct  date    today;
     struct  date    *ptr;

     ptr=&today;

     ptr->day = 25;
     ptr->month = 12;
     ptr->year = 1997;

     if (today.day==25 && today.month==12)
          printf("Merry X'mas!\n");

     return 0;
}
```

Program struc_12.c

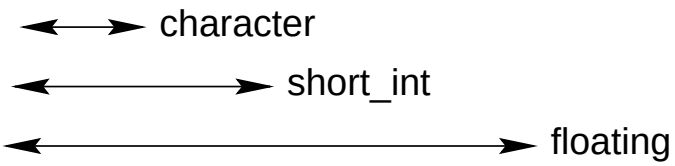
```
#include <stdio.h>

struct date {
     int    day;
     int    month;
     int    year;
};

int main(void)
{
     struct  date    today;
     struct  date    *ptr=&today;

     printf("day? ");
     scanf("%d", &ptr->day);

     printf("month? ");
     scanf("%d", &ptr->month);

     printf("year? ");
     scanf("%d", &ptr->year);

     if (ptr->day==25 && ptr->month==12)
          printf("\nMerry X'mas!\n");

     return 0;
}
```

11.4 Union

What is a Union?

- Union is **similar** to structure in that both have several members.
- But a union can hold value in **only one** of its member at any time.
- Members are **overlaid** in the storage allocation for a union variable.
- The compiler allocates sufficient storage to accommodate the **largest** member of a union.
- Example,

```
union num {
     char     character;
     short    short_int;
     float    floating;
};

union num data;
```



Program struc_13.c

```
#include <stdio.h>

union dual {
     int       integer;
     float     floating;
};

int main(void)
{
     union dual  data;

     data.integer = 20000;
     printf("%12d %10.4f\n", data.integer, data.floating);

     data.floating = 123.0;
     printf("%12d %10.4f\n", data.integer, data.floating);

     return 0;
}
```

```
       20000     0.0000
  1123418112   123.0000
```

Explanation

- Just like `struct`, we use the . operator to access members of a union variable.
- It's the programmer's responsibility to keep track of what data type is currently stored in a union variable.

11.4.1 The use of a Tag in `union`

Using a Tag in `union`

- Unions are commonly used with a **tag**.
- A tag indicates which member of the union variable is **active**.
- A structure is used to hold both a tag and a union variable.

Program struc_14.c

```
#include <stdio.h>

enum type {INT, FLOAT};

union udual {
     int       integer;
     float     floating;
};

struct dual {
     enum type     tag;
     union udual   value;
};
```

```
void inc_print(struct dual data)
{
     switch (data.tag) {
          case INT:
               data.value.integer++;
               printf("Integer: %d\n", data.value.integer);
               break;
          case FLOAT:
               data.value.floating++;
               printf("Decimal: %.2f\n", data.value.floating);
               break;
     }
}

int main(void)
{
     struct dual data;

     data.tag = INT;
     data.value.integer = 123;
     inc_print(data);

     data.tag = FLOAT;
     data.value.floating = 999.0;
     inc_print(data);

     return 0;
}
```

□ **End.**

Lecture 12

Elementary Data Structures

NOTE

- We need data structures that can grow and shrink during execution.
- Implementing dynamic data structures needs two important aspects:
  - *Dynamic memory allocations.*
  - *Self-referential structures.*

12.1 Dynamic Memory Manipulations

12.1.1 Dynamic Memory Allocation — `malloc()`

`malloc()`

```
#include <stdlib.h>

void *malloc(size_t size);
```

where `size_t` has been `typedef`-ed to `unsigned int`, and `size` is the number of bytes required.

- `malloc()` returns a pointer to a block of memory of `size` bytes.
- Returns `NULL` when no memory can be allocated successfully.

Program dyn_1.c

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

int main(void)
{
     int  amount;
     void *ptr;

     printf("How many bytes? ");
     scanf("%d", &amount);

     ptr = malloc(amount);

     if (ptr != NULL)
          printf("Address: %p\n", ptr);
     else
          printf("Failed to allocate memory!\n");

     return 0;
}
```

Program dyn_2.c

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

int main(void)
{
     int     *ptr;

     ptr = malloc(sizeof(int));

     if (ptr!=NULL) {
          *ptr = 97;
          printf("The int stored at %p is %d\n.", ptr, *ptr);
     }

     return 0;
}
```

```
The int stored at 208b0 is 97.
```

12.1.2 Releasing Dynamic Memory — `free()`

`free()`

```
#include <stdlib.h>

void free(void *ptr);
```

where `ptr` is a pointer to a block of memory previously allocated by `malloc()`.

- Release the block of memory pointed to by `ptr`.
- That block of memory is made available for further dynamic allocation.
- If `ptr` is a `NULL` pointer, no action occurs.
- It's important to `free` those memory blocks right after they are no more useful. Why?
- The previous example should thus be:

Program dyn_3.c

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

int main(void)
{
     int     *ptr;

     ptr = malloc(sizeof(int));

     if (ptr!=NULL) {
          *ptr = 97;
          printf("The int stored at %p is %d\n.", ptr, *ptr);
     }

     free(ptr);
     return 0;
}
```

12.1.3 A More Comprehensive Example

Program dyn_4.c

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

int main(void)
{
     int     i, num, *ptr, *baseptr;

     printf("How many integers? ");
     scanf("%d", &num);

     if ( (baseptr=malloc(sizeof(int)*num)) == NULL)
          return 0;

     ptr = baseptr;
     for (i=0; i<num; i++) {
          scanf("%d", ptr);
          ptr++;
     }

     printf("The numbers are: ");
     ptr = baseptr;
     for (i=0; i<num; i++) {
          printf("%d ", *ptr);
          ptr++;
     }

     free(baseptr);

     return 0;
}
```

12.2 Self-Referential Structures

12.2.1 Pointer to Structures

Program dyn_5.c

```
#include <stdio.h>

struct date {
    int    day;
    int    month;
    int    year;
};

int main(void)
{
     struct  date    today;
     struct  date    *ptr;

     ptr=&today;
     ptr->day = 25;
     ptr->month = 12;
     ptr->year = 1997;

     if (today.day==25 && today.month==12)
          printf("Merry X'mas!\n");

     return 0;
}
```

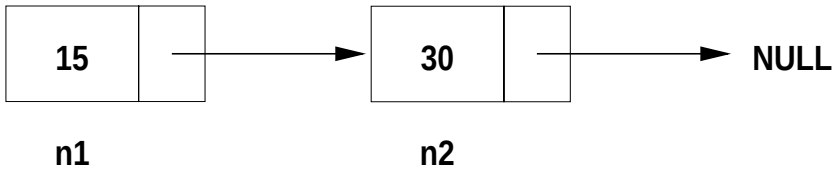
12.2.2 Self-Referential Structures

Basic Idea

- A self-referential structure is a structure that contains a pointer member that points to a structure of the same structure type.
- For example,

```
struct node {
     int            data;
     struct node    *nextptr;
};
```

- The member `nextptr` can be used as a link to "tie" a structure of type `struct node` to another structure of the same type.



Program dyn_6.c

```
#include <stdio.h>

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

int main(void)
{
     struct node    n1, n2;

     n1.data = 15;
     n1.nextptr = &n2;

     n2.data = 30;
     n2.nextpr = NULL;

     return 0;
}
```

NOTE

- The constant `NULL`, defined in `<stdio.h>`, is used when the pointer points to nothing.
- Self-referential structures can be linked together to form useful data structures like lists, queues, stacks, and trees.

12.2.3 Self-Referential Structures and `malloc()`

NOTE

- Dynamic memory allocation is used such that a data structure contains varying number of nodes as required.
- A node is created dynamically when needed. This can be done as follows.

```
struct node {
     int            data;
     struct node    *nextptr;
};

struct node *ptr;

ptr = malloc(sizeof(struct node));
```

- `sizeof(struct node)` determines the size in bytes of a structure of type `struct node`.
- `malloc()` allocates a new area in the computer's physical memory of `sizeof(struct node)` bytes.
- `malloc()` returns the address of the first byte of the newly allocated memory block.
- That address is then stored into the pointer variable `ptr`.

Program dyn_7.c

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

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

int main(void)
{
     struct node    n1;
     struct node    *ptr;

     n1.data = 15;
     n1.nextptr = NULL;

     ptr = malloc(sizeof(struct node));
     ptr->data = 30;
     ptr->nextptr = NULL;

     n1.nextptr = ptr;

     free(ptr);
     return 0;
}
```

□ **End.**

Appendix A

Character Processing

Review Questions

- How characters are stored?
- What is the ASCII Table?
- What is the relationship between a `char` and an integer *value*?

The Use of `getchar()` and `putchar()`

- `getchar()`
  Gets a character from the input stream (keyboard).
- `putchar()`
  Puts a character to the output stream (screen).

Program char_1.c

```
#include <stdio.h>

int main(void)
{
     int  c;

     c = getchar();
     putchar(c);

     printf("\n");
     return 0;
}
```

```
c:\> char_1.exe
H
H
c:\>
```

Program char_2.c

```
#include <stdio.h>

int main(void)
{
     int  c;

     while ((c=getchar()) != EOF) {
          putchar(c);
          putchar(c);
     }
     return 0;
}
```

```
H
HH

Hello!
HHeellllloo!!

<-------------- ^Z is pressed.
```

NOTE

- In <stdio.h>,

```
#define EOF (-1)
```

- In the above program,

```
int c;
```

Program char_3.c

```
#include <stdio.h>

int main(void)
{
     int  c;

     while ((c=getchar()) != EOF)
          if ('a' <= c && c <= 'z')
               putchar (c + 'A' - 'a');
          else
               putchar(c);

     return(0);
}
```

```
m
M
Hello!
HELLO!
<-------------- ^Z is pressed.
```

The Macros in <ctype.h>

- Character processing
- Takes a single `int` and returns a single `int`.

Character macros	
Macro	Nonzero (true) is returned if
`isalpha(c)`	c is a letter
`isupper(c)`	c is an uppercase letter
`islower(c)`	c is a lowercase letter
`isdigit(c)`	c is a digit
`isalnum(c)`	c is a letter or digit
`isxdigit(c)`	c is a hexadecimal digit
`isspace(c)`	c is a white space character
`ispunct(c)`	c is a punctuation character
`isprint(c)`	c is a printable character
`isgraph(c)`	c is printable, but not a space
`iscntrl(c)`	c is a control character
`isascii(c)`	c is an ASCII code

Character macros and functions	
Function or macro	Effect
`toupper(c)`	changes `c` from lowercase to uppercase
`tolower(c)`	changes `c` from uppercase to lowercase
`toascii(c)`	changes `c` to ASCII code

Program char_4.c

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

int main(void)
{
     int  c;

     while ((c=getchar()) != EOF) {
          if (isalpha(c)) {
               printf("Lower: %c ", tolower(c));
               printf("Upper: %c\n", toupper(c));
          }
     }

     return(0);
}
```

```
m
Lower: m Upper: M
X
Lower: x Upper: X
4
=
<-------------- ^Z is pressed.
```

Program char_5.c

```
#include <stdio.h>

int main()
{
     int  number;
     int  reply;

     printf("Please enter a number: ");
     scanf("%d", &number);
     printf("Correct (y/n)? ");
     reply = getchar();

     if (reply=='Y' || reply=='y')
          printf("Correct\n");
     else
          printf("Incorrect\n");

     return 0;
}
```

```
c:\> char_5.exe
Please enter a number: 10
Correct (y/n)? Incorrect
c:\>
```

Program char_6.c

```
#include <stdio.h>

int main()
{
     int  number;
     int  reply;

     do {
          printf("Please enter a number: ");
          scanf("%d", &number);
          printf("Correct (y/n)? ");
          while (isspace(reply=getchar()));
     } while (reply!='y' && reply!='Y');

     printf("\nThe number you entered is %d.\n", number);

     return 0;
}
```

```
Please enter a number: 30
Correct (y/n)? n
Please enter a number: 10
Correct (y/n)?

n
Please enter a number: 20
Correct (y/n)? a
Please enter a number: 97
Correct (y/n)? y

The number you entered is 97.
```

□ **End.**

Appendix B

Recursion

What are Recursive Functions?

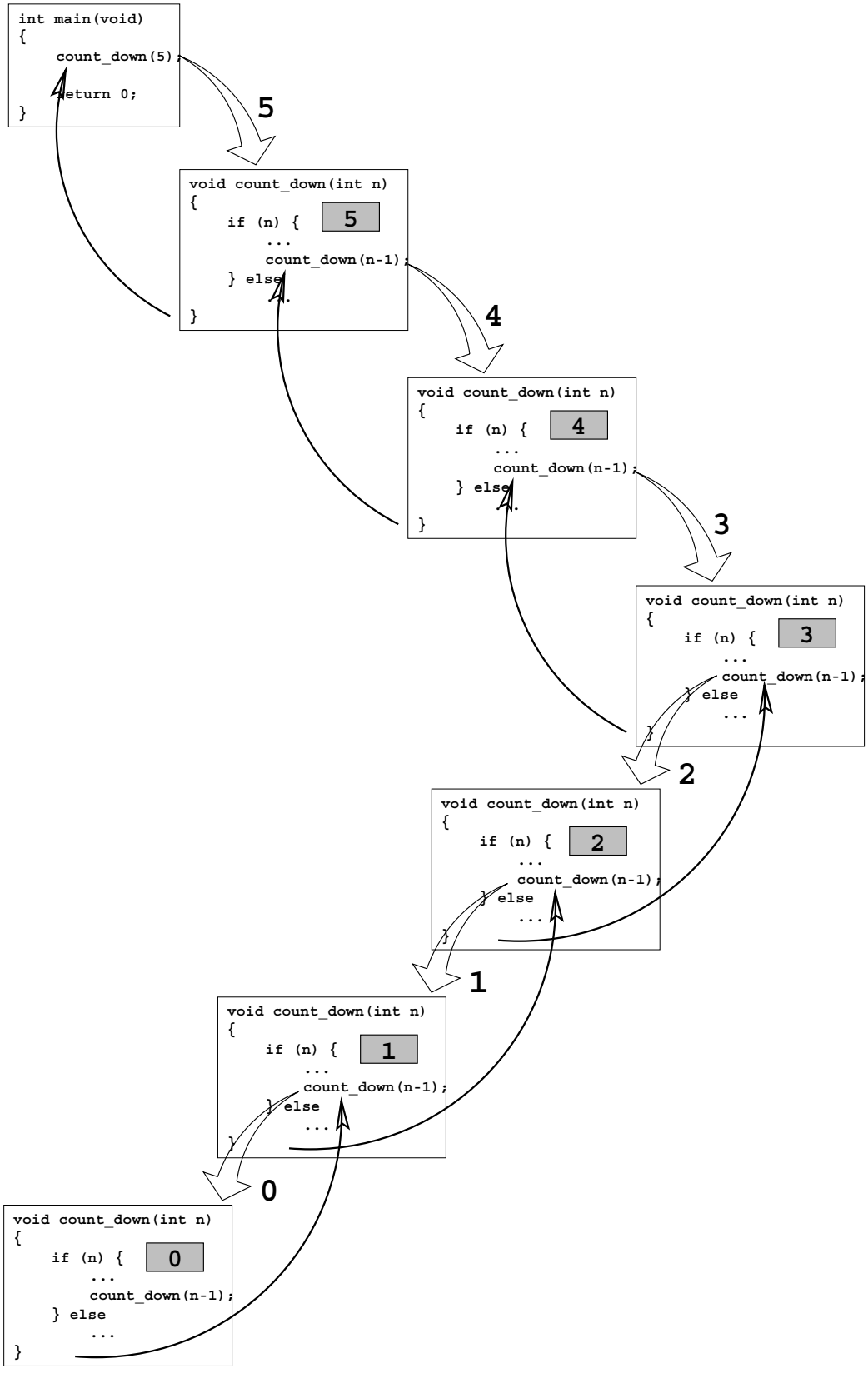
- A function which invokes itself, either directly or indirectly.
- An alternative to iteration.
- Naturally implements **divide-and-conquer** problem solving methodology.
  - Divide the problem into smaller pieces.
  - Tackle each sub-task either directly or indirectly.
  - Combine the solutions of the parts to form the solution of the whole.

B.1 Example — Count Down

Program recur_1.c

```
#include <stdio.h>

void   count_down(int);

int main(void)
{
   count_down(5);

   return 0;
}

void count_down(int n)
{
   if (n) {
      printf("%d !  ", n);
      count_down(n - 1);
   } else
      printf("\nGO!");
}
```

```
5 !  4 !  3 !  2 !  1 !
GO!
```


int main(void)
{
count_down(5);
return 0;
}
5
void count_down(int n)
{
if (n) {
5
...
count_down(n-1);
} else
...
}
4
void count_down(int n)
{
if (n) {
4
...
count_down(n-1);
} else
...
}
3
void count_down(int n)
{
if (n) {
3
...
count_down(n-1);
} else
...
}
2
void count_down(int n)
{
if (n) {
2
...
count_down(n-1);
} else
...
}
1
void count_down(int n)
{
if (n) {
1
...
count_down(n-1);
} else
...
}
0
void count_down(int n)
{
if (n) {
0
...
count_down(n-1);
} else
...
}

B.2 Example — Sum

Program recur_2.c

```
#include <stdio.h>

void   count_down(int);

int main(void)
{
     int  n;

     printf ("Enter a number? ");
     scanf("%d", &n);

     printf("Sum of %d = %d\n", n, sum(n));

     return 0;
}

int sum (int n)
{
     if (n <= 1)
          return n;
     else
          return (n + sum(n-1));
}
```

```
Enter a number? 4
Sum of 4 = 10
```

B.3 Example — Factorial

Program recur_3.c

```
#include <stdio.h>

long factorial(unsigned);

int main(void)
{
     unsigned    n;

     printf ("N=? ");
     scanf("%d", &n);

     printf("%u! = %ld\n", n, factorial(n));
     return 0;
}

long factorial(unsigned n)
{
     if (n <= 1)
          return 1L;
     else
          return (n * factorial(n-1));
}
```

```
N=? 5
5! = 120
```

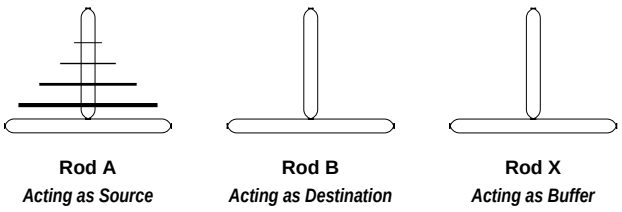
B.4 Example — The Tower of Hanoi

Program recur_4.c

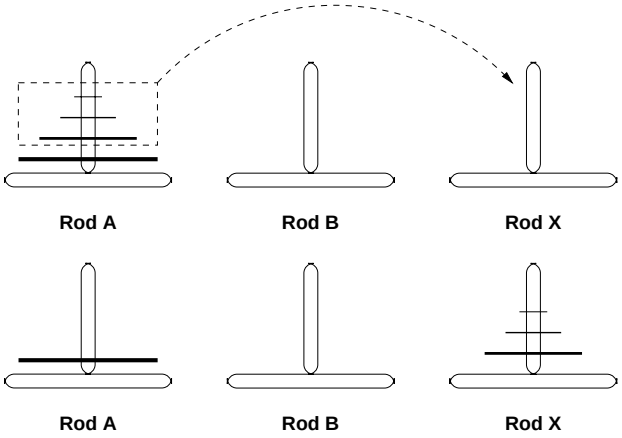
```
#include <stdio.h>

void hanoi(int, char, char, char);

int main(void)
{
     int  n;

     printf("How many discs? ");
     scanf("%d", &n);
     hanoi(n, 'A', 'B', 'X');
     return 0;
}

void hanoi(int n, char source, char dest, char tmp)
{
     if (n > 0) {
          hanoi(n-1, source, tmp, dest);
          printf("rod %c to rod %c\n", source, dest);
          hanoi(n-1, tmp, dest, source);
     }
}
```

```
How many discs? 3
rod A to rod B
rod A to rod X
rod B to rod X
rod A to rod B
rod X to rod A
rod X to rod B
rod A to rod B
```

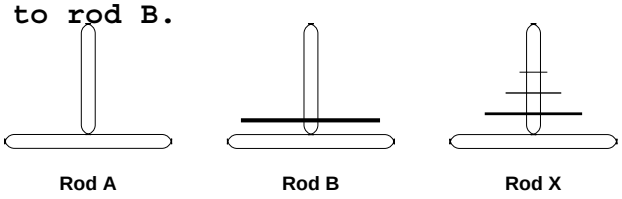
```
main()
{
  ...
  hanoi (4, 'A', 'B', 'X');
}
```



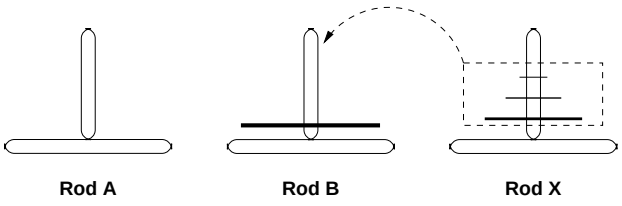

```
hanoi (n==4, source==A, dest==B, tmp==X) /* not in C syntax */
{

   if (n > 0) {
     hanoi (3, A, X, B);
```



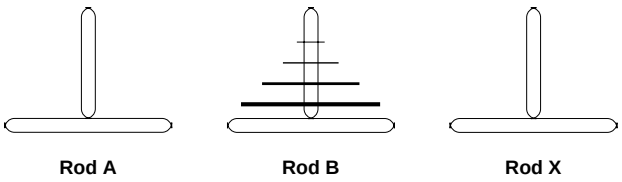

```
     Put a disc from rod A to rod B.
```




```
     hanoi (3, X, B, A);
```



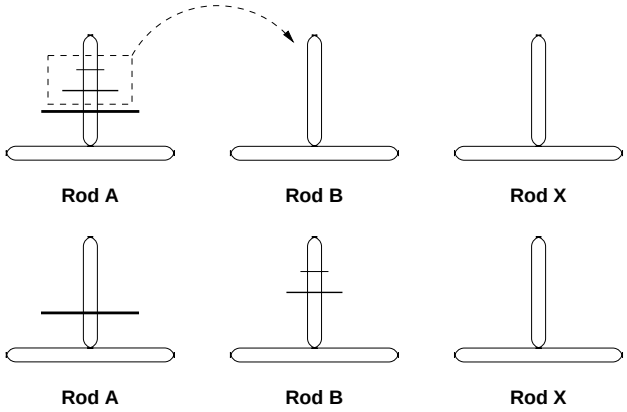

```
   }
```



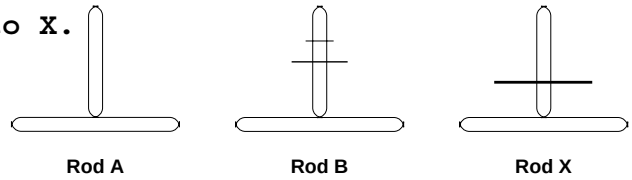

```
}
```

Current Objective: Move 3 discs from Rod A to Rod X using Rod B as buffer.

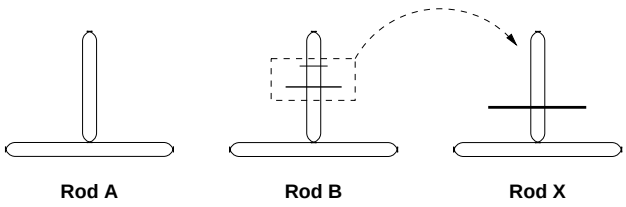
```
hanoi (n==3, source==A, dest==X, tmp==B)
{
   if (n > 0) {
      hanoi (2, A, B, X);
```

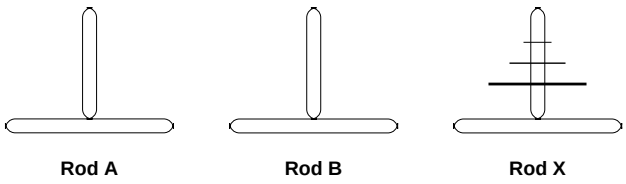



```
      Put a disc from rod A to X.
```




```
      hanoi (2, B, X, A);
```

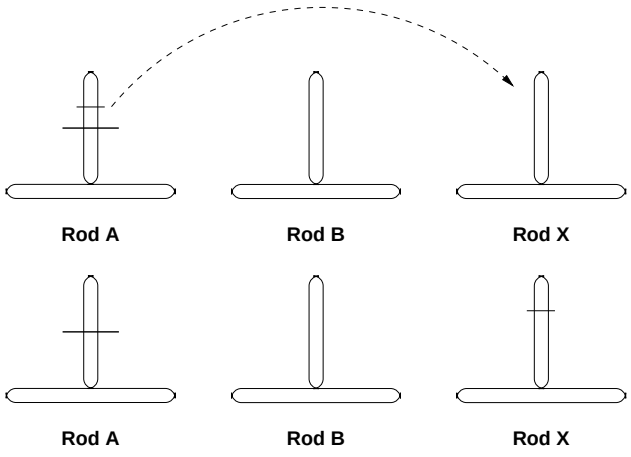




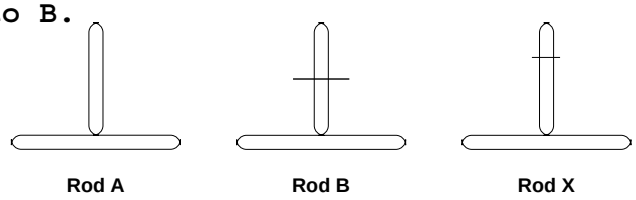

```
   }
}
```

Current Objective: Move 2 discs from Rod A to Rod B using Rod X as buffer.

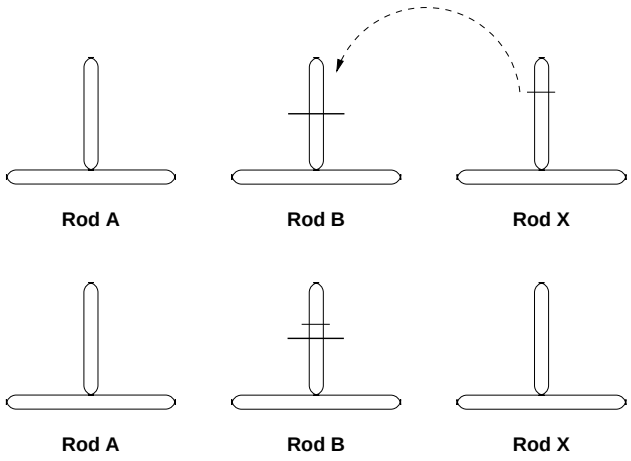
```
hanoi (n==2, source==A, dest==B, tmp==X)
{
   if (n > 0) {
      hanoi (1, A, X, B);
```




```
      Put a disc from rod A to B.
```



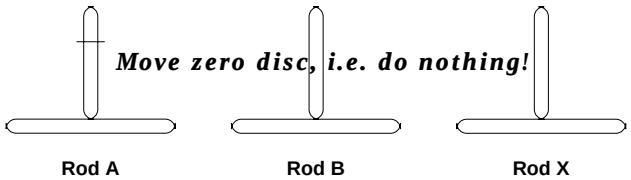

```
      hanoi (1, X, B, A);
```

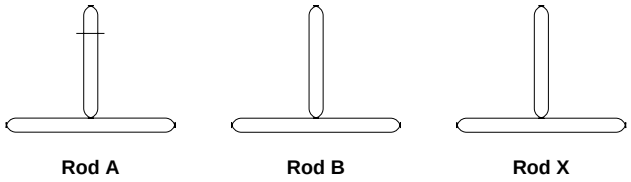



```
   }
}
```

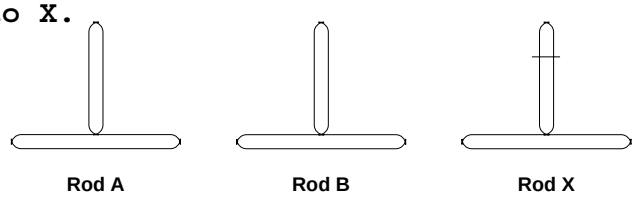
Current Objective: Move 1 discs from Rod A to Rod X using Rod B as buffer.

```
hanoi (n==1, source==A, dest==X, tmp==B)
{
   if (n > 0) {
      hanoi (0, A, B, X);
```

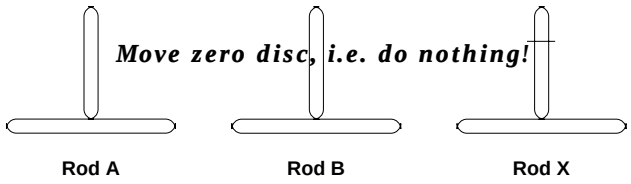


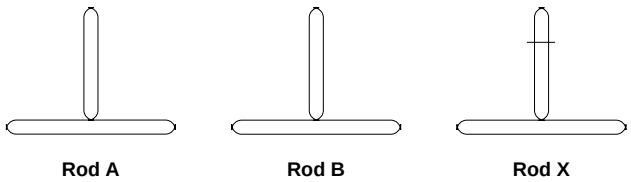



```
      Put a disc from rod A to X.
```




```
      hanoi (0, B, X, A);
```






```
   }
}
```

Summary

Recursive functions contain the following elements:

- Distinct cases governed by a selection structure.
- One or more of the cases have a simple, non-recursive solution. These are the **terminating cases**.
- Recursion is used to reduce the other cases to cases that are closer to the termination cases.
- Computation recurses until the terminating cases are reached and answers of the parts are returned and combined to form the answer of the whole.

□ **End.**

Appendix C

User Defined Types — `enum` and `typedef`

Overview

- `enum` allows programmers to define a new data type for a finit set of elements.
- `typedef` allows programmers to give an *alias* to an existing data type.
- Both attempts to improve program readability.

C.1 Enumeration Type — enum

What are enum's?

- User defined types that allow programmers to name a finite set of elements.
- Example,

```
enum day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

- "enum day" now becomes a new data type.
- SUN, MON, ..., SAT are called *enumerators*.

```
#include <stdio.h>

int main(void)
{
     enum day {SUN, MON, TUE, WED, THU, FRI, SAT};
     enum day today;

     ...
     today = SUN;
     ...
     if (today == SAT || today == SUN)
          printf("On vacation!");
     ...

     return 0;
}
```

Program enum_1.c

```
#include <stdio.h>

enum day {SUN, MON, TUE, WED, THU, FRI, SAT};
enum day find_next_day(enum day);

int main(void)
{
     enum day     next_day;

     next_day = find_next_day(MON);
     return 0;
}

enum day find_next_day(enum day d)
{
     enum day     next_day;

     switch (d) {
        case SUN:
             next_day=MON;
             break;
        case MON:
             next_day=TUE;
             break;
        ...
        case SAT:
             next_day=SUN;
             break;
     }
     return next_day;
}
```

Program enum_2.c

```
#include <stdio.h>

enum menu {SALAD, WINGS, FRIES, BURGER};

int main(void)
{
     enum menu choice;
     int reply;

     do {
          printf("1. Salad\n");
          printf("2. Chicken wings\n");
          printf("3. French fries\n");
          printf("4. Burger\n\n");
          printf("\nPlease select: ");
          while (isspace(reply = getchar()));
     } while (reply<'1' || reply>'4');

     switch (reply) {
        case '1':
          printf("Salad = 25.00\n");
          choice = SALAD;
          break;
        case '2':
          printf("Wings = 15.00\n");
          choice = WINGS;
          break;
        case '3':
          printf("Fries = 10.50\n");
          choice = FRIES;
          break;
```

```
        case '4':
          printf("Burger = 9.80\n");
          choice = BURGER;
          break;
        default:
          printf("Invalid choice!\n");
    }

    return 0;
}
```

Enumerators as Integer Constants

- Enumerators are in fact handled by the compiler as integer constants.
- Given

  ```
  enum menu {SALAD, WINGS, FRIES, BURGER};
  ```

  - The 1st enumerator (i.e. `SALAD`) is given the value 0.
  - The 2nd enumerator (i.e. `WINGS`) is given the value 1.
  - etc.
- We can make use of this property in our programs.

Program enum_3.c

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

enum prs {PAPER, ROCK, SCISSOR};
enum outcome {WIN, LOSE, TIE};

/* function prototype */
enum outcome compare (enum prs, enum prs);
enum prs player_choice();

int main(void)
{
     enum prs  player, computer;
     enum outcome   result;

     /* generate computer choice */
     srand(time(NULL));
     computer = (enum prs) rand() % 3;

     /* get user's choice */
     player = player_choice();

     /* who wins? */
     result = compare(player, computer);

     if (result==WIN)
          printf("You win!\n");
     else if (result==LOSE)
          printf("You lose!\n");
     else
```

```
          printf("Tie!\n");

     return 0;
}

enum prs player_choice()
{
     enum prs  player;
     int       ans;

     do {
          printf ("Your choice (p/r/s)? ");
          while (isspace(ans=getchar()));
          ans = toupper(ans);
     } while (ans!='P' && ans!='R' && ans!='S');

     switch (ans) {
        case 'P':
          player = PAPER;
          break;
        case 'R':
          player = ROCK;
          break;
        case 'S':
          player = SCISSOR;
     }

     return player;
}

enum outcome compare (enum prs player, enum prs computer)
{
     enum outcome   result;
```

```
     if (player == computer)
          result = TIE;
     else {
          switch (player) {
             case PAPER:
                 result = (computer==ROCK) ? WIN : LOSE;
                 break;
             case ROCK:
                 result = (computer==SCISSOR) ? WIN : LOSE;
                 break;
             case SCISSOR:
                 result = (computer==PAPER) ? WIN : LOSE;
          }
     }
     return result;
}
```

C.2 Data Type Definition — typedef

What are typedef's

- Allows programmers to give a new name to a type.
- Example,

```
typedef int color_t;
```

- Identifier color_t is now synonymous with int.

```
#include <stdio.h>

typedef int color_t;

int main()
{
     color_t    red, blue, green;

     red = 15;
     ...
}
```

Further Example

- Simplifies declarations involving lengthy user-defined types.
- Example,

```
typedef unsigned long int    time_t;
```

- Example,

```
enum day {SUN,MON,TUE,WED,THU,FRI,SAT};
typedef enum day day_t;
```

- Can be simplified as

```
typedef enum {SUN,MON,TUE,WED,THU,FRI,SAT} day_t;
```

Exercise

Try to `typedef` the `enum prs` type and the `enum outcome` type in the "Paper, Rock, and Scissor" program and rewrite the whole program.

Program enum_4.c

```
#include <stdio.h>

/*
enum DAY {SUN, MON, TUE, WED, THU, FRI, SAT};
typedef enum DAY   day_t;
*/

typedef enum {SUN, MON, TUE, WED, THU, FRI, SAT} day_t;

day_t find_next_day_t(day_t);

int main(void)
{
        day_t     next_day;

        next_day = find_next_day(MON);

        return 0;
}

day_t find_next_day(day_t d)
{
        day_t     next_day;

        switch (d) {
           case SUN:
                next_day=MON;
                break;
           case MON:
                next_day=TUE;
                break;
           ...
```

```
            case FRI:
                 next_day=SAT;
                 break;
            case SAT:
                 next_day=SUN;
                 break;
        }

        return next_day;
}
```

□ **End.**

Appendix D

Strings and Pointers

Overview

- A string is a sequence of characters enclosed by a pair of braces { } .
- A string is stored as a one-dimensional array of type `char`.
- The standard library provides many useful string handling functions.

D.1 Basic String Concepts

Program str_1.c

```
#include <stdio.h>

int main(void)
{
     char name[15]="C Program";

     printf("Hello, %s\n", name);
     return 0;
}
```

```
Hello, C Program
```

Address	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414
`name[i]`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Character	C		P	r	o	g	r	a	m	\0	?	?	?	?	?
Note:	characters in the string									NULL	Unused				

NOTE

- `name` is a character array.
- `NULL` (`'\0'`) signifies the end of the string in the array.
- `NULL` is predefined as `'\0'`.

Initialization of Strings

- As a literal string with the size specified explicitly:

```
char name[15]="C Program";
```

  The size of the `char` array is 15 with `name[9]` automatically initialized to `'\0'`.

- As a literal string without specifying the array size explicitly:

```
char name[]="C Program";
```

  The size of the `char` array is the length of the string *plus one*, that is, 10. `name[9]` is automatically initialized to `'\0'`.

Program str_2.c

```
#include <stdio.h>
#define MAXLEN 20

int main(void)
{
     char name[MAXLEN+1];
     int  i, len;

     printf ("Hi! What's your name? ");
     scanf("%s", name);
     printf ("\n");

     len=0;
     while (name[len] != '\0')
          len++;

     printf ("Length                = %d\n", len);
     printf ("Your name reversed is = ");
     for (i=len-1; i>=0; i--)
          printf("%c", name[i]);

     printf ("\n");
     return 0;
}
```

```
Hi! What's your name? C-Program

Length                 = 9
Your name reversed is = margorP-C
```

D.2 String Handling Functions

NOTE

- The standard library contains many useful string handling functions.
- They all require that strings passed as arguments be **null-terminated**.
- Their function prototypes are given in the header file `<string.h>`.

D.2.1 Checking String Length — `strlen()`

Description

```
unsigned int strlen(char s[]);
```

Determines the length of string `s`. The number of characters preceding the termination `NULL` character is returned.

Program str_3.c

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

int main(void)
{
     char    s[]="This is a string!";
     int     len;

     len = strlen(s);

     printf("%d\n", len);
     return 0;
}
```

17

D.2.2 Displaying Strings — puts()

Description

```
int puts(char s[]);
```

Print the string s followed by a newline character.

Program str_4.c

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

int main(void)
{
     char    s[]="This is a string!";

     puts(s);
     return 0;
}
```

```
This is a string!
```

printf()

You should know how to use printf() for displaying strings.

```
printf("%s", string);
```

D.2.3 Getting Strings — scanf()

Program str_5.c

```
#include <stdio.h>

int main(void)
{
     char    s1[20], s2[20];

     scanf("%s", s1);
     printf("s1=%s\n", s1);

     printf("\n");
     scanf("%s", s2);
     printf("s2=%s\n", s2);

     return 0;
}
```

```
this-is-first-string.
s1=this-is-first-string.

this is second string.
s2=this
```

NOTE

- When inputing a string using `scanf()`, the argument(s) does not need to be preceded by '&'. Why?
- White spaces (such as space) are regarded as delimiters by `scanf()`.

D.2.4 Getting Strings — gets()

Description

```
char *gets(char s[]);
```

Input characters from the standard input into the array s until a newline or end-of-file character is encountered. A terminating NULL character is appended to the array. The array s is returned.

Program str_6.c

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

int main(void)
{
     char    name[20];

     printf("Your name? ");
     printf("Hello %s!\n", gets(name));

     return 0;
}
```

```
Your name? C Program
Hello C Program!
```

D.2.5 Copying Strings — `strcpy()`

Description

```
char *strcpy(char s1[], char s2[]);
```

Copies the string `s2` into the array `s1`. `s1` is null-terminated afterwards. The array `s1` is returned.

Program str_7.c

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

int main(void)
{
     char    s1[20];
     char    s2[20]="Hello world!";

     strcpy (s1, s2);
     printf("%s\n", s1);

     return 0;
}
```

```
Hello world!
```

D.2.6 Copying Strings — `strncpy()`

Description

`char *strncpy(char s1[], char s2[], unsigned n);`

Copies at most `n` characters of the string `s2` into the array `s1`. `s2` is truncated if necessary. `s1` will not be null-terminated if the length of `s2` equals `n` or more. The array `s1` is returned.

Program str_8.c

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

int main(void)
{
     char    s_short[6];
     char    s2[20]="Hello world!";
     char    s_long[30];

     strncpy(s_short, s2, 5);
     s_short[5]='\0';
     printf("%s\n", s_short);

     printf("%s\n", strncpy(s_long, s2, 29));
     return 0;
}
```

```
Hello
Hello world!
```

D.2.7 Concatenating Strings — strcat()

Description

```
char *strcat(char s1[], char s2[]);
```

Appends the string s2 to the array s1. The first character of s2 overwrites the terminating NULL character of s1. The value of s1 is returned.

Program str_9.c

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

int main(void)
{
     char    s1[20]="Happy ";
     char    s2[]="New Year";

     printf ("s1=%s, s2=%s\n", s1, s2);
     printf("s1+s2=%s\n", strcat(s1, s2));
     return 0;
}
```

```
s1=Happy , s2=New Year
s1+s2=Happy New Year
```

D.2.8 Comparing Strings — `strcmp()`

Description

```
int strcmp(char s1[], char s2[]);
```

Compares the string `s1` to the string `s2`. The function returns 0, less than 0, or greater than 0 if `s1` is equal to, less than, or greater than `s2`, respectively.

Program str_10.c

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

int main(void)
{
     char    myname[]="C Program";
     char    yourname[20];
     short   code;

     printf("Your name? ");
     gets(yourname);

     code = strcmp(yourname, myname);
     if (code > 0)
          printf("You are bigger!\n");
     else if (code < 0)
          printf("You are smaller!\n");
     else
          printf("Same!\n");

     return 0;
}
```

```
Your name? Au Yeung Siu Ming
You are smaller!
```

```
Your name? Wong Tai Man
You are bigger!
```

```
Your name? C Program
Same!
```

D.2.9 String Conversion Functions — `atoi()`, `atof()`, `atol()`

Description
The functions examines the string `s` to produce a numerical value of type `int`, `double`, and `long`, respectively.

Program str_11.c

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

int main(void)
{
     char    s[20];

     printf("Input a number: ");
     scanf("%s", s);
     printf("String:  %s\n", s);
     printf("Integer: %d\n", atoi(s));
     printf("Double:  %.2f\n", atof(s));
     printf("Long:    %ld\n", atol(s));
     return 0;
}
```

```
Input a number: 1385.97
String:  1385.97
Integer: 1385
Double:  1385.97
Long:    1385
```

Remark

- There are many other string handling functions in the string library and standard library.
- Check the documentation to see if an existing function suits your need before attempting to write your own string handling functions.
- Remember, browsing through reference manuals is the essential technique that every C programmer should equip with.

D.3 Strings and Pointers

Program str_12.c

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

int main(void)
{
     char    message[]="Welcome to C Programming!";
     char    *cptr;

     cptr = message + 11;
     printf("%s\n", cptr);

     return 0;
}
```

```
C Programming!
```

□ **End.**

Appendix E

File Manipulations

Overview

- General input/output for files.
- Files must be opened before use.
- Files must be closed after use.
- Basic file operations: *reading*, *writing*, and *appending*.

Program file_1.c

```
#include <stdio.h>

int main(void)
{
        FILE    *fptr;
        char    name[30];

        fptr=fopen("data.txt", "r");  /* open the file */

        fscanf(fptr, "%s", name);     /* process the file */
        printf("Name = %s\n", name);

        fclose(fptr);                 /* close the file */

        return 0;
}
```

FILE: `data.txt`

```
Programming
```

```
Name = Programming
```

DON'T ATTEMPT TO UNDERSTAND IT NOW!!!

E.1 Opening Files

`fopen()`

Function Prototype:

```
#include <stdio.h>

FILE *fopen(char *filename, char *mode);
```

Description:

- Opens the file named by `filename` in the way specified by the character string `mode`.

Return Values:

- `fopen()` returns a non-`NULL` pointer (of type `FILE *`, usually referred to as a *stream*) if the file can be opened successfully.
- Otherwise, a `NULL` pointer is returned.

Mode	Description
`"r"`	Open for reading
`"w"`	Truncate to zero length or create for writing
`"a"`	Append; open for writing at end-of-file, or create for writing
`"r+"`	Open for update (reading and writing)
`"w+"`	Truncate or create for update
`"a+"`	Append; open or create for update at end-of-file

Example on `fopen()`

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

int main(void)
{
     FILE *fp;

     fp=fopen("data.txt", "r");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     ...
}
```

E.2 Reading From Files

`fscanf()`

Function Prototype:

```
#include <stdio.h>

int fscanf(FILE *fp, format_specifier, arg1, ...);
```

Description:

- Similar to the use of `scanf()`, except that the data comes from the stream `fp` (instead of from the standard input).
- Advances the "file pointer" associated with `fp`. (*To avoid confusion, we use the term "file position indicator".*)

Return Values:

- Returns the number of arguments read from the stream `fp`.
- Returns `EOF` when the file pointer reaches the end-of-file.

Example on `fscanf()`

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

int main(void)
{
     FILE *fp;
     int x;

     fp=fopen("data.txt", "r");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     fscanf(fp, "%d", &x);
     ...
}
```

`fgetc()`

Function Prototype:

```
#include <stdio.h>

int fgetc(FILE *fp);
```

Description:

- Returns the next character from the stream `fp`.
- Advances the "file pointer" associated with `fp` one character ahead.

Return Values:

- Returns `EOF` at end-of-file or upon an error.

Example on `fgetc()`

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

int main(void)
{
     FILE *fp;
     int c;

     fp=fopen("data.txt", "r");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     c=fgetc(fp);
     if (c==EOF) {
          ...  /* error processing */
     }
     ...
}
```

fgets()

Function Prototype:

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *fp);
```

Description:

- `fgets()` reads characters from the stream into the array pointed to by `s`, until n-1 characters are read, or a newline character is read and transferred to `s`, or an end-of-file condition is encountered. The string is then terminated with a `NULL` character.

Return Values:

- Upon successful completion, `s` is returned.
- If end-of-file is encountered and no characters have been read, no characters are transferred to `s` and a `NULL` pointer is returned.
- If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a `NULL` pointer is returned.

E.3 Writing to Files

fprintf()

Function Prototype:

```
#include <stdio.h>

int fprintf(FILE *fp, format_specifier, arg1,...);
```

Description:

- Similar to the use of `printf()`, except that the data is written to the stream `fp` (instead of to the standard output).
- Advances the "file pointer" associated with `fp`.

Return Values:

- Returns the number of characters written to the stream `fp`.
- Returns `EOF` on error.

Example on `fprintf()`

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

int main(void)
{
     FILE *fp;

     fp=fopen("data.txt", "w");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     fprintf(fp, "%s\n", "Hello!");
     ...
}
```

`fputc()`

Function Prototype:

```
#include <stdio.h>

int fputc(int c, FILE *fp);
```

Description:

- Writes a single character `c` to the stream `fp`.
- Advances the "file pointer" associated with `fp` one character ahead.

Return Values:

- Returns the value written (i.e. `c`).
- Returns `EOF` on error.

Example on `fputc()`

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

int main(void)
{
     FILE *fp;

     fp=fopen("data.txt", "w");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     fputc('A', fp);
     ...
}
```

E.4 Closing Files

`fclose()`

Function Prototype:

```
#include <stdio.h>

int fclose(FILE *fp);
```

Description:

- Causes the buffered data associated with the stream `fp` to be written out to disk and the corresponding file to be closed.

Return Values:

- Upon successful completion, `fclose()` returns a value of zero. Otherwise `EOF` is returned.

Example on `fclose()`

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

int main(void)
{
     FILE *fp;

     fp=fopen("data.txt", "r");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     ...  /* processing the file data */

     if (fclose(fp)==EOF) {
          printf("Cannot close file.\n");
          exit(0);
     }

     return 0;
}
```

E.5 Inquiring End-Of-File

`feof()`

Function Prototype:

```
#include <stdio.h>

int feof(FILE *fp);
```

Description/Return Values:

- Returns non-zero when `EOF` has *previously been detected* for the stream `fp`, otherwise zero.

Example on `feof()`

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

int main(void)
{
     FILE *fp;
     int c;

     fp=fopen("data.txt", "r");
     if (fp == NULL) {
          printf("Cannot open file.\n");
          exit(0);
     }

     ...

     if (!feof(fp))
          c = fgetc(fp);

     ...

}
```

E.6 Examples

E.6.1 Displays a file

Program file_2.c

```
#include <stdio.h>

int main(void)
{
     FILE    *fp;
     char    filename[13];  /* DOS filename format 8.3 */
     int     inchar;

     printf("Name of file? ");
     gets(filename);

     if ((fp=fopen(filename, "r"))==NULL) {
          printf("Error: Cannot open %s.", filename);
          return -1;
     }

     inchar = fgetc(fp);
     while (inchar!=EOF) {
          putchar(inchar);
          inchar = fgetc(fp);
     }

     fclose(fp);
     return 0;
}
```

E.6.2 File Backup

Program file_3.c

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

#define NAMELEN  12        /* DOS filename format 8.3 */

int main(void)
{
     FILE *fin, *fout;
     char source[NAMELEN+1], backup[NAMELEN+1];
     int  c;

     printf("Source file: ");
     scanf("%s", source);

     if ((fin=fopen(source, "r"))==NULL) {
          printf("Error: Cannot open %s.\n", source);
          return -1;
     }

     printf("Backup file: ");
     scanf("%s", backup);

     if ((fout=fopen(backup, "w"))==NULL) {
          printf("Error: Cannot open %s.\n", backup);
          fclose(fin);
          return -1;
     }

     while (!feof(fin)) {
          c = fgetc(fin);
```

```
          if (c!=EOF)
               fputc(c, fout);
     }

     fclose(fin);
     fclose(fout);

     return 0;
}
```

E.6.3 Data File Processing

Program file_4.c

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

int main(void)
{
    FILE      *fin;
    int       code, quantity;
    char      name[30];
    float     price;

    if ((fin=fopen("client.dat", "r"))==NULL) {
        printf("Error: Cannot open %s.\n", "clients.dat");
        exit (0);
    }

    printf("%-7s%-18s%12s%10s%10s\n",
        "Code", "Name", "Unit Price", "Quantity", "Amount");
    fscanf(fin, "%d%s%f%d", &code, name, &price, &quantity);

    while (!feof(fin)) {
        printf("%-7d%-18s%12.2f%10d%10.2f\n",
            code, name, price, quantity, price*quantity);
        fscanf(fin, "%d%s%f%d", &code, name, &price, &quantity);
    }
    fclose(fin);

    return 0;
}
```

FILE: `client.dat`

```
100 Eraser              4.90   10
200 Box-File           25.00   5
300 Correction-Fluid   13.50   15
400 Time-Magazine      34.00   2
500 A4-Paper           28.50   5
```

Code	Name	Unit Price	Quantity	Amount
100	Eraser	4.90	10	49.00
200	Box-File	25.00	5	125.00
300	Correction-Fluid	13.50	15	202.50
400	Time-Magazine	34.00	2	68.00
500	A4-Paper	28.50	5	142.50

□ **End.**