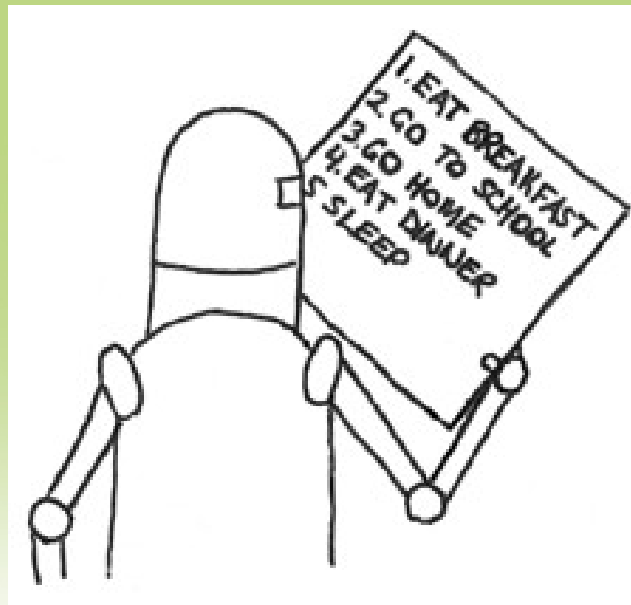


Chapter 1.1

Introduction to C Programming

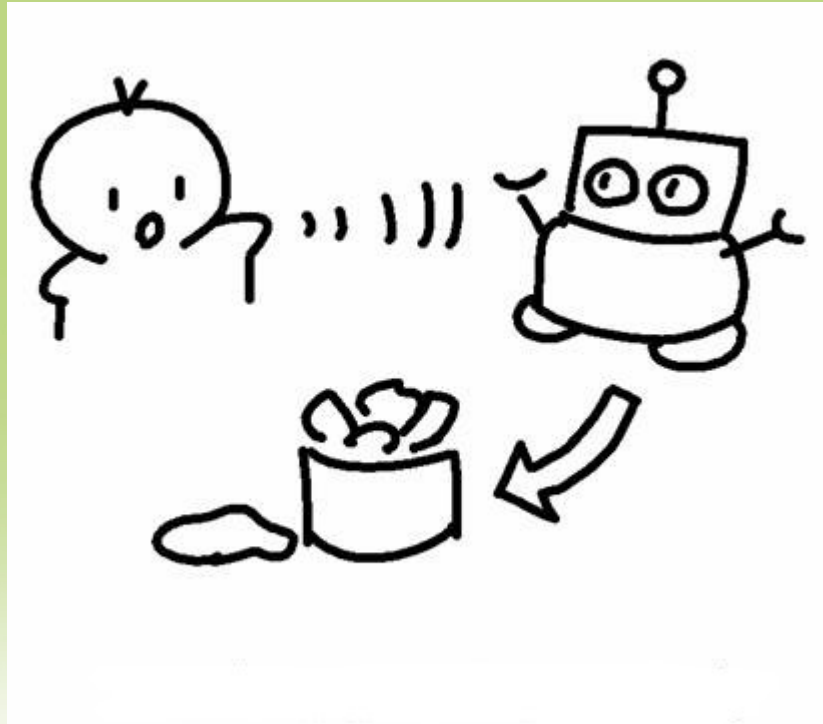
What is a program?

- A list of instructions.
- Written in a programming language (c, c++, java).
- Determines the behavior of a machine (computer, robot).



Programming Language

- A machine language that has its own set of grammatical rules.
- It is used to instruct a machine to perform specific tasks.



History

- C was developed in 1972 by Dennis Ritchie at Bell Laboratories.
- Based on a programming language called B (which is derived from BCPL language).
- C was designed and developed for the development of UNIX operating system.
- C has been widely used to develop many type of application software from a basic to operating system.

History

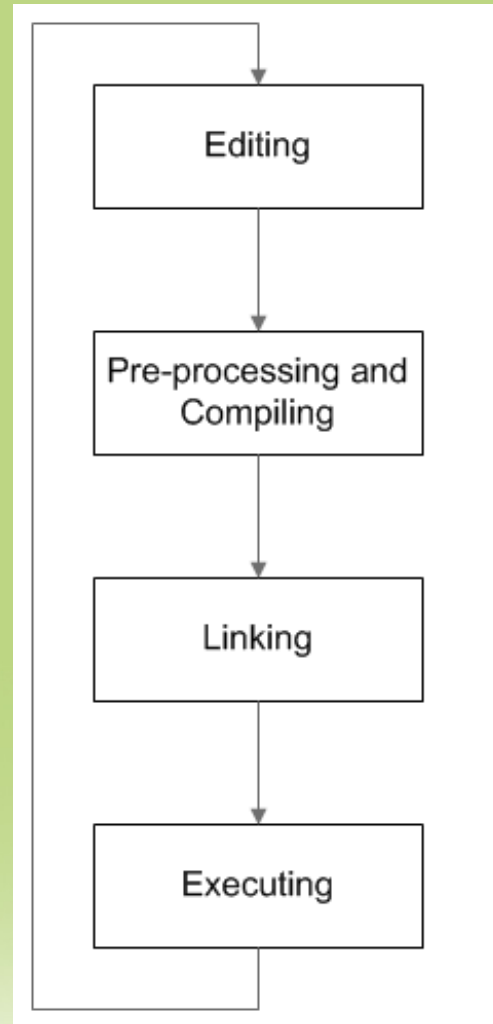
1963 CPL (Combined Programming Language)
Christopher Strachey
Uni. Cambridge & Uni. London
Complex thus not popular

1966 BCPL
Martin Richards
Powerful and Portable
One data type

1969 B
Ken Thompson & Dennis Ritchie
Bell Labs
One data type

1972 C
Dennis Ritchie at Bell Laboratories
Based on a programming language called B
Designed and developed for the development of UNIX operating system

Development stages of C program



```
main()
{
    printf("Hello World\n");
}
```

"Fantasy"



"C" Compiler



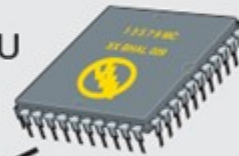
Linker



"Reality"

Memory

CPU



IO Interface



Editing

- Process of writing the C source code.
- Using editor program (i.e. vi and emacs for Linux, Notepad for Windows)
- Software package provides programming environment
 - Integrated editor
 - Managing and maintaining source code
 - i.e. C++ Builder, Code::Blocks, Dev-C++, Microsoft Visual Studio

Pre-processing and Compiling

- Pre-processing – performs directives such as inclusion of header files and substitution.
- Compiling – process of translating the source code (programming language) to machine code (machine language).

Linking

- Process of combining generated object files.
- Add required codes from C standard library.
- Produce an executable file.

Executing

- Run the program to check whether it produce the desired output.
- Testing stage where you check the output of your program.
- The program does not produces the suppose output, then you have made logic (semantic) error.

C Program Layout

- C program consists of two sections:
 - Pre-processor directives
 - Main function

pre-processor directives

main ()

{

statements

}

Pre-processor section



Main function section

Pre-processor Directives

- Specify what compiler should do before compilation.
- Include header files.
- Header file keep information about the library functions (printf, scanf).
- Pre-processor directives begins with the # symbol.

Main Function

- The program starts its execution.
- Define the program definition.

An Example of C Program

```
/* An example of c program */  
#include<stdio.h>  
  
/* main function definition */  
int main()  
{  
    printf("C program example\n");  
    return 0;  
}
```

Sample output:

C program example

An Example of C Program

```
/* An example of c program */
```

- A comment, not actually part of the program.
- Provide information on what the program does.
- Begin with `/*` and end with `*/` or written as `/**`.
- Good habit to include comments.
- Keep inform what are the code actually doing.

An Example of C Program

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

- Important line in C
- Symbol # indicates that this is a pre-processor directive
- Include `stdio.h` header file in our program
- `stdio.h` contains information on input/output routines.
- We are using the `printf()` function from `stdio.h`

An Example of C Program

```
int main()
```

- Defines the start of the function `main()` .
- Keyword `int` signifies main function returns an integer value.
- The `()` is where we specify information to be transferred to function `main()` .

An Example of C Program

```
{  
    ...  
    ...  
}
```

- The portion that begins with { and ends with } is called block or body.
- Within the block is where we write the statements.

An Example of C Program

```
printf("C program example\n");
```

- Function `printf()` instructs the computer to display characters or a string enclosed by the double quotes on the monitor screen.
- A string is a combination of numerous characters.

An Example of C Program

```
return 0;
```

- Function `main()` is defined to return a value of integer type.
- `return 0` is needed to indicate that the program has terminated successfully.

Flow Chart

- Sequence flow

Flow Chart

- Looping flow

Flow Chart

- Combination
flow

Exercise

Draw a flowchart to solve below problem:

Prepare cupcakes. Ingredients are flour, yeast, and sugar. Make sure you taste the dough before baking process.

Find solution to calculate the average marks for 100 of student. Then, display the average value.



Chapter 1.2

Variables, Keywords and Operators

Variables

- Every variable has **name, type and value**.
- **Variable name / Identifier :**
 - Must start with character or underscore (_), cannot start with number or symbol and not a keyword .
 - Must not include any space and it is letter case sensitive.
 - Eg: `stud1, Stud1, stud_name, _student` (valid)
 `@stud, stud name, 1stud` (invalid)
- **Variable type :**
 - Based on the data type – integer, floating point numbers and characters.
 - The value can be change any time.
 - Whenever a new value is placed into a variable, it replaces the previous value.

Variables

- Variable must be **declare / initialize** before it can be used in the next line of statements.
- **Variable declaration :**
 - To declare a variable with a name and suitable data type.
 - Eg:

```
int student; int staff;  
double x, y;
```
- **Variable initialization :**
 - To declare a variable and assign a certain value to the variable.
 - Eg:

```
int student=5; int staff=1;  
double x=1.12;  
double y=9.999;
```

Data Types

- **Integer:**

- Whole number (decimal).
- Positive, negative or zero : 25, 0, -9.
- Variable declaration: `int x;`

- **Floating point numbers:**

- double / float
- Variable declaration: `double y;`
`float z;`

- **Characters:**

- A character is a single letter, digit, symbol or space. Eg:
`'m', 'B', '#', '3', '=', '@', ' ' .`
- A string is series of characters .
Eg: `"student" , "F#", "$35",`
`"class_93", "ALI BABA".`

Keywords

- Special words reserved for C.
- Cannot be used as variable names.
- Eg: data types (`int, void, const, static`)
instruction statements (`if, for, case, break, return, include`)

Constants

- **Constant name** is declared just like variable name.
- **Constant value** cannot be changed after its declaration.
- 2 ways for **constant declaration** :
 - `const double pi = 3.147; or`
 - `#define pi 3.147`

Assignment Operator

- To assign a value to a variable.
- Is a binary operator (at least has two operands).
- Variable that receiving the value is on the left.

• Eg: `int num;` →

| |
|-----|
| num |
| |

 (in memory)

`num=3;` →

| |
|---|
| 3 |
|---|

- `num` is assigned to value of 3.
- Arithmetic operation → `c = a + b`
- Multiple assignment → `x = y = z = 7`
- Compound assignment → `+=, -=, *=, /=, %=`
→ `sum += 5`
`sum = sum+5`

Arithmetic Operators

| Operators | Symbol |
|----------------|--------|
| Addition | + |
| Substraction | - |
| Multiplication | * |
| Division | / |
| Modulus | % |

- **x % y** produces **remainder** when x / y .
- Eg: $5 / 3 = 1$, $5 \% 3 = 2$.

Relational Operators

| Operator | Symbol | |
|--------------------------|--------|--------------------------------------------|
| Less than | < | |
| Less than or equal to | <= | |
| Greater than | > | |
| Greater than or equal to | >= | |
| Equal to | == | Also known as Equality Operators |
| Not equal to | != | |

- Used to compare numerical values .

Logical Operators

Operators

NOT

AND

OR

Symbol

!

&&

||

Expression

! 1

! 0

Value

0

1

Expression

0 && 0

0 && 1

1 && 0

1 && 1

Value

0

0

0

1

Expression

0 || 0

0 || 1

1 || 0

1 || 1

Value

0

1

1

1

Increment and Decrement Operators

- Increment operator → **++**
→ add 1
- Decrement operator → **--**;
→ subtract 1
- These operators can be used as **prefix** (**++a/--a**) or **postfix** (**a++/a--**).
- Prefix → increase/decrease the operand **before** it is used.
- Postfix → increase/decrease the operand **after** it is used.

Increment and Decrement Operators

- **Example 1.2.1**

| | y | c | x | (in memory) |
|-----------------------------|----------|----------|----------|-------------|
| <code>int y, c, x;</code> | | | | |
| <code>y=5, c=1, x=3;</code> | 5 | 1 | 3 | |
| <code>c++;</code> | 5 | 1 | 3 | |
| <code>x+=2;</code> | 5 | 2 | 5 | |
| <code>--y;</code> | 4 | 2 | 5 | |
| <code>y=c++;</code> | 2 | 2 | 5 | |
| <code>y=++x;</code> | 6 | 3 | 6 | |

Operator Precedence

Highest Precedence

| | |
|--------------|-------|
| parenthesis, | () ! |
| logical op. | |

| | |
|----------------|-------|
| arithmetic op. | * / % |
|----------------|-------|

| | |
|----------------|-----|
| arithmetic op. | + - |
|----------------|-----|

| | |
|----------------|-----------|
| relational op. | < <= > >= |
|----------------|-----------|

| | |
|--------------|-------|
| equality op. | == != |
|--------------|-------|

| | |
|-------------|----|
| logical op. | && |
|-------------|----|

| | |
|-------------|--|
| logical op. | |
|-------------|--|

Lowest Precedence

Operator Precedence

- **Example 1.2.2**

$7 \% 3 + 4 * 2 - (1 + 12) / 4$

- **Example 1.2.3**

$x > 5 \ || \ y < 7 \ \&\& \ z \ != \ 2$

let $x=4$, $y=5$, $z=9$.

Operator Precedence

- **Example 1.2.2**

$$7 \% 3 + 4 * 2 - (1 + 12) / 4$$

$$= 7 \% 3 + 4 * 2 - 13 / 4$$

$$= (7 \% 3) + (4 * 2) - (13 / 4)$$

$$= 1 + 8 - 3$$

$$= 6$$

Operator Precedence

- **Example 1.2.3**

x > 5 || y < 7 && z != 2

let x=4 , y =5 , z=9 .

- x > 5 returns False (0)
- y < 7 returns True (1)
- z != 2 returns True (1)

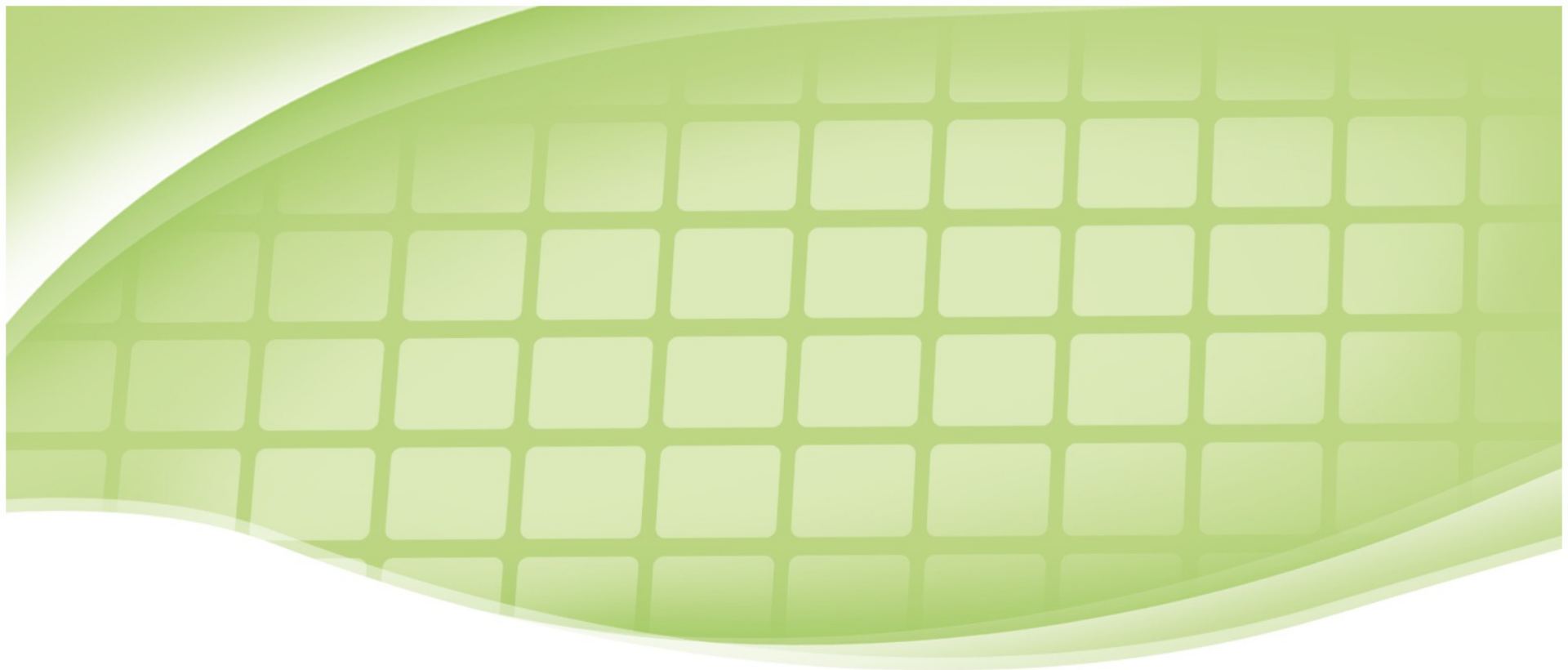
= (x > 5) || (y < 7) && (z != 2)

= 0 || 1 && 1

= 0 || (1 && 1)

= 0 || 1

= **1**



Chapter 2

Input Output Operation

Input Output Device

- Input devices are used to perform the input operation.
- Eg: keyboard, mouse, microphone, scanner.
- Output devices are used to perform the output operation.
- Eg: monitor, printer, speaker.

Input Output Operation

- Input operation in basic C programming is to get input data from the keyboard using `scanf` statement.
- While output operation is to display the data at the monitor screen using `printf` statement.

Input Output Operation



display the
data at the
monitor

get input data
from the
keyboard



Formatting Output with `printf`

- **`printf`**
 - Precise output formatting
 - Conversion specifications: flags, field widths, precisions, etc.
 - Can perform rounding, aligning columns, right/left justification, inserting literal characters, exponential format, hexadecimal format, and fixed width and precision
- Format
 - **`printf`**(*format-control-string*, *other-arguments*) ;
 - Format control string: describes output format
 - Other-arguments: correspond to each conversion specification in format-control-string
 - Each specification begins with a percent sign(%), ends with conversion specifier

Printing Integers

- Integer
 - Whole number (no decimal point): 25, 0, -9
 - Positive, negative, or zero
 - Only minus sign prints by default

| Conversion Specifier | Description |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| d | Display a signed decimal integer. |
| i | Display a signed decimal integer. (<i>Note:</i> The i and d specifiers are different when used with scanf .) |
| o | Display an unsigned octal integer. |
| u | Display an unsigned decimal integer. |
| x or X | Display an unsigned hexadecimal integer. X causes the digits 0-9 and the letters A-F to be displayed and x causes the digits 0-9 and a-f to be displayed. |
| h or l (letter l) | Place before any integer conversion specifier to indicate that a short or long integer is displayed respectively. Letters h and l are more precisely called <i>length modifiers</i> . |

Printing Floating-Point Numbers

- Floating Point Numbers
 - Have a decimal point (**33.5**)
 - **f** – print floating point with at least one digit to left of decimal

Printing Strings and Characters

- **c**
 - Prints **char** argument
 - Cannot be used to print the first character of a string
- **s**
 - Requires a pointer to **char** as an argument
 - Prints characters until **NULL** (' \0 ') encountered
 - Cannot print a **char** argument
- Remember
 - Single quotes for character constants (' **z** ')
 - Double quotes for strings "**z**" (which actually contains two characters, ' **z** ' and ' \0 ')

Printing with Field Widths and Precisions

- Field width
 - Size of field in which data is printed
 - If width larger than data, default right justified
 - If field width too small, increases to fit data
 - Minus sign uses one character position in field
 - Integer width inserted between % and conversion specifier
 - **%4d** – field width of **4**

- Precision
 - Meaning varies depending on data type
 - Integers (default **1**)
 - Minimum number of digits to print
 - If data too small, prefixed with zeros
 - Floating point
 - Number of digits to appear after decimal (**e** and **f**)
 - For **g** – maximum number of significant digits
 - Strings
 - Maximum number of characters to be written from string
 - Format
 - Use a dot (.) then precision number after %
%.3f

- Field width and precision
 - Can both be specified
 - **%width.precision**
%5.3f
 - Negative field width – left justified
 - Positive field width – right justified
 - Precision must be positive
 - Can use integer expressions to determine field width and precision values

Printing Literals and Escape Sequences

- Printing Literals
 - Most characters can be printed
 - Certain "problem" characters, such as the quotation mark "
 - Must be represented by escape sequences
 - Represented by a backslash \ followed by an escape character

- Table of all escape sequences

| <i>Escape sequence</i> | <i>Description</i> |
|------------------------|--------------------------------------------------------|
| <code>\'</code> | Output the single quote (') character. |
| <code>\"</code> | Output the double quote (") character. |
| <code>\?</code> | Output the question mark (?) character. |
| <code>\\</code> | Output the backslash (\) character. |
| <code>\a</code> | Cause an audible (bell) or visual alert. |
| <code>\b</code> | Move the cursor back one position on the current line. |
| <code>\f</code> | Move the cursor to the start of the next logical page. |
| <code>\n</code> | Move the cursor to the beginning of the next line. |
| <code>\r</code> | Move the cursor to the beginning of the current line. |
| <code>\t</code> | Move the cursor to the next horizontal tab position. |
| <code>\v</code> | Move the cursor to the next vertical tab position. |

`scanf` Statement

- Input operation in basic C programming is to get input data from the keyboard using `scanf` statement.
- While output operation is to display the data at the monitor screen using `printf` statement.

Formatting Input with `scanf`

- **`scanf`**
 - Input formatting
 - Capabilities
 - Input all types of data
 - Input specific characters
 - Skip specific characters
- Format
 - **`scanf`**(*format-control-string*, *other-arguments*);
 - Format-control-string
 - Describes formats of inputs
 - Other-arguments
 - Pointers to variables where input will be stored
 - Can include field widths to read a specific number of characters from the stream

Formatting Input with `scanf`

| Conversion specifier | Description |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Integers</i> | |
| d | Read an optionally signed decimal integer. The corresponding argument is a pointer to integer. |
| o | Read an octal integer. The corresponding argument is a pointer to unsigned integer. |
| x or X | Read a hexadecimal integer. The corresponding argument is a pointer to unsigned integer. |
| <i>Floating-point numbers</i> | |
| f | Read a floating-point value. The corresponding argument is a pointer to a floating-point variable. |
| <i>Characters and strings</i> | |
| c | Read a character. The corresponding argument is a pointer to char , no null (<code>'\0'</code>) is added. |
| s | Read a string. The corresponding argument is a pointer to an array of type char that is large enough to hold the string and a terminating null (<code>'\0'</code>) character—which is automatically added. |

Exercise

What data types would you use to hold the following data?

Customer name

Your house number

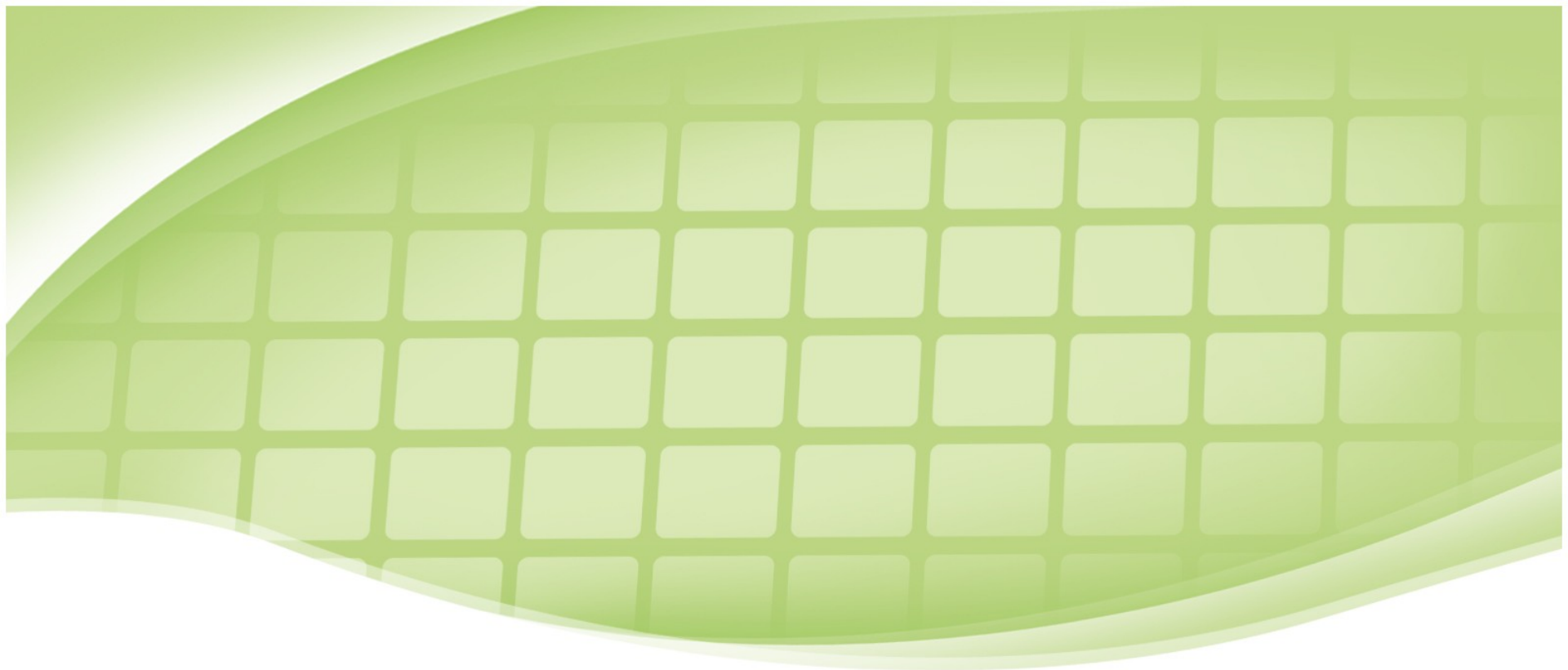
A price

Car registrations

The time

A six digit number

Write C statements to declare them all.



Chapter 3.1

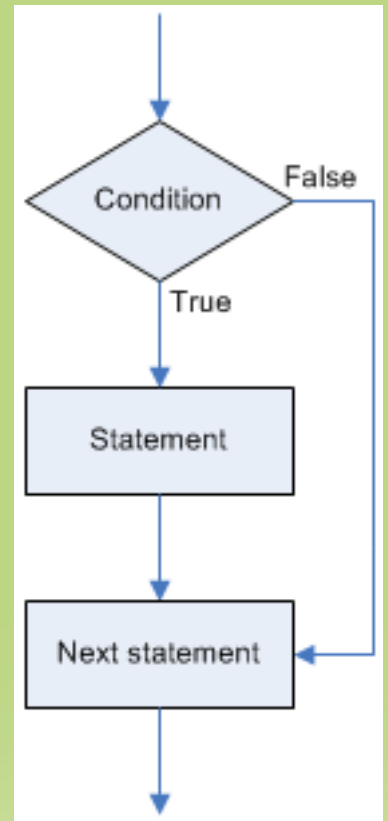
Selection Structure

Introduction

- Ability to change the flow of program execution.
- Allows the program to decide an action based upon user's input or other processes.

if Selection Statement

- Primary tool of selection structure.
- The condition is TRUE – execute the statement.
- The condition is FALSE – skip the statement.



if Selection Statement

```
if ( condition )
```

Execute this statement if
condition is TRUE

i.e.

```
if (5 < 10)
```

```
    printf("Five is less than ten");
```

- Evaluate the statement, "is five less than ten"

Example

```
#include <stdio.h>

int main()
{
    int num;
    printf("Enter a number between -10 and 10: ");
    scanf("%d", &num);
    if (num > 0)
        printf("%d is a positive number\n", num);
    return 0;
}
```

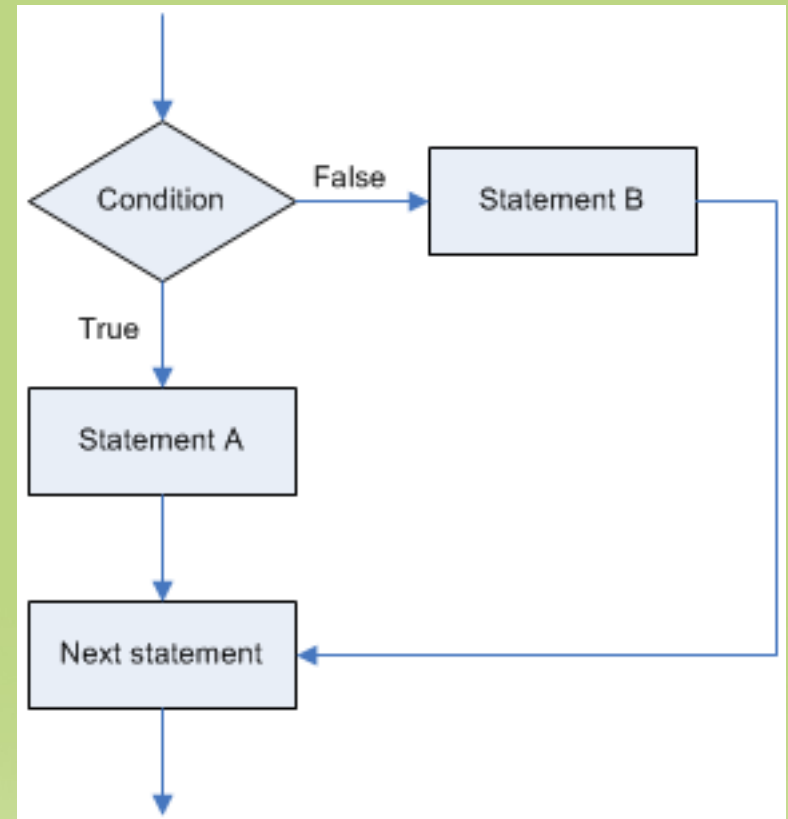
Sample output:

Enter a number between -10 and 10: 6

6 is a positive number

if else Selection Statement

- Execute certain statement if the condition is false
- The condition is TRUE – execute the statement A
- The condition is FALSE – execute the statement B



if else Selection Statement

```
if ( condition )  
    Statement A;  
else  
    Statement B;
```

i.e.

```
if ( 5 < 10 )  
    printf("Five is less than ten");  
else  
    printf("Five is greater than  
ten");
```

Example

```
#include <stdio.h>

int main()
{
    int num;
    printf("Enter a number between -10 and 10: ");
    scanf("%d", &num);
    if (num > 0)
        printf("%d is a positive number\n", num);
    else
        printf("%d is a negative number\n", num);
    return 0;
}
```

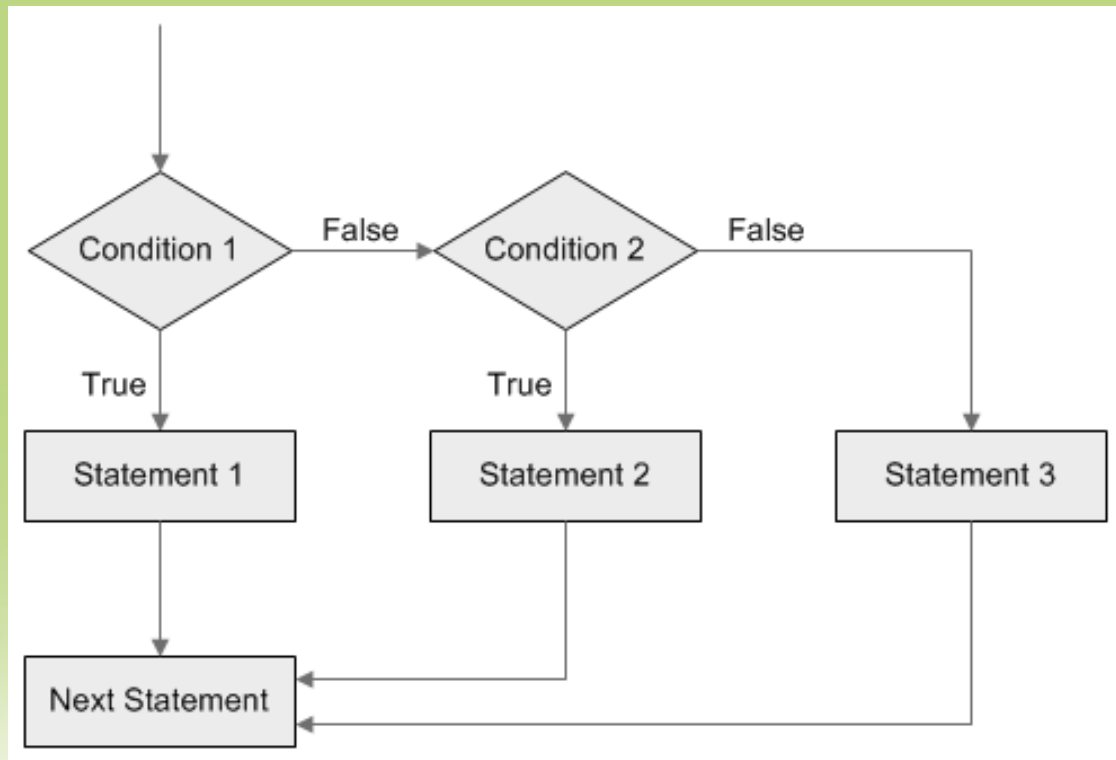
Sample output:

Enter a number between -10 and 10: -2

-2 is a negative number

`if else if` Selection Statement

- Test additional conditions
- Works the same way as normal if statement



Example

```
/* You are young, old or really old :) */
#include <stdio.h>

int main()
{
    int age;
    printf( "Please enter your age: " );
    scanf( "%d", &age );
    if ( age <= 40 )
        printf ( "You are pretty young!\n" );
    else if ( age > 40 && age < 80 )
        printf( "You are old\n" );
    else
        printf( "You are really old\n" );
    return 0;
}
```

Sample output:

Please enter your age: 58

You are old

More than one statement

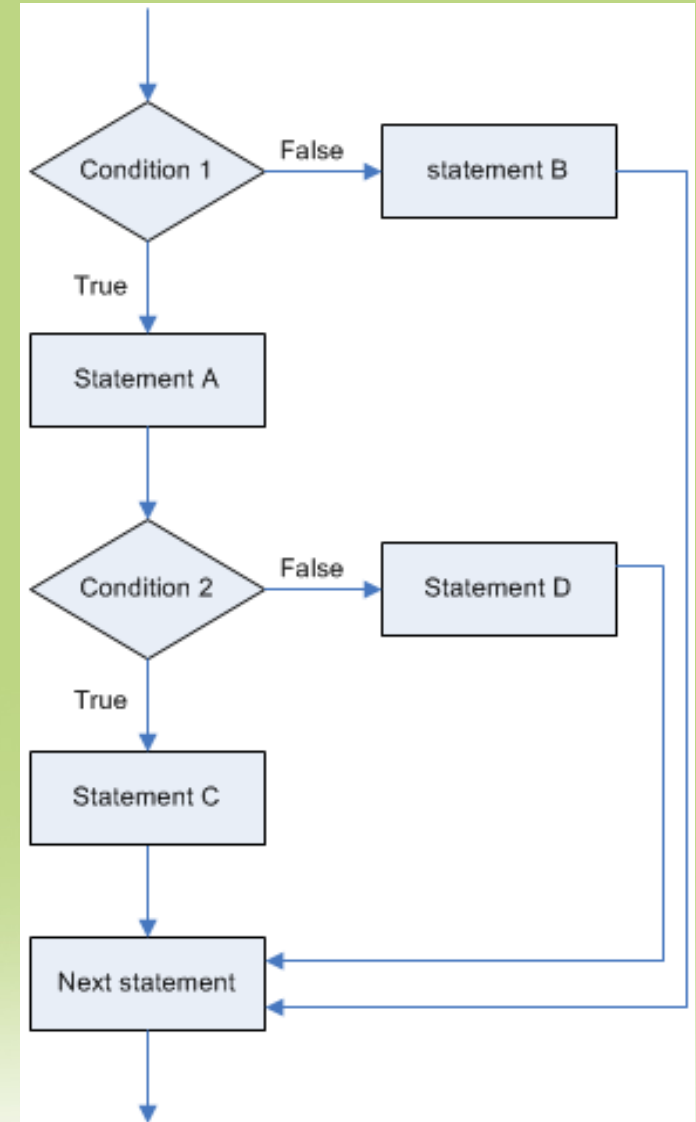
- Multiple statements within selection structure
- Use braces { }
- Block of code

```
if ( condition ) {  
    Statement 1  
    Statement 2  
    ...  
    Statement N  
}
```

Nested **if** Selection Structure

- **if** within an **if** is called nested if

```
if ( condition 1 )  
    Statement A;  
    if ( condition 2 )  
        Statement C;  
    else  
        Statement D;  
else  
    Statement B;
```



Example

```
int main()
{
    int choice;
    printf("Enter a number between -10 and 10: ");
    scanf("%d", &choice);
    if (num > 0) {
        printf("%d is a positive number\n", num);
        if (num % 2 == 0)
            printf("%d is an even number\n", num);
        else
            printf("%d is an odd number\n", num);
    }
    else
        printf("%d is a negative number\n", num);
    return 0;
}
```

Sample output:

Enter a number between -10 and 10: 6

6 is a positive number

6 is an even number

switch Selection Statement

- Substitutes for long **if** statements
- Select from multiple options
- Consists of a series of case label sand optional default case
- Compares the value of variable or expression with values specified in each case

switch Selection Statement

```
switch (variable ){  
    case value 1:  
        Statement A;  
        break;  
    case value 2:  
        Statement B;  
        break;  
    ...  
    default:  
        Statement N;  
        break;  
}
```

Example

```
#include <stdio.h>

int main()
{
    int choice;
    printf("1. Create a new database\n");
    printf("2. Edit a database\n");
    printf("3. Delete a database\n");
    printf("4. Merge databases\n");
    printf("5. Exit system\n");
    printf("Choose an option: ");
    scanf("%d", &choice);
```

```
    switch(choice) {
        case 1:
            printf("Creating...\n");
            break;
        case 2:
            printf("Editing...\n");
            break;
        case 3:
            printf("Deleting...\n");
            break;
        case 4:
            printf("Merging...\n");
            break;
        case 5:
            printf("Thank you, Bye.\n");
            break;
        default:
            printf("Invalid input!\n");
            break;
    }
    return 0;
}
```

Exercise

Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words “is larger.” If the numbers are equal, print the message “These numbers are equal.” Use only the single-selection form of the if statement you learned in this chapter.

Draw a flowchart for this task:-

Obtains two integer numbers from the keyboard, and display which is larger of the two numbers.

Exercise

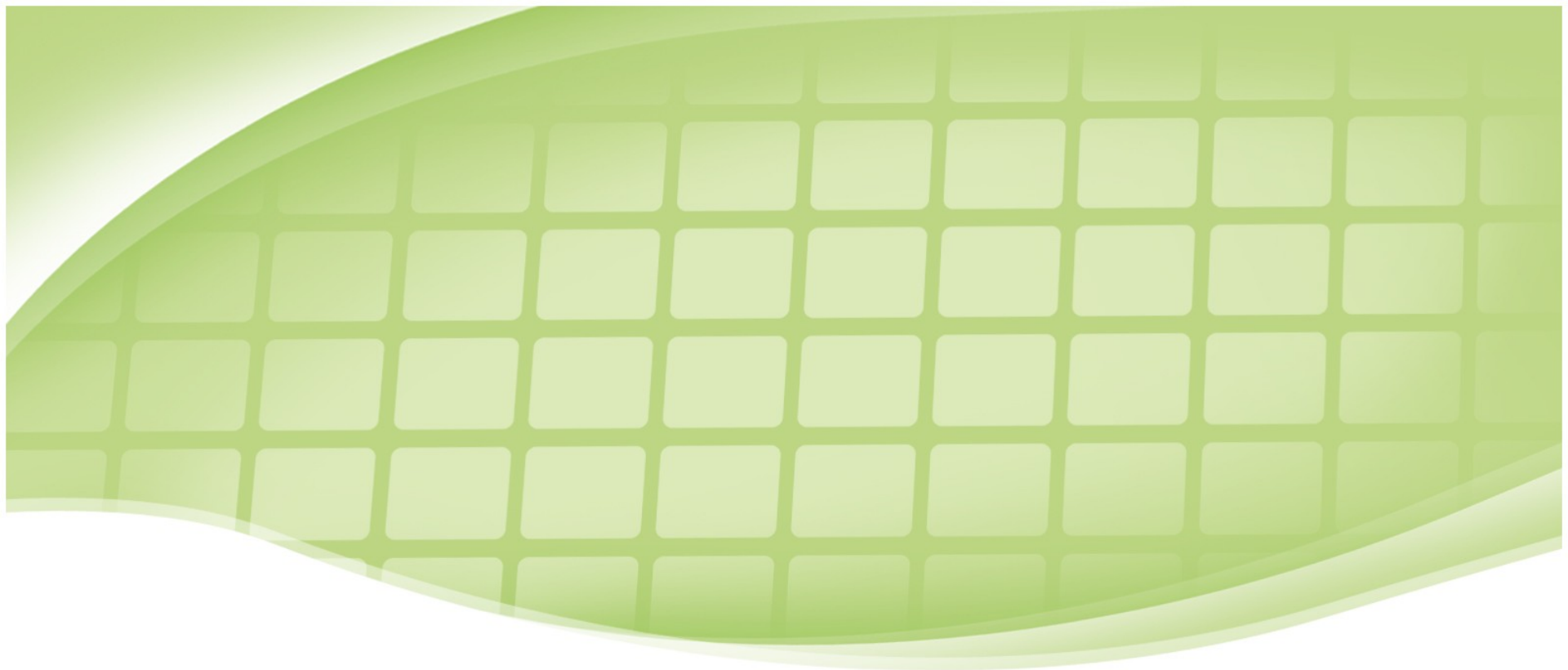
Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words “is larger.” If the numbers are equal, print the message “These numbers are equal.” Use only the single-selection form of the if statement you learned in this chapter.

Draw a flowchart for this task:-

Obtains two integer numbers from the keyboard, and display which is larger of the two numbers.

Exercise

Using 'switch' and 'case statement', write a program determine whether the number we key in is odd or even and that print out adjacent "*" sign same as that number.
(Example: For number 6, we will print out
*****).



Chapter 3.2

Repetition Structure

Introduction

- What if we want to display numbers from 1 to 100?
- What if we want to repeat a thousand times of 10 statements?
- It is possible to do it, but it is not a convenience and it would take a long time to complete
- Ability to specify repetition of compound statements is as important in programming –
Loop

Introduction

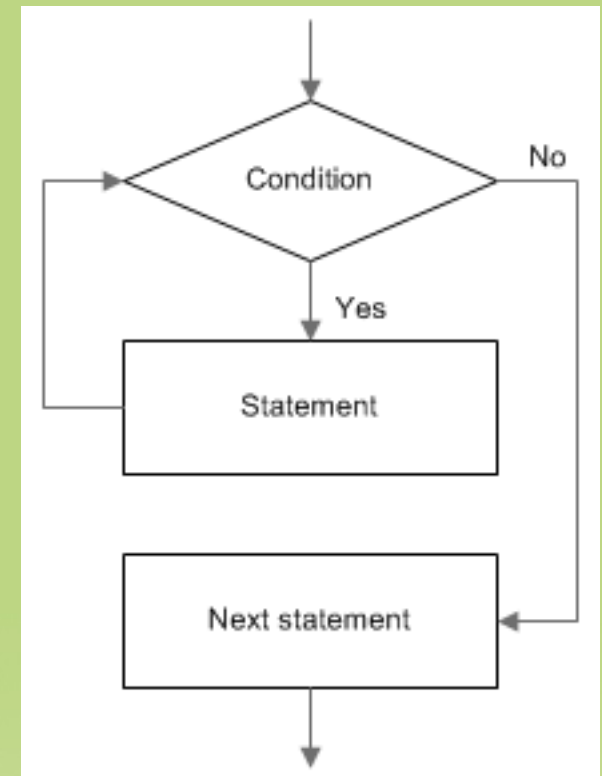
- Two types of loop:
 - Finite loop
 - Infinite loop
- Finite (counter-controlled) loop is a repetition in which it will continue looping until certain condition is met
- Infinite loop has no certain condition in which it will continue looping forever

Introduction

- Three requirements of defining a definite loop
 - 1.counter initialization
 - 2.increment of counter
 - 3.loop condition
- **while, do while** and **for** loop

while Repetition Structure

- Allows the repetition of a set of statements execution to continue for as long as a specified condition is TRUE
- When the condition becomes false, the repetition terminates



while Repetition Structure

```
while ( condition ) {  
    statements;  
}
```

```
while ( a < b ) {  
    printf("a is less than b\n");  
}
```

Example

- Test the condition
- Execute statements within **while** loop's block
- Jump back to the top and re-test the condition

```
#include <stdio.h>

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

Example

Sample output:

0

1

2

...

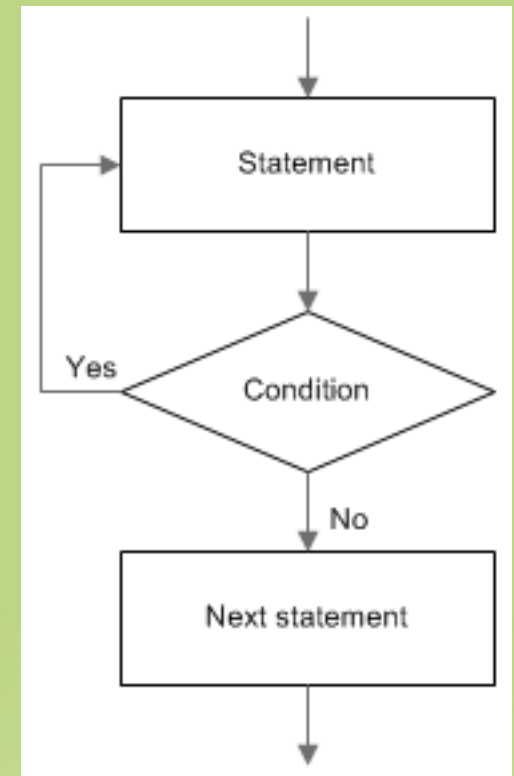
9

```
#include <stdio.h>

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

do while Repetition Structure

- Condition is tested at the end of the block instead of at the beginning
- Block will be executed at least once



do while Repetition Structure

```
do {  
    statements;  
} while ( condition );
```

```
do {  
    printf("a is less than b\n");  
} while ( a < b );
```


Example

- Execute statements within **do** **while** loop's block
- Test the condition
- Jump back to the top and re-execute statements

```
#include <stdio.h>

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

Example

Sample output:

0

1

2

...

9

```
#include <stdio.h>

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

for Repetition Structure

- **for** loop combines the loop control parameters into single statement
- Loop control parameters – variable initialization, variable update and loop condition

for Repetition Structure

```
for ( a; b; c ) {  
    statements;  
}
```

- **a** (variable initialisation) is evaluated once only - before entering the loop
- **b** (condition) determines whether or not the program loops
- **c** (variable update) is evaluated after an iteration of the loop – loop counter

Example

- Initializes count equals to zero
- Test the condition
- Execute statements within **for** loop's block
- Jump back to the top and re-test the condition

```
#include <stdio.h>

int main()
{
    int count;
    for(count = 0; count < 10; ++count) {
        printf("%d\n", count);
    }
    return 0;
}
```

Example

Sample output:

0

1

2

...

9

```
#include <stdio.h>

int main()
{
    int count;
    for(count = 0; count < 10; ++count) {
        printf("%d\n", count);
    }
    return 0;
}
```

Nested Loop

- A loop within a loop

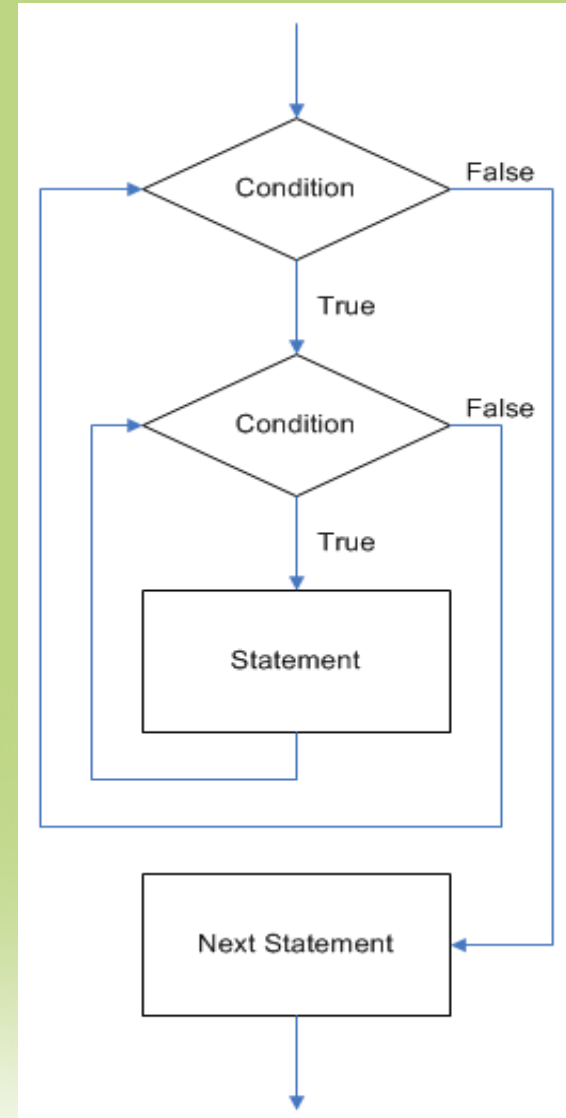
Loop1

Loop2

statement

return

return



Example

```
#include <stdio.h>

int main()
{
    int i, j;
    int sum;
    for( i = 1; i < 4 ; i++ ) {
        j = 1;
        sum = 0;
        do {
            sum += j++;
        }while( j <= i);
        printf("%d\t\t%d\n", i, sum);
    }
    return 0;
}
```

Sample output:

1 1

2 3

3 6

Infinite Loop

- Loops forever
- Contains no condition
- Condition that can never be satisfied

Example

```
#include <stdio.h>

int main()
{
    int i = 0;
    while (1) { // loop condition always true
        printf("%d\n", i);
        i++;
    }

    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int i;
    // loop condition always true
    for ( i = 0; 1 ; i++) {
        printf("%d\n", i);
    }

    return 0;
}
```

break and continue Statements

- Can be executed in **while**, **do while** and **for** statement
- Used to alter the flow of control

Example

- Causes an immediate exit from that statement

```
for(i = 20; i > 0; i--) {  
    if(i % 2 != 0)  
        break;  
    sum += i;  
}
```

Example

- Skip the remaining statements
- Performs the next iteration of the loop

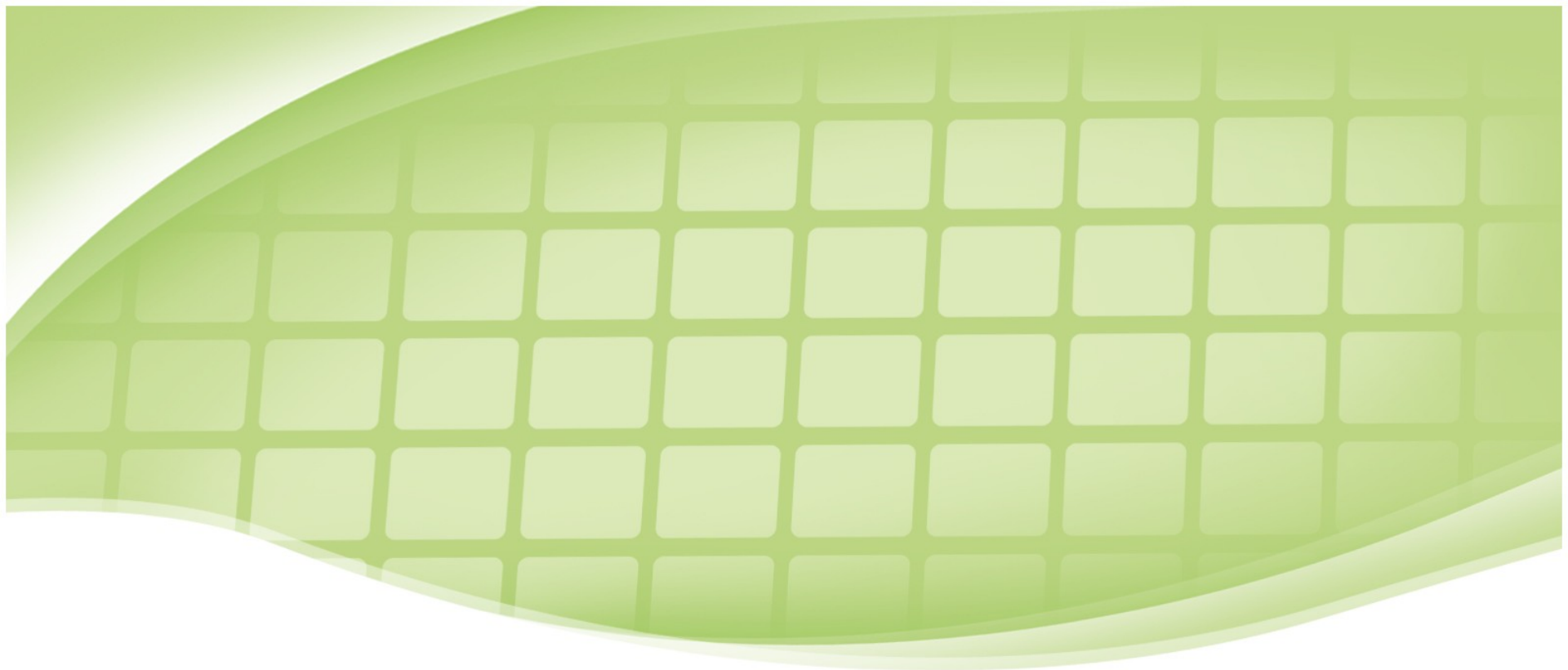
```
for(i = 0; i < 20; i++) {  
    if(i % 2 != 0)  
        continue;  
    sum += i;  
}  
  
return 0;
```

Exercise

Use a loop to produce the following output:

- a) 12345678910
- b) 3, 8, 13, 18, 23
- c) 20, 14, 8, 2, -4, -10

Write a program that will display all odd numbers between 1 to 40.



Chapter 4

Functions

Introduction

- Mini-program where a group of statements are executed.
- Programmer-defined functions.
- Break a large program into smaller sets of statements according to their tasks.

Benefits of Function

- To make program more manageable
 - divide-and-conquer approach (construct a program from smaller components/modules).
- Software reusability
 - reuse existing functions for new programs
- Avoid repeating code

```

#include <stdio.h>

int main()
{
    int count;
    float score;
    float sum = 0;
    float ave;
    printf("Enter scores for");
    printf(" class A (-1 to end input):\n");
    for(count = 0; ; count++) {
        scanf("%f", &score);
        if (score == -1) break;
        else sum += score;
    }

    ave = sum / count;
    printf("Class A average is %.2f\n\n", ave);

    sum = 0;
    printf("Enter scores for");
    printf(" class B (-1 to end input):\n");
    for(count = 0; ; count++) {
        scanf("%f", &score);
        if (score == -1) break;
        else sum += score;
    }

    ave = sum / count;
    printf("Class B average is %.2f\n\n", ave);
    return 0;
}

```

```

#include <stdio.h>
void UserInstruct (char cl);
float GetSum(void);
float CalcAvg (float summation, int cnt);
void PrintAverage(float ave, char cl);
int count; // global variable

int main()
{
    float sum;
    float ave;
    UserInstruct('A'); //Class A
    sum = GetSum(); //use GetSum
    ave = CalcAvg(sum, count); //use CalcAvg
    PrintAverage(ave, 'A'); //Use PrintAverage
    UserInstruct('B');
    sum = GetSum();
    ave = CalcAvg(sum, count);
    PrintAverage(ave, 'B');
    return 0;
}

void UserInstruct(char cl)
{
    printf("Enter scores for");
    printf(" class %c (-1 to end):\n", cl);
}

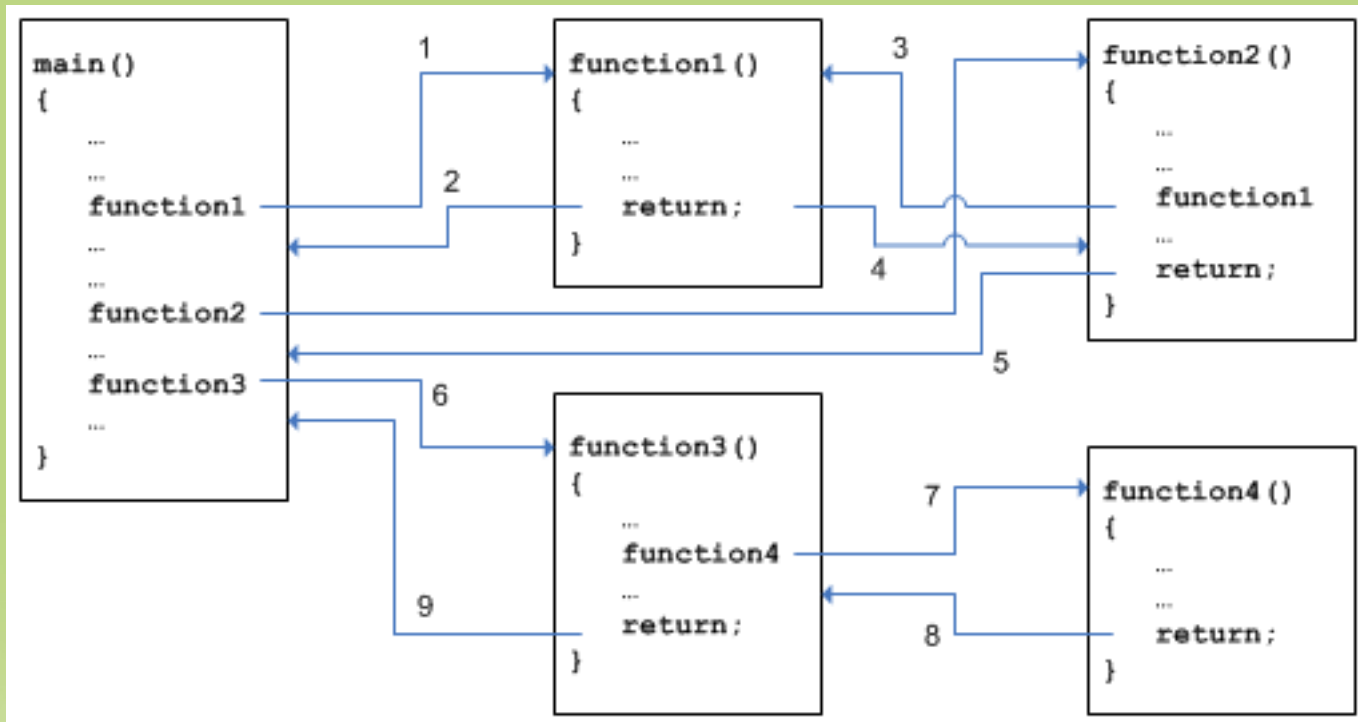
float GetSum(void)
{
    float score;
    float sum = 0;
    for(count = 0; ; count++) {
        scanf("%f", &score);
        if (score == -1) break;
        else sum += score;
    }
    return sum;
}

float CalcAvg (float summation, int cnt)
{ return summation/cnt; }

void PrintAverage(float ave, char cl)
{ printf("Class %c average is %.2f\n\n", cl, ave); }

```

Sequence of Execution



3 Elements of Function

- Function definition
- Function prototype / declaration
- Calling function

Function Definition

- Specifies the specification of a function
 - return value type, function name, arguments.
- Defines the operation to be performed when a function is invoked.

Function Definition

```
return_value_type function_name(parameter_list)
{
    statements;
}
```

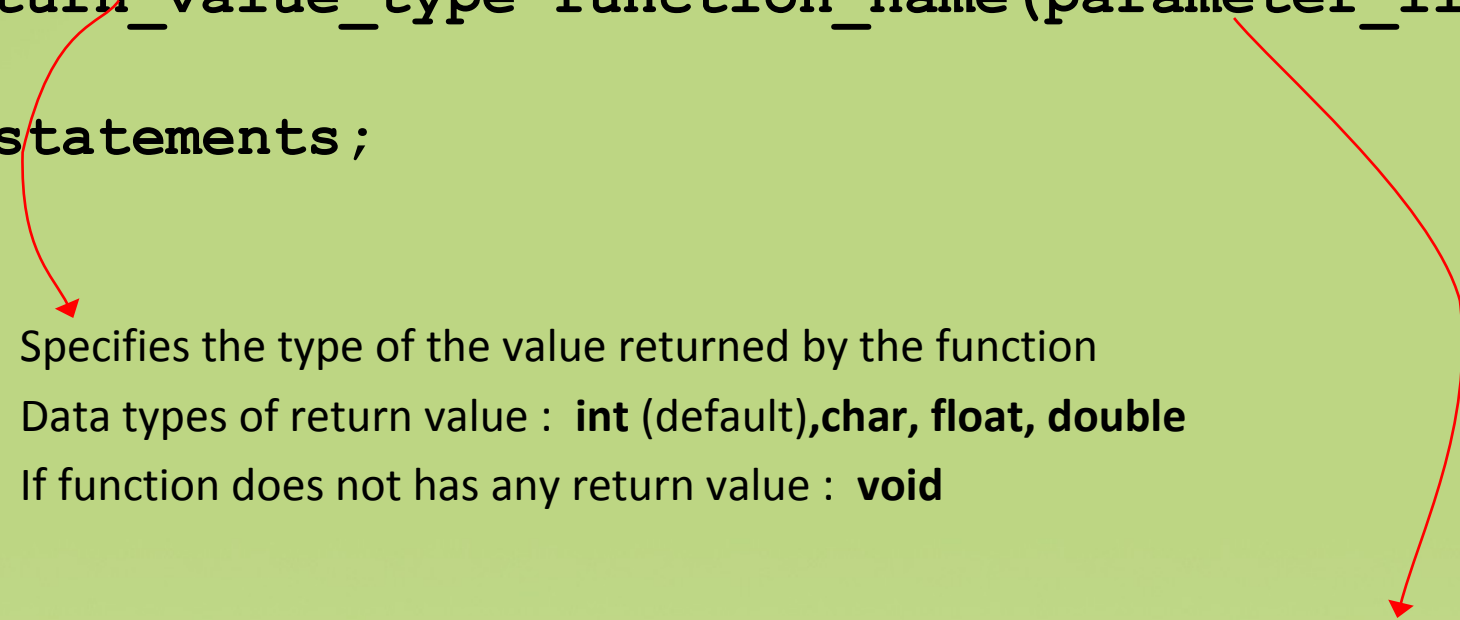
function body

function header



Function Definition

```
return_value_type function_name(parameter_list)
{
    statements;
}
```



- ❑ Specifies the type of the value returned by the function
- ❑ Data types of return value : **int** (default), **char**, **float**, **double**
- ❑ If function does not has any return value : **void**

- List of variable names received by the function from the caller – arguments (data input)
- parameter_list : (**parameter_type parameter name**)
- parameter_type: **int** ,**char**, **float**, **double**
- If function does not has any parameter_list : (**void**) or **()**

Function Prototype

- To declare function if the function definition is written below the main function.
- Otherwise, no need function prototype.
(function definition above the main function)
- Almost similar with function definition header, but must be written before the main function with semicolon at the end of statement.

Function Prototype

`return_value_type function_name(parameter_list) ;`



- ❑ Specifies the type of the value returned by the function
- ❑ Data types of return value : **int** (default), **char**, **float**, **double**
- ❑ If function does not has any return value : **void**

- List of variable names received by the function from the caller – arguments (data input)
- parameter_list : (**parameter_type**)
- parameter_type: **int** ,**char**, **float**, **double**
- If function does not has any parameter_list : (**void**) or (**()**)

Calling Function

- Two ways to invoke/call function
 - Call-by-value
 - Call-by-reference.
- Calling function statement is written in the caller function (main function).

Call-by-value VS Call-by-reference

Call-by-value

- Copies of the arguments' value are made and passed to the called function
- Changes to the copy do not effect an original variable's value
- Available in C language

Call-by-reference

- References of the arguments' value are passed to the called function
- Called function can modify the original variable's value
- Possible to simulate

Example 1

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


```
void function_name(void) ;
```

```
int main() {  
    statements;  
    function_name() ;  
    return 0;  
}
```

Function prototype



Calling function



```
void function_name(void) {  
    statements;  
}
```

Function definition



Example 2

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

```
double function_name(int,double) ;
```

Function prototype



```
int main() {  
    int p1, double p2;  
    statements;  
    function_name(p1,p2) ;  
    return 0;  
}
```

Calling function



```
double function_name(int pm1,double pm2) {  
    statements;  
}
```

Function definition



Example 3

```
#include <stdio.h>

void func1(int n1);

main()
{
    int n1;
    n1 = 5;
    printf("Call-by-value");
    printf("Before func1 is called\n");
    printf("n1 = %d\n\n", n1);
    func1(n1);
    printf("After func1 is called\n");
    printf("n1 = %d\n\n", n1);
    return 0;
}

void func1(int n1)
{
    n1 = 10;
    printf("In func1\n");
    printf("n1 = %d\n\n", n1);
}
```

Sample output:
Call-by-value

Before func1 is called

n1 = 5

In func1

n1 = 10

After func1 is called

n1 = 5

Example 4

```
#include <stdio.h>

int power(int base, int n);

main()
{
    int b, n, res;
    printf("Enter base and power: ");
    scanf("%d%d", &b, &n);
    res = power(b, n);
    printf("The result is %d.\n", res);
    return 0;
}

int power(int base, int n)
{
    int i, p = 1;
    for(i = 0; i < n; i++)
        p = p * base;
    return p;
}
```

Sample output:

Enter base and power: 5 4

The result is 625.

function prototype

function call

function definition

Scope Variable

- Variables declared in function **main()** are known only within function **main()** – not accessible outside of the block
- It is applied to variables defined within a block that is inside other block
- But variables defined in the outer block are accessible in the inner block

Scope Variable

```
#include <stdio.h>

int main()
{
    int a = 0;
    while(++a < 10) {
        int b = 1;
        printf("%d\t%d\n", a, b);
        ++b;
    }
    printf("%d\t%d\n", a, b);
    return 0;
}
```

- Variable **a** is declared in **main()**
- Variable **b** is declared within while loop block (inner block)
- **a** is the outer block variable therefore it is accessible within inner block
- **b** is not accessible in the outer block because it is while loop variable

Scope Variable

```
#include <stdio.h>

int func(int n);

main()
{
    int num1, num2, res;
    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);
    res = func(num1);
    printf("%d\n", num2);
    return 0;
}

int func(int n)
{
    int r;
    r = n + num2 * 2;
    return r;
}
```

num2 is declared in main()

num2 is accessible in main() only

num2 is not accessible in func()
num2 is considered undeclared in func()

Global Variable

- Variables declared outside any function is called global variables
- Global variables are variables that can be accessed in any function
- Normally they are declared before function **main()**

Global Variable

```
#include <stdio.h>

int func(int n);
int num2;

main()
{
    int num1, res;
    printf("Enter two numbers: ");
    scanf("%d%d", &num1, &num2);
    res = func(num1);
    printf("%d\n", num2);
    return 0;
}

int func(int n)
{
    int r;
    r = n + num2 * 2;
    return r;
}
```

num2 is a global variable

num2 is accessible in func()

Recursive Function

- Function that calls itself either directly or indirectly through another function

Example 5

```
#include <stdio.h>

long factorial(long num);

int main()
{
    int i;
    for(i = 0; i < 5 ; i++)
    {
        printf("%d! = %d\n", i, factorial(i));
    }
    return 0;
}

long factorial(long num)
{
    if(num <= 1)
        return 1;
    else
        return(num * factorial(num - 1));
}
```

Sample output:

0! = 1

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

Exercise

- Write a program that utilizes a function with parameters. The program reads 3 integer numbers from the user and sends the 3 input to the function. The function finds the smallest value between the 3 integer numbers and return the value to the main function.

Enter three inter numbers:

30

80

20

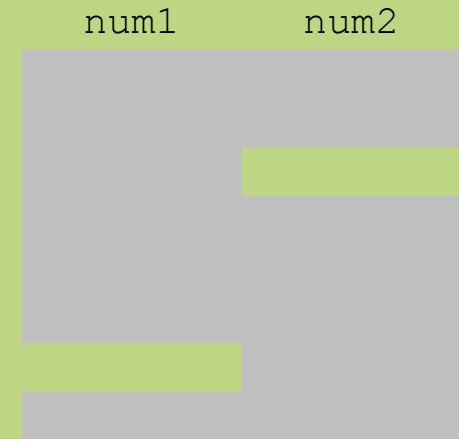
Smallest value: 20

Exercise

- Consider the following program, what is the value of num1 and num2 after each statement is executed?

| num1 | num2 |
|------|------|
| | |
| | 2 |
| | |
| 1 | |
| | |
| 1 | 3 |
| 1 | 7 |
| 1 | 21 |


```
1 void funcA(int);
2 int funcB(int);
3 int num2 = 2;
4
5 int main(void)
6 {
7     int num1 = 1;
8
9     num2 = funcB(num1);
10    funcA(num1);
11    funcA(num2);
12
13    return 0;
14 }
15 void funcA(int num1)
16 {
17     num1 = num1 + num2;
18     num2 = num1 + num2;
19 }
20 int funcB(int num1)
21 {
22     num1 = num1 + num2;
23     return num1;
24 }
```





Chapter 5

Arrays

Introduction

- A group of memory locations that have the same name and the same type.
- Allows to store a sequence of values.

Introduction cont.

- First element in an array is the zeroth element, $a[0]$
- In the example to the right: _
Second element is $a[1] = 6$. Element two is $a[2] = 11$. Different way of statement resulting in different meaning!
- If $b=2$ and $c=4$, then
 $a[b+c] = a[b+c]+2;$
 $a[2+4] = a[2+4]+2;$
 $a[6] = a[6]+2;$
 $a[6] = 10;$

Defining and Initializing of One Dimensional Array

- `int a[5];`
- `int a[5] = {0};` `//all elements equal zero`
- `int a[5] = {10, 20, 30, 40, 50};`
- `int a[5] = {10, 20, 30, 40, 50, 60};` `//error`
- `float a[5] = {1.1, 2.2, 3.3, 4.4, 5.5};`
- `char a[6] = {'h', 'e', 'l', 'l', 'o'};`

Note: `a[5]` carries the value `'/0'`, more on this later.

Example

```
#include<stdio.h>

int main()
{
    int i;
    int a[10]={1,2,3,4,5,6,7,8,9,10};

    for(i=0;i<10;i++)
        printf("array a[%d] = %d\n",i,a[i]);

    return 0;
}
```

```
array a[0] = 1
array a[1] = 2
array a[2] = 3
array a[3] = 4
array a[4] = 5
array a[5] = 6
array a[6] = 7
array a[7] = 8
array a[8] = 9
array a[9] = 10
```

```
Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

Input/Output

- Declare an integer array named a with 10 elements
`int a[10];`
- Input an integer into 6th element
`scanf("%d ", &a[5]);`
- Output the content of 6th element
`printf("a[5] is %d.\n ", a[5]);`

Character Arrays

- Store characters
- Character arrays are capable of storing strings
- Revise: %c for characters (eg. A, b, c, D)
 %s for strings (eg. hi, welcome, good)

Character Arrays

- `char string[]="hello";`
- Initializes the elements of array string to the individual characters in the string "hello"
- string "hello" has 5 characters plus a special string termination character called **null character** '\0'

Character Arrays

```
char x[]={ 'h', 'e', 'l', 'l', 'o', '\0' };
```

- `x[0] = 'h'`
- `x[1] = 'e'`
- `x[2] = 'l'`
- `x[3] = 'l'`
- `x[4] = 'o'`
- `x[5] = '\0'`

Input/Output

- Declare a char array named string1 with 20 elements
char string1[20];
- Input a string into the array
scanf("%s", string1);
- Output the array's content
printf("The content is %s.\n", string1);
- Function scanf will read characters until a space, tab, newline or EOF is encountered

```
#include <stdio.h>

int main()
{
    char string1[20];
    char string2[]="Good morning";

    printf("Enter a string: ");
    scanf("%s", string1);

    printf("string1 is: %s\nstring2 is: %s\n", string1, string2);
    return 0;
}
```

```
Enter a string: hello there
string1 is: hello
string2 is: Good morning
Press any key to continue
```

Passing Arrays to Functions

- Modify the elements in function body is modifying the actual elements of the array

Cont.

```
int array[30];           //array declaration
```

```
void func(int array[]) {  
    ...                  //function prototype  
}
```

```
func(array)              //call function
```

the passed array



Passing Elements to Functions

- Passing an individual array elements are like passing a variable
- The actual value is not modified

Cont.

```
int array[30];
```

```
void func(int e) {
```

```
    ...
```

```
}
```

```
func(array[5])
```

variable e copies the value of array element



the passed element



Passing Elements to Functions

```
#include<stdio.h>

void entire_array(int []);
void element_array(int, int, int);

main()
{
    int i;
    int x[3]={1,2,3};
    for(i=0;i<3;i++)
        printf("%d\t",x[i]);

    printf("\n");

    entire_array(x);

    for(i=0;i<3;i++)
        printf("%d\t",x[i]);

    printf("\n");

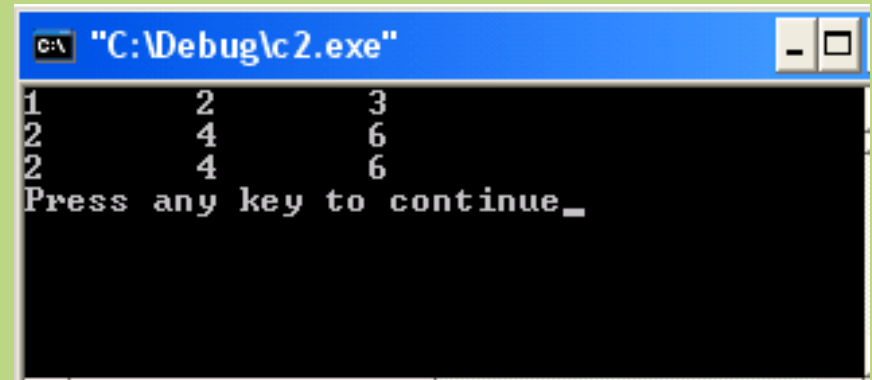
    element_array(x[0],x[1],x[2]);

    for(i=0;i<3;i++)
        printf("%d\t",x[i]);

    printf("\n");
}

void entire_array(int y[])
{
    int i;
    for(i=0;i<3;i++)
        y[i]*=2;
}

void element_array(int j, int k, int l)
{
    j+=1;
    k+=2;
    l+=3;
}
```



```
C:\Debug\c2.exe
1      2      3
2      4      6
2      4      6
Press any key to continue_
```

Two Dimensional Array

- To represent table of values consisting of information arranged in rows and columns

Defining and Initializing

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

1st Row



2nd Row



- Eg. `int b[3][4];`

Example

```
void printArray(const int a[][3]);

int main()
{
    int array1[2][3] = {{1,2,3},{4,5,6}};
    int array2[2][3] = {1,2,3,4,5};
    int array3[2][3] = {{1,2},{4}};

    printf("Values in array1 by row are:\n");
    printArray(array1);

    printf("Values in array2 by row are:\n");
    printArray(array2);

    printf("Values in array3 by row are:\n");
    printArray(array3);

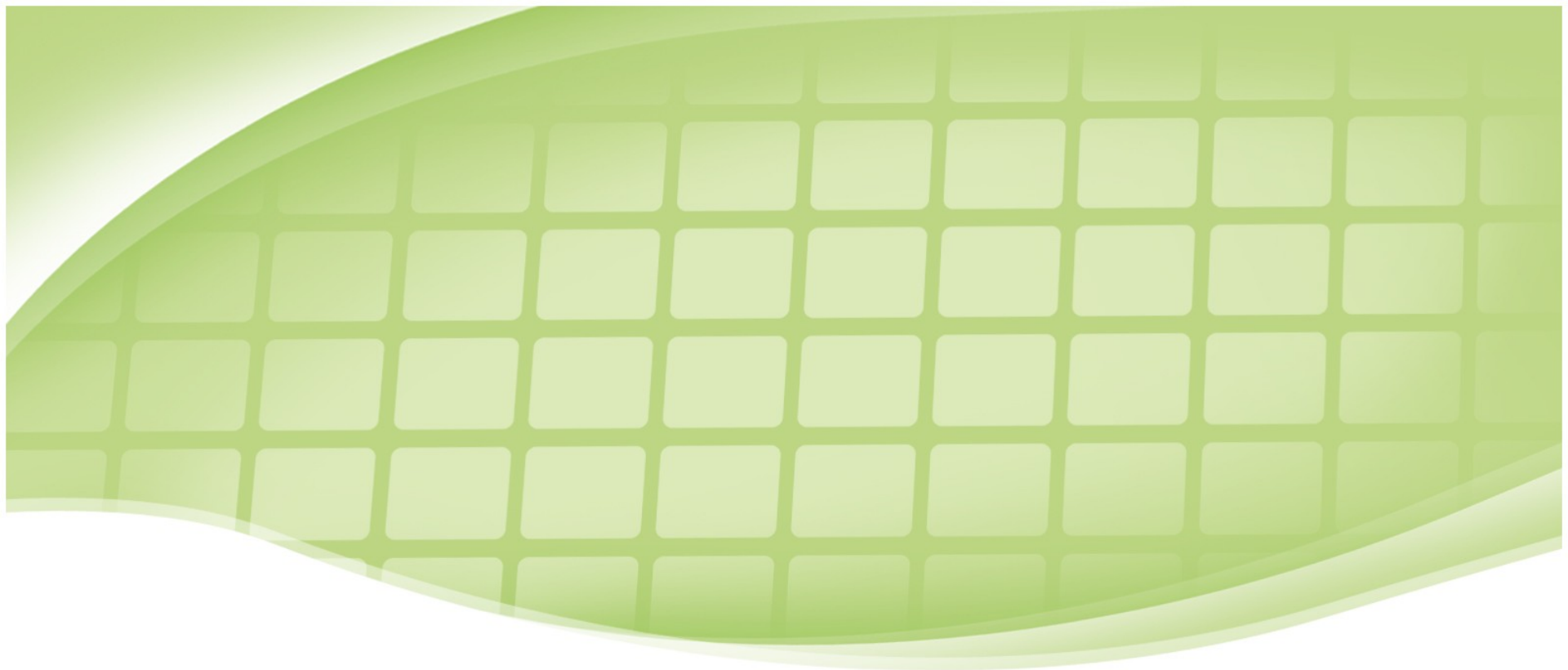
    return 0;
}

void printArray(const int a[][3])
{
    int i,j;
    for(i=0;i<=1;i++)
    {
        for(j=0;j<=2;j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
}
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
Press any key to continue
```

Exercise

- Write a program to print the values of each element of a 3-by-2 array using nested loop.. Initialize the array as:
 $\{12.0, 14.2\}, \{13.0, 16.5\}, \{10.5, 11.0\}$



Chapter 6

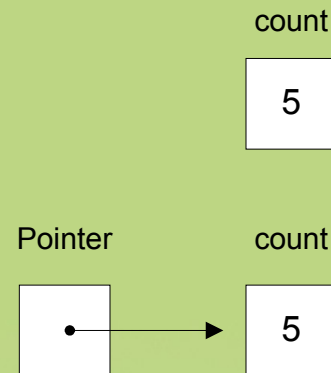
Pointers

Introduction

- Pointers are variables whose values are memory addresses
- A pointer contains an address of a variable that contains a specific value

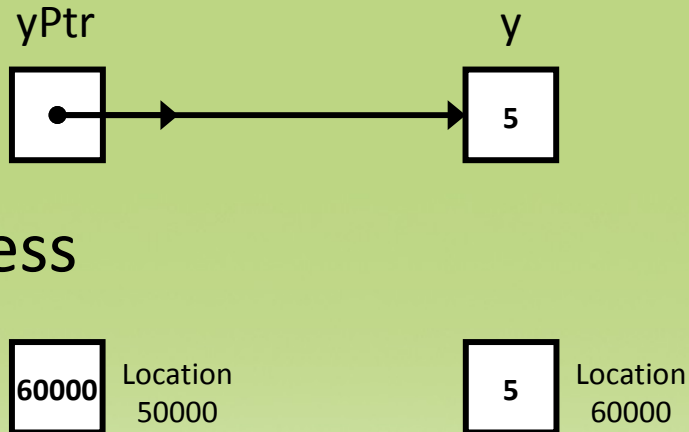
Introduction cont.

- Count directly
references a variable
that contains the value
5
- Pointer indirectly
references a variable
that contains the value
5 (indirection)



Declaring and Initializing

- A pointer of type **int** is a pointer to a variable type **int**
- `int y = 5;`
- `int *yPtr;`
- `yPtr = &y;`
- `&` returns the address of its operand



Using Pointers

- * (indirection) operator returns the value of the object a pointer refers to
- Assume `y = 5`
- `yPtr = &y;`
- `z = *yPtr + 5;`
- `*yPtr = 7;`
- `z = *yPtr + 5;`
- `yPtr = &z;`

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

```
main()
```

```
{
```

```
    int *p;
```

```
    int z;
```

```
    int y=5;
```

```
    p = &y;
```

```
    printf("%p %p %p\n", &y, p, &z);
```

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

```
    z = *p + 5;
```

```
    printf("%d %d %d\n", y, *p, z);
```

```
    *p = 7;
```

```
    printf("%d %d %d\n", y, *p, z);
```

```
    z = *p + 5;
```

```
    printf("%d %d %d\n", y, *p, z);
```

```
    p = &z;
```

```
    printf("%d %d %d\n", y, *p, z);
```

```
}
```

| 0012FF74 | 0012FF74 | 0012FF78 |
|----------|----------|----------|
| 5 5 | | |
| 5 5 10 | | |
| 7 7 10 | | |
| 7 7 12 | | |
| 7 12 12 | | |

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

```
main()
```

```
{
```

```
    int *p;
```

```
    int *z;
```

```
    int x=3, y=5;
```

```
    p = &y;
```

```
    z = &x;
```

```
    printf("%p %p %p %p\n", &x, &y, p, z);
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    *z = *p + 5;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    p = z;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    *p = y + 1;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    z = &y;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    y = *p + 2;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
    x = *z;
```

```
    printf("%d %d %d %d\n", x, y, *p, *z);
```

```
}
```

| 0012FF74 | 0012FF70 | 0012FF70 | 0012FF74 |
|----------|----------|----------|----------|
| 3 | 5 | 5 | 3 |
| 10 | 5 | 5 | 10 |
| 10 | 5 | 10 | 10 |
| 6 | 5 | 6 | 6 |
| 6 | 5 | 6 | 5 |
| 6 | 8 | 6 | 8 |
| 8 | 8 | 8 | 8 |

Call-by-reference

- The **original variable address** is passed to the function by using **&** operator.
- Ability to access the variables directly and to modify them if required.

Example

```
/* Functions prototype */
int squareByValue(int x);
void squareByReference(int *numRef);

int main()
{
    int num1=3;
    int num2=5;
    int temp;
    printf("num1 = %d - before squareByValue is called\n",num1);
    temp=squareByValue(num1);
    printf("Value returned by squareByValue: %d\n",temp);
    printf("num1 = %d - after squareByValue is called\n",num1);

    printf("\nnum2 = %d - before squareByReference is called\n",num2);
    squareByReference(&num2);
    printf("num2 = %d - after squareByReference is called\n",num2);
    return 0;
}

/* Functions definition */
int squareByValue(int x)
{
    return x*x; //caller's argument not modified
}

void squareByReference(int *numRef)
{
    *numRef*=*numRef;
}
```

& (address) operator is specified

a pointer

numRef = numRef * numRef

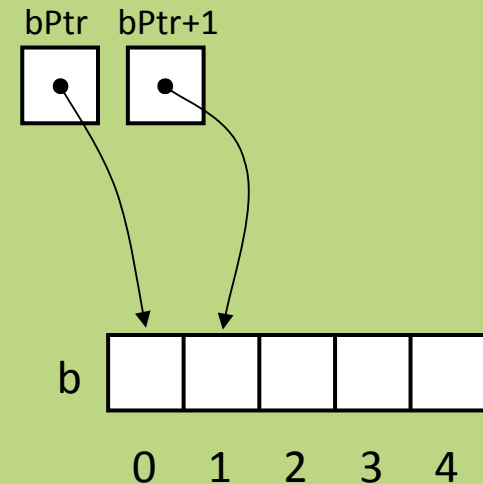
num1 = 3 - before
Value returned: 9
num1 = 3 - after
num2 = 5 - before
num2 = 25 - after

Pointers and Arrays

- Any operation that can be achieved by array subscripting can also be done with pointers

Cont.

- `bPtr = &b[0];`
- `bPtr+1` points to next element
- `bPtr+i` points `i` elements after `bPtr`
- `bPtr-i` points `i` elements before `bPtr`
- `bPtr+i` is the address of `b[i]`
- `*(bPtr+i)` is the content of `b[i]`



Arrays of Pointers

- Since pointers are variables
- Can be stored in arrays
- Array of memory addresses

Example

Sample output:

3

5

10

```
#include <stdio.h>

int main()
{
    int x = 3, y = 5, z = 10;
    int *p[3] = {&x, &y, &z};
    printf("%d\n", *p[0]);
    printf("%d\n", *p[1]);
    printf("%d\n", *p[2]);
    return 0;
}
```

Pointers to Functions

- A pointer to a function contains the address of the function
- Similarly, a function called through a pointer must also pass the number and type of the arguments and the type of return value that is expected

Example

```
#include <stdio.h>

int add(int x, int y); /* declare function */

int main()
{
    int x=6, y=9;
    int (*ptr)(int, int); /* declare pointer to function*/

    ptr = add;             /* set pointer to point to "add"

    printf("%d plus %d equals %d.\n", x, y, (*ptr)(x,y));

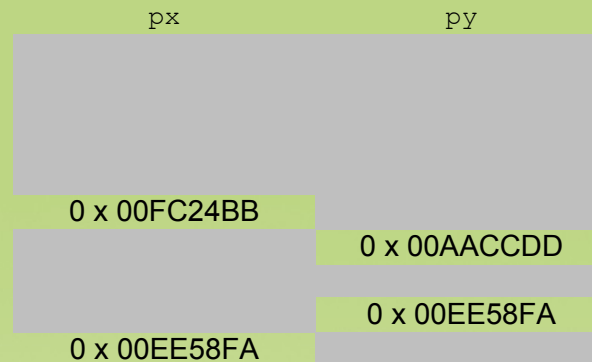
    /* call function using pointer */
    return 0;
}

int add(int x, int y)
{ /* function definition */
    return x+y;
}
```

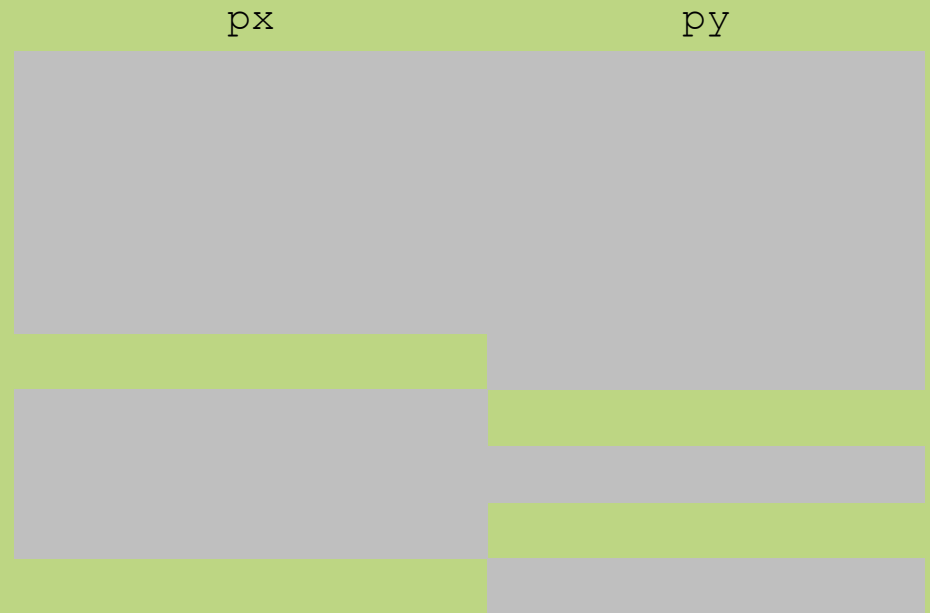
6 plus 9 equals 15.

Exercise

- Consider the following statements, what is the value of px and py after each statement is executed?



```
1  int x, y, z;
2  int *px, *py, *pz;
3
4  x = 5;
5  y = 10;
6  px = &x;
7  py = &y;
8  z = 15;
9  py = &z;
10 px = py
```



Given:

address x = 0 x 00FC24BB

address y = 0 x 00AACCCDD

address z = 0 x 00EE58FA

Working with Files

Opening a File

- FILE pointer which will let the program keep track of the file being accessed
- `FILE *fp;`
- Function `fopen` use to open a file
- `fp = fopen("filename", "mode");`

Filename

- Use double backslashes rather than a single backslash (in the case of string literal)
 - "c:\test.txt" – \t is considered as escape sequence
 - "c:\\test.txt" – correct way to specify backslash

Modes

- r - open for reading
- w - open for writing
- a - open for appending

Example

- `FILE *p1, *p2;`
- `p1 = fopen("data", "r");`
- `p2 = fopen("results", "w");`

Closing a File

- A file must be closed as soon as all operations on it have been completed
- `fclose (p1) ;`
- `fclose (p2) ;`

Get a character from file

- `fgetc (file);`
- Returns the character currently pointed by the internal file position indicator of the specified *file*

Example

- `FILE *fptr;`
- `fptr = fopen("logfile.txt",
"r");`
- `char inpF;`
- `inpF = fgetc(fptr);`

Get string from File

- `fgets(char *str, int num, file);`
- Reads characters from *file* until
 - (num-1) characters have been read or
 - a newline or
 - End-of-File is reached
 - whichever comes first.
- A null character is automatically appended in *str* after the characters read to signal the end of the C string.

Example

- `FILE *fptr;`
- `fptr = fopen("logfile.txt",
"r");`
- `char inpF[20] = " ";`
- `fgets(inpF, 20, fptr);`

Write character to file

- `fputc(int character, file);`
- Writes a character to the *file* and advances the position indicator

Example

- `FILE *fptr = NULL;`
- `fptr = fopen("logfile.txt",
"a");`
- `char outF = 'A';`
- `fputc(outF, fptr);`

Write string to file

- `fputs(const char * str, FILE * stream);`
- Writes the string pointed by *str* to the *file*

Example

- `FILE *fptr = NULL;`
- `fptr = fopen("logfile.txt",
"a");`
- `char outF[20] = "Hello
World\n";`
- `fputs(outF, fptr);`

Read formatted data from file

- `fscanf(file, fmt-control-string, var);`
- Reads data from the *file* and stores them according to the parameter *format* into variables

Example

- `FILE *fptr = NULL;`
- `fptr = fopen("logfile.txt",
"r");`
- `char inpF[20] = " ";`
- `fscanf(fptr, "%s", inpF);`

Write formatted output to file

- `fprintf(file, fmt-control-string, var);`
- Writes to the specified *file* a sequence of data formatted as the *format* argument specifies

Example

- `FILE *fptr = NULL;`
- `fptr = fopen("logfile.txt", "a");`
- `int n = 1;`
- `char name[20] = "Muhammad";`
- `fprintf (fptr, "Name %d - %s\n", n, name);`