Chapter 1 & 2

Introduction to C Language

Chapter 1 & 2 - Introduction to C Language

Outline

Objectives

- In chapter 1&2, you will learn:
  - The history of the C programming language.
  - To become aware of the C standard library.
  - The elements of a typical C program development environment.

  - To be able to write simple programs in C.
  - To be able to use simple input and output statements.
  - To become familiar with fundamental data types.
  - To understand computer memory concepts.
  - To be able to use arithmetic operators.
  - To understand the precedence of arithmetic operators.
  - To be able to write simple decision making statements.

1.1 History of C

- C Language
  - Evolved by Dennis Ritchie from two previous programming languages, BCPL and B
  - Used to develop UNIX
  - Used to write modern operating systems
  - Hardware independent (portable)

- Standardization
  - Many slight variations of C existed, and were incompatible
  - Committee formed to create a "unambiguous, machine-independent" definition
  - Standard created in 1989 (ANSI) , updated in 1999 (ISO)

1.2 The C Standard Library

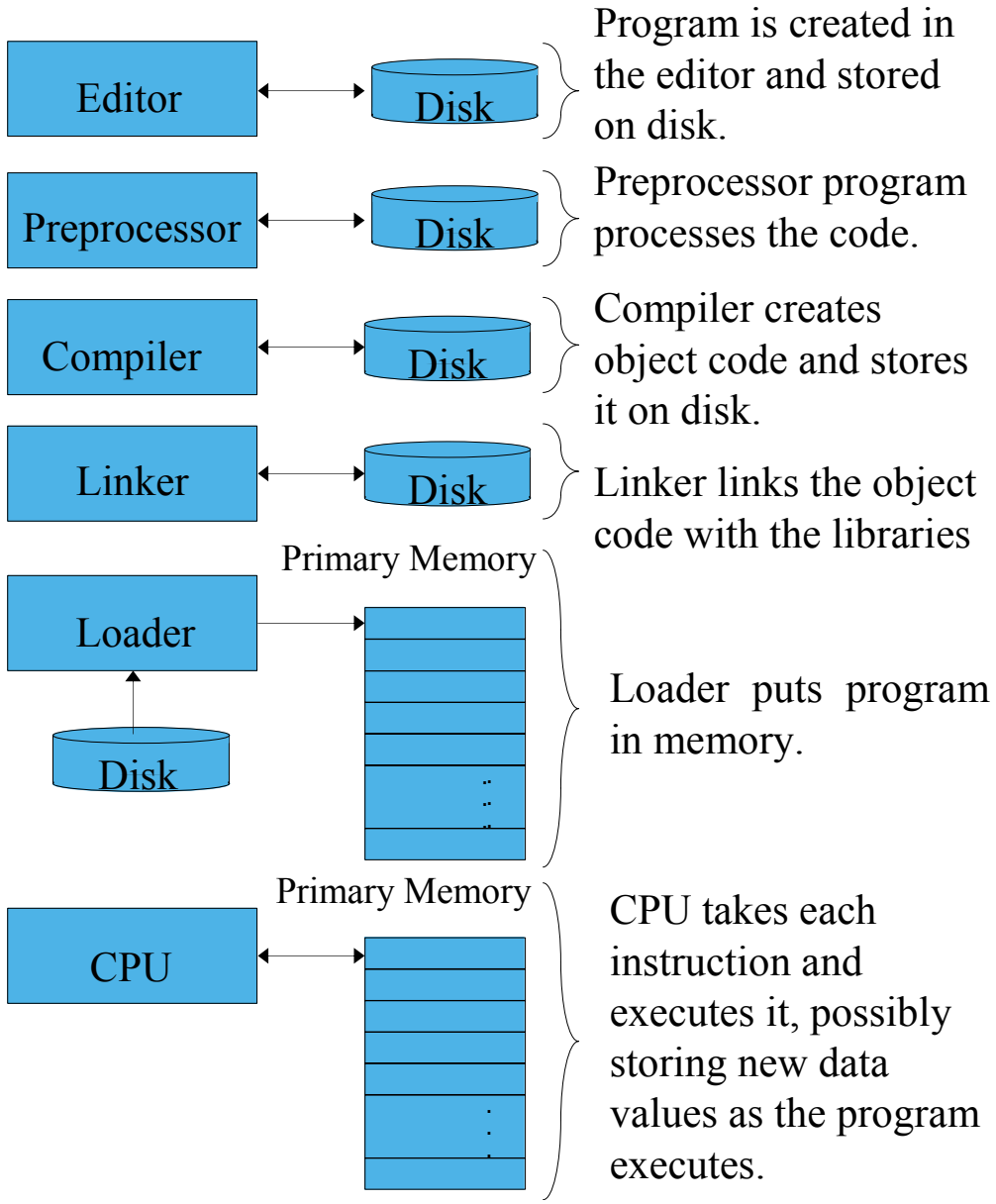
- C programs consist of pieces/modules called functions
  - A programmer can create his own functions
    - Advantage: the programmer knows exactly how it works
    - Disadvantage: time consuming
  - Programmers will often use the C library functions
    - Use these as building blocks
  - Avoid re-inventing the wheel
    - If a premade function exists, generally best to use it rather than write your own
    - Library functions carefully written, efficient, and portable

1.3 C++

- C++ Language
  - Superset of C developed by Bjarne Stroustrup at Bell Labs
  - Extends the C, and provides object-oriented capabilities
  - Object-oriented design is very powerful
  - Dominant language in industry and academia
- Learning C++
  - Because C++ includes C, it is best to master C, then learn C++

1.4 Basics of a Typical C Program Development Environment

- Phases of C Programs:
  1. *Edit*
  2. *Preprocess*
  3. *Compile*
  4. *Link*
  5. *Load*
  6. *Execute*



2.1 A Simple C Program: Printing a Line of Text

```
1    /* Fig. 2.1: fig02_01.c
2       A first program in C */
3    #include <stdio.h>
4
5    /* function main begins program execution */
6    int main()
7    {
8       printf( "Welcome to C!\n" );
9
10      return 0; /* indicate that program ended successfully */
11
12   } /* end function main */
```

```
Welcome to C!
```

Comments

- Text surrounded by `/*` and `*/` is ignored by computer
- Used to describe program

- `#include <stdio.h>`
  - Preprocessor directive
    - Tells computer to load contents of a certain file
  - `<stdio.h>` allows standard input/output operations

2.1 A Simple C Program: Printing a Line of Text

- `int main()`
  - C++ programs contain one or more functions, exactly one of which must be `main`
  - Parenthesis used to indicate a function
  - `int` means that `main` "returns" an integer value
  - Braces (`{` and `}`) indicate a block
    - The bodies of all functions must be contained in braces

2.1 A Simple C Program: Printing a Line of Text

- `printf( "Welcome to C!\n" );`
  - Instructs computer to perform an action
    - Specifically, prints the string of characters within quotes (`" "`)
  - Entire line called a statement
    - All statements must end with a semicolon (`;`)
  - Escape character (`\`)
    - Indicates that printf should do something out of the ordinary
    - `\n` is the newline character

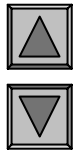
2.1 A Simple C Program: Printing a Line of Text

Escape Sequence	Description
`\n`	Newline. Position the cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Insert a backslash character in a string.
`\"`	Double quote. Insert a double quote character in a string.

Fig. 2.2 Some common escape sequences.

2.1 A Simple C Program: Printing a Line of Text

- `return 0;`
  - A way to exit a function
  - `return 0`, in this case, means that the program terminated normally
- Right brace `}`
  - Indicates end of `main` has been reached
- Linker
  - When a function is called, linker locates it in the library
  - Inserts it into object program
  - If function name is misspelled, the linker will produce an error because it will not be able to find function in the library



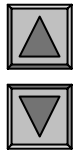
Outline

fig02_03.c

```
1  /* Fig. 2.3: fig02_03.c
2     Printing on one line with two printf statements */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "Welcome " );
9     printf( "to C!\n" );
10
11    return 0; /* indicate that program ended successfully */
12
13 } /* end function main */
```

Program Output

```
Welcome to C!
```



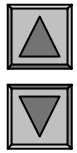
Outline

fig02_04.c

```
1  /* Fig. 2.4: fig02_04.c
2     Printing multiple lines with a single printf */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     printf( "Welcome\nto\nC!\n" );
9
10    return 0; /* indicate that program ended successfully */
11
12 } /* end function main */
```

```
Welcome
to
C!
```

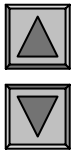
Program Output



Outline

fig02_05.c

```
1  /* Fig. 2.5: fig02_05.c
2     Addition program */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     int integer1; /* first number to be input by user  */
9     int integer2; /* second number to be input by user */
10    int sum;      /* variable in which sum will be stored */
11
12    printf( "Enter first integer\n" );  /* prompt */
13    scanf( "%d", &integer1 );           /* read an integer */
14
15    printf( "Enter second integer\n" ); /* prompt */
16    scanf( "%d", &integer2 );           /* read an integer */
17
18    sum = integer1 + integer2;          /* assign total to sum */
19
20    printf( "Sum is %d\n", sum );       /* print sum */
21
22    return 0;  /* indicate that program ended successfully */
23
24 } /* end function main */
```



Outline

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Program Output

2.2 Another Simple C Program: Adding Two Integers

- As before
  - Comments, `#include <stdio.h>` and `main`
- `int integer1, integer2, sum;`
  - Definition of variables
    - Variables: locations in memory where a value can be stored
  - `int` means the variables can hold integers (`-1`, `3`, `0`, `47`)
  - Variable names (identifiers)
    - `integer1`, `integer2`, `sum`
    - Identifiers: consist of letters, digits (cannot begin with a digit) and underscores(_)
      - Case sensitive
  - Definitions appear before executable statements
    - If an executable statement references and undeclared variable it will produce a syntax (compiler) error

Variable Naming Examples

- **Invalid Variable Names:**

  ÖğrenciNum, Öğr Num, Ogr-Num, 4.Ogr

  $\sum$, α , θ , a^2, π

- **Valid Variable Names:**

  OgrenciNum, OgrNum, Ogr_Num, Ogr4

  Sum , alfa , teta , aSquare , Pi

2.2 Another Simple C Program: Adding Two Integers

- `scanf( "%d", &integer1 );`
  - Obtains a value from the user
    - `scanf` uses standard input (usually keyboard)
  - This `scanf` statement has two arguments
    - `%d` - indicates data should be a decimal integer
    - `&integer1` - location in memory to store variable
    - `&` is confusing in beginning – for now, just remember to include it with the variable name in `scanf` statements
  - When executing the program the user responds to the `scanf` statement by typing in a number, then pressing the *enter* (return) key

2.2 Another Simple C Program: Adding Two Integers

- = (assignment operator)
  - Assigns a value to a variable
  - Is a binary operator (has two operands)

```
sum = variable1 + variable2;
```

```
sum gets variable1 + variable2
```

  - Variable receiving value must be on left (target)

- Common mistake:
  - The following gives a compiler error, becuase the left of the assignment operator (=) must always be the target variable.

```
variable1 + variable2 = sum;
```

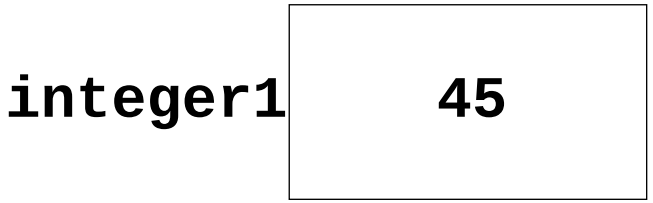
2.2 Another Simple C Program: Adding Two Integers

- `printf( "Sum is %d\n", sum );`
  - Similar to `scanf`
    - `%d` means decimal integer will be printed
    - `sum` specifies what integer will be printed
  - Calculations can be performed inside `printf` statements

```
printf( "Sum is %d\n", integer1 + integer2 );
```

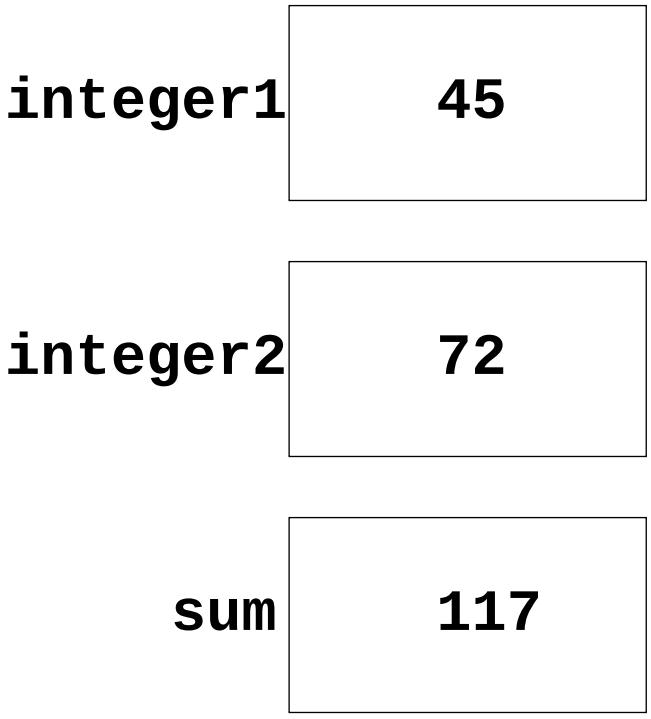
2.3 Memory Concepts

- Variables
  - Variable names correspond to locations in the computer's memory
  - Every variable has a name, a type, a size and a value
  - Whenever a new value is placed into a variable (through `scanf`, for example), it replaces (and destroys) the previous value
  - Reading variables from memory does not change them
- A visual representation



2.3 Memory Concepts

- A visual representation (continued)



Storing Data in Variables

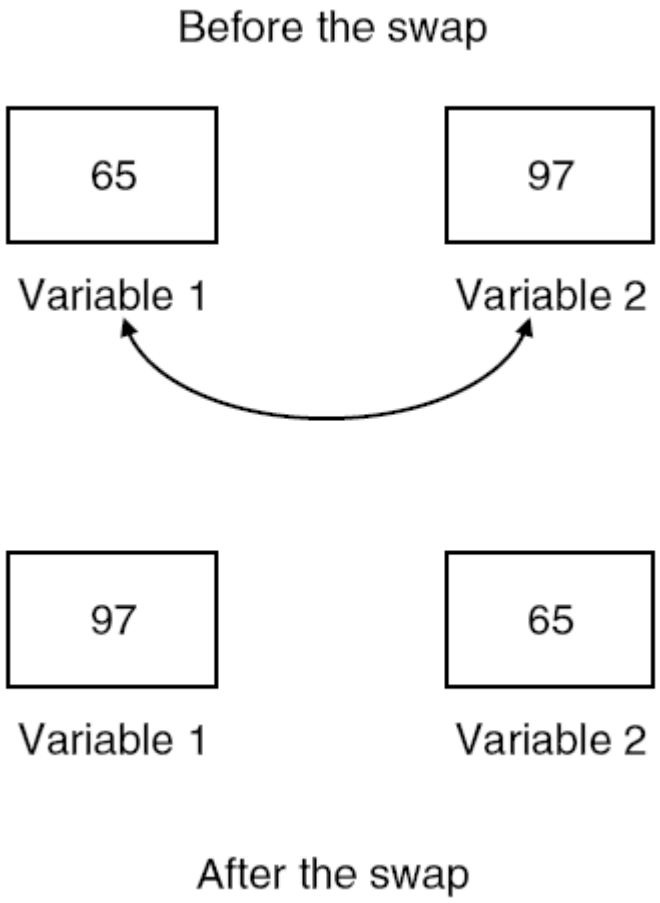
- You can think of a variable as if it were a box inside your computer holding a data value.
- The value might be a number, character, or string of characters.

- Data is stored inside memory locations (RAM) which are defined as variables.
- Instead of remembering a specific storage location (called an address), you only have to remember the name of the variables you define.
- The variable is like a box that holds data, and the variable name is a label for that box.

- Examples:

Swapping Variables (1)

Swapping values simply means replacing one variable's contents with another's and vice versa.



Swapping Variables (2)

- Suppose we assigned two variables named variable1 and variable2 with the following statements:

```
int  variable1 = 65 ;
int  variable2 = 97 ;
```

- Now we want to swap (i.e. exchange) their content values:

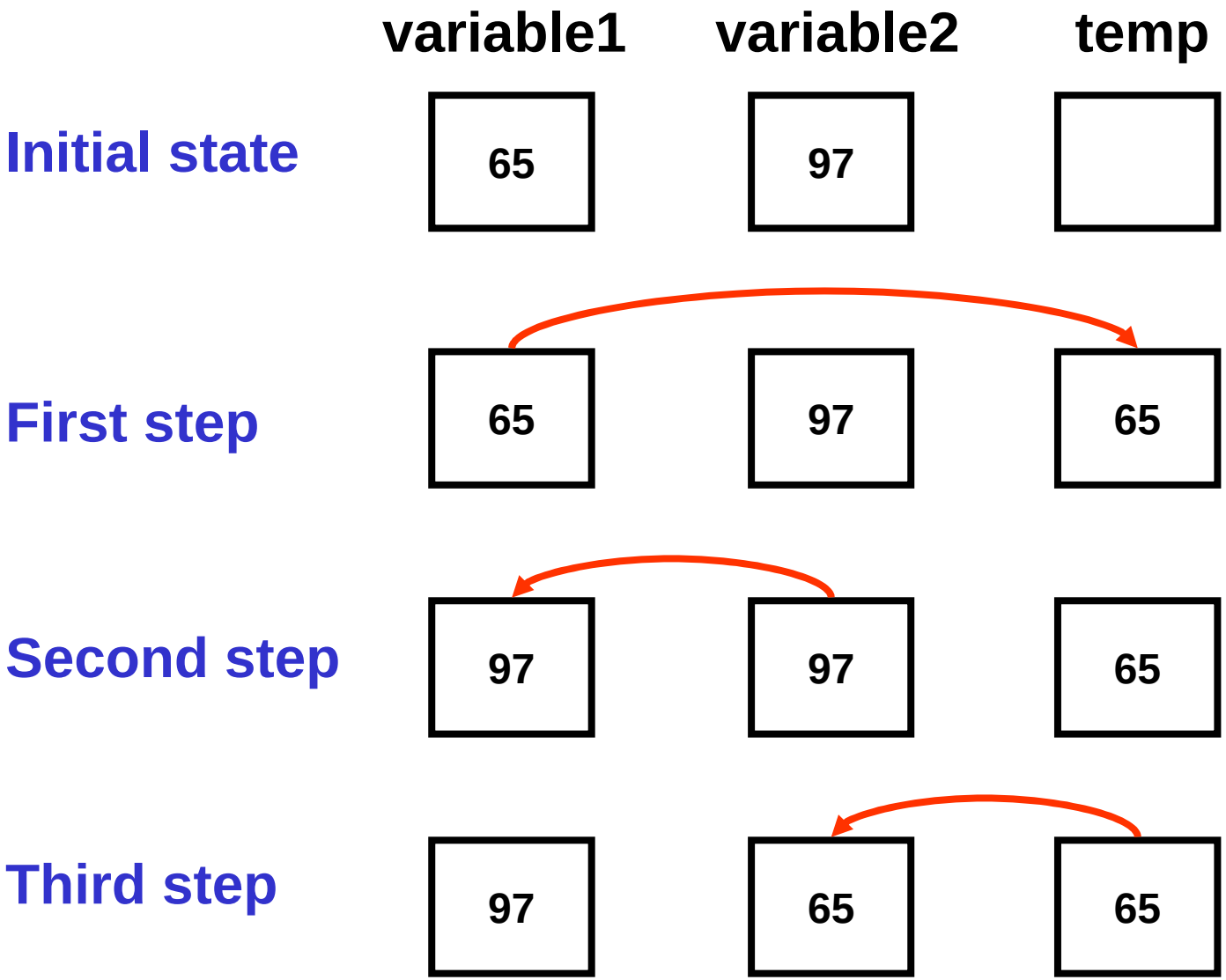
WRONG METHOD

```
variable1 = variable2;
variable2 = variable1;
```

CORRECT METHOD

```
int  temp;
temp = variable1;
variable1 = variable2;
variable2 = temp;
```

Swapping Variables (3)



Example: Swapping Correctly

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

int main()
{
int  variable1 = 65 ;
int  variable2 = 97 ;

int  temp;

printf("SWAP'TEN ONCE DEGISKENLER : %d %d \n\n", variable1, variable2);

temp = variable1;
variable1 = variable2;
variable2 = temp;

printf("SWAP'TEN SONRA DEGISKENLER : %d %d \n\n", variable1, variable2);

system("PAUSE");
return 0;
}
```

PROGRAM OUTPUT

```
SWAP'TEN ONCE DEGISKENLER : 65  97

SWAP'TEN SONRA DEGISKENLER : 97 65
```

2.4 Arithmetic

- Arithmetic calculations
  - Use `*` for multiplication and `/` for division
  - Integer division truncates remainder
    - `7 / 5` evaluates to `1`
  - Modulus operator(%) returns the remainder
    - `7 % 5` evaluates to `2`
- Operator precedence
  - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Use parenthesis when needed
  - Example: Find the average of three variables `a`, `b` and `c`
    - Do not use: `a + b + c / 3`
    - Use: `(a + b + c ) / 3.0`

2.4 Arithmetic

- Arithmetic operators:

C operation	Arithmetic operator	Algebraic expression	C expression
Addition	+	*f + 7*	`f + 7`
Subtraction	-	*p – c*	`p - c`
Multiplication	*	*b.r*	`b * r`
Division	/	*x / y*	`x / y`
Modulus	%	*r mod p*	`r % p`

- Rules of operator precedence:

Operator(s)	Operation(s)	Order of evaluation (precedence)
`()`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right.
`*`, `/`, or `%`	Multiplication,Division, Modulus	Evaluated second. If there are several, they are evaluated left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

2.4 Arithmetic

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

`2 * 5 is 10`

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

`10 * 5 is 50`

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

`3 * 5 is 15`

Step 4. `y = 50 + 15 + 7;` *(Leftmost addition)*

`50 + 15 is 65`

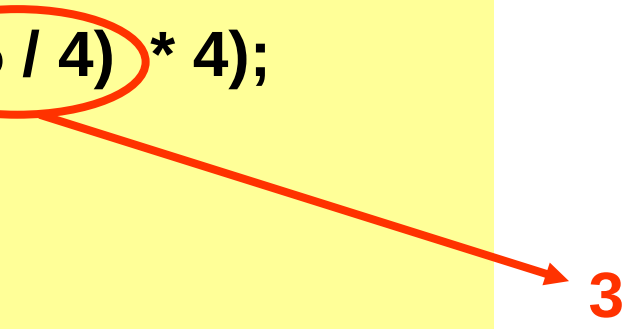
Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

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

Example: Integer Division

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

int main()
{
    // This will display 12, not 15!!

    printf("%d\n\n", (15 / 4) * 4);

   system("pause");
   return 0;
}
```



Example: Divisions

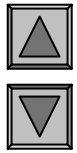
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int X = 15;

  printf("%d \n\n", X/2);  // 7

  printf("%f \n\n", X/2);  // 0.000000

  printf("%f \n\n", X/2.0);  // 7.500000

   printf("%f \n\n", (float) X / 2);  // 7.500000  (float) means typecasting

   printf("%.3f \n\n", (float) X / 2);  // 7.500

   printf("%.1f \n\n", (float) X / 2);  // 7.5

   //printf("%d \n\n", 60 / 0);  // Compiler error

   printf("%d \n\n", 60 / (X-15));  // Run-time error: Program will crash

   system("pause");
   return 0;
}
```

2.5 Decision Making: Equality and Relational Operators

- Executable statements
  - Perform actions (calculations, input/output of data)
  - Perform decisions
    - May want to print `"pass"` or `"fail"` given the value of a test grade
- `if` control statement
  - Simple version in this section, more detail later
  - If a condition is `true`, then the body of the `if` statement executed
    - `0` is `false`, non-zero is `true`
  - Control always resumes after the `if` structure
- Keywords
  - Special words reserved for C
  - Cannot be used as identifiers or variable names

2.5 Decision Making: Equality and Relational Operators

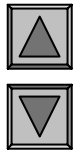
Standard algebraic equality operator or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality Operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational Operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
>=	>=	x >= y	x is greater than or equal to y
<=	<=	x <= y	x is less than or equal to y



Outline

fig02_13.c
(Part 1 of 2)

```
1  /* Fig. 2.13: fig02_13.c
2     Using if statements, relational
3     operators, and equality operators */
4  #include <stdio.h>
5
6  /* function main begins program execution */
7  int main()
8  {
9     int num1, /* first number to be read from user  */
10    int num2; /* second number to be read from user */
11
12    printf( "Enter two integers, and I will tell you\n" );
13    printf( "the relationships they satisfy: " );
14
15    scanf( "%d%d", &num1, &num2 ); /* read two integers */
16
17    if ( num1 == num2 ) {
18       printf( "%d is equal to %d\n", num1, num2 );
19    } /* end if */
20
21    if ( num1 != num2 ) {
22       printf( "%d is not equal to %d\n", num1, num2 );
23    } /* end if */
24
```



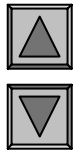
Outline

```
25    if ( num1 < num2 ) {
26       printf( "%d is less than %d\n", num1, num2 );
27    } /* end if */
28
29    if ( num1 > num2 ) {
30       printf( "%d is greater than %d\n", num1, num2 );
31    } /* end if */
32
33    if ( num1 <= num2 ) {
34       printf( "%d is less than or equal to %d\n", num1, num2 );
35    } /* end if */
36
37    if ( num1 >= num2 ) {
38       printf( "%d is greater than or equal to %d\n", num1, num2 );
39    } /* end if */
40
41    return 0;   /* indicate that program ended successfully */
42
43 } /* end function main */
```

fig02_13.c
(Part 2 of 2)

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

Program Output



Outline

Program Output (continued)

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

2.5 Decision Making: Equality and Relational Operators

Operators				Associativity
*	/	%		left to right
+	-			left to right
<	<=	>	>=	left to right
==	!=			left to right
=				right to left

Fig. 2.14 Precedence and associativity of the operators discussed so far.

2.5 Decision Making: Equality and Relational Operators

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

Fig. 2.15 C's reserved keywords.

Basic Data Types of Variables

- char
- int
- float
- double

Modifiers for Sign and Size

- unsigned
- signed *(by default)*
- short
- long *(by default)*

Data Type Ranges (1)

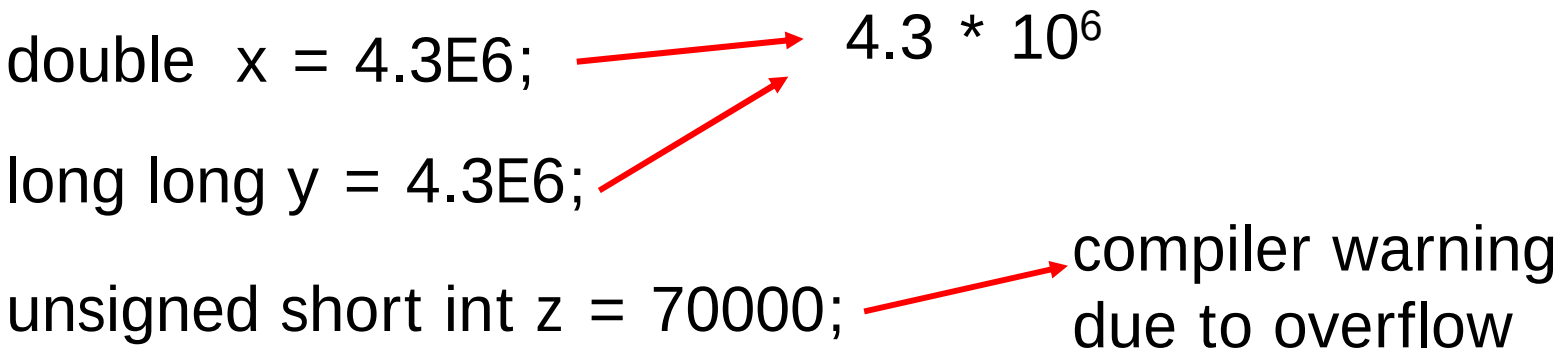
Keyword	Size in Bytes	Variable Type	Range
char	1	Character (or string)	-128 to 127
int	4	Integer	-2,147,483,648 to 2,147,483,647
long	4	Long integer	-2,147,483,648 to 2,147,483,647
long int	4	Long integer	-2,147,483,648 to 2,147,483,647
short	2	Short integer	–32,768 to 32,767
short int	2	Short integer	–32,768 to 32,767

Data Type Ranges (2)

Keyword	Size in Bytes	Variable Type	Range
unsigned char	1	Unsigned character	0 to 255
unsigned int	4	Unsigned integer	0 to 4,294,967,295
unsigned long	4	Unsigned long integer	0 to 4,294,967,295
unsigned short	2	Unsigned short integer	0 to 65,535

Data Type Ranges (3)

Keyword	Size in Bytes	Variable Type	Range
float	4	Single-precision floating-point (7 digits)	$-3.4 * 10^{-38}$ to $3.4 * 10^{38}$
double	8	Double-precision floating-point (15 digits)	$-1.7 * 10^{-308}$ to $1.7 * 10^{308}$



Example: Range Overflow

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   unsigned short int X, Y; // Length of these are 2 bytes (16-bit) each
   X = 65535;  // Maximum possible value for unsigned short integer numbers
   Y = X + 4;  // Overflow is expected here (Y will be 3, instead of 65539)
   printf("SONUC = %d \n\n", Y);  // It will display 3 !!
   system("pause");
}
```