SEQUENTIAL STRUCTURE

Erkut ERDEM
Hacettepe University

October 2010

History of C

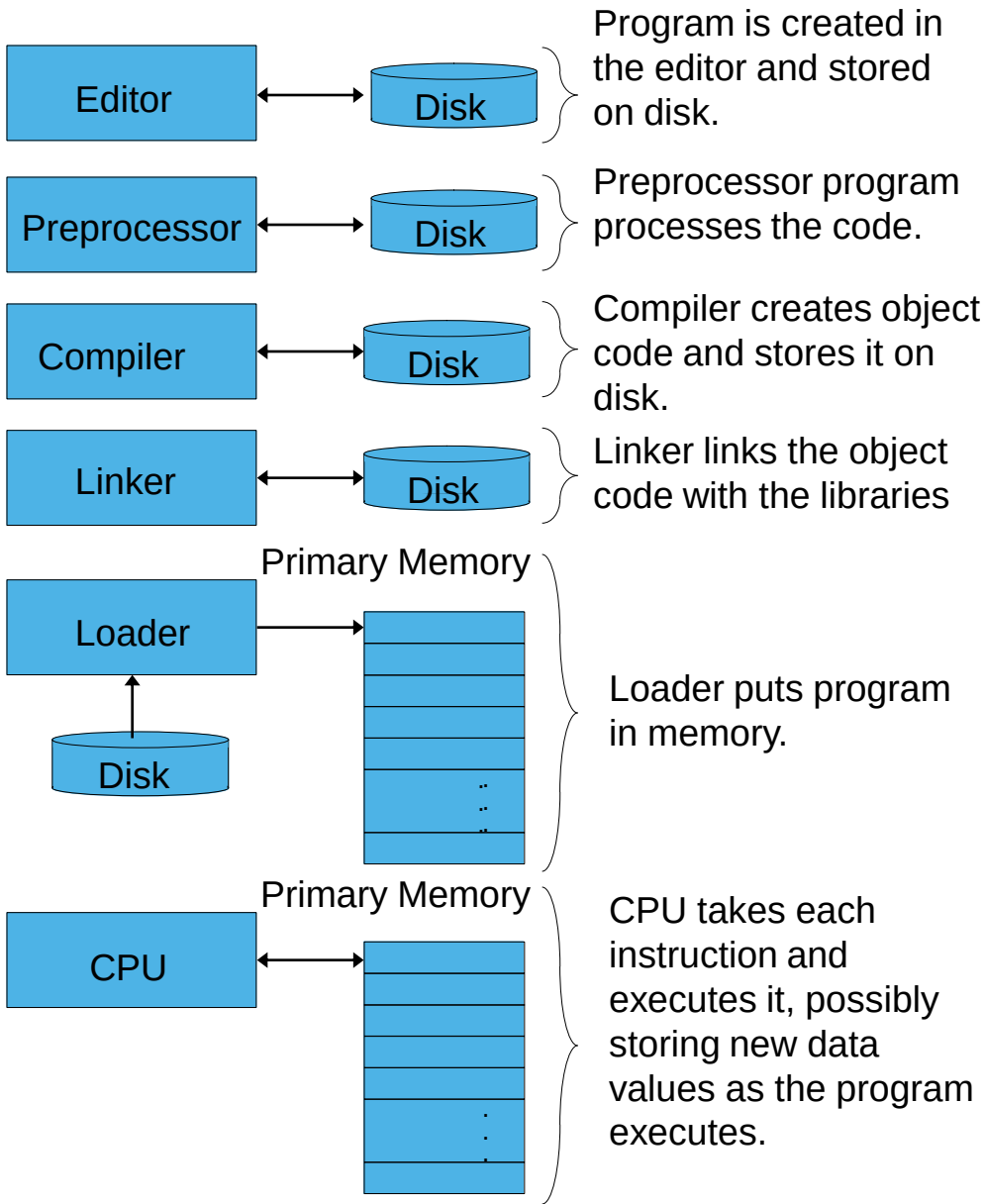
- C
  - Developed by by Denis M. Ritchie at AT&T Bell Labs from two previous programming languages, BCPL and B
  - Used to develop UNIX
  - Used to write modern operating systems
  - Hardware independent (portable)
- Standardization
  - Many slight variations of C existed, and were incompatible
  - Committee formed to create a "unambiguous, machine-independent" definition
  - Standard created in 1989, updated in 1999

The C Standard Library

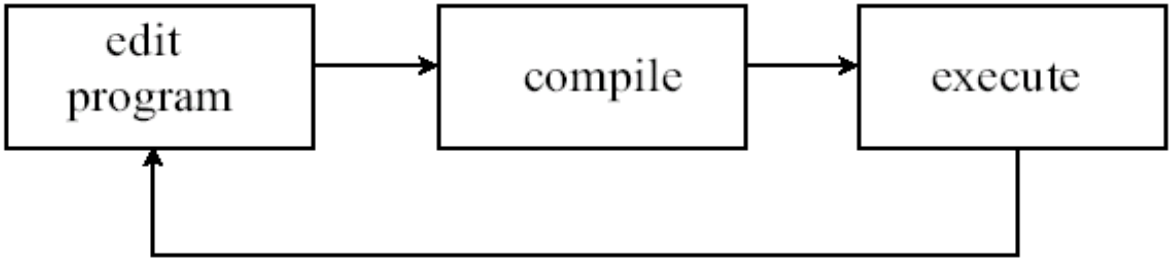
- C programs consist of pieces/modules called functions
  - A programmer can create his own functions
    - Advantage: the programmer knows exactly how it works
    - Disadvantage: time consuming
  - Programmers will often use the C library functions
    - Use these as building blocks
  - Avoid re-inventing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

Basics of a Typical C Program Development Environment

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



The Programming Process



The cycle ends once the programmer is satisfied with the performance of the program.

General Form of a C Program

Preprocessor directives

Declarations – variables (değişkenler)
(bildirimler) functions (işlevler)

`main` function {
declarations

statements
}

A Simple C Program

```
/* Hello world! Example */
#include <stdio.h>

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

```
Hello world!
```

A Simple C Program (Cont.)

- Comment Line
  (açıklama satırı)
  - Text surrounded by **/*** and ***/**
    is ignored by computer
  - Used to describe program

```
/* Hello world! Example */
#include <stdio.h>

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

- **#include <stdio.h>**
  - Preprocessor directive (ön işleyici komutu)
    - Tells computer to load contents of a certain file
  - **<stdio.h>** allows standard input/output operations

    Standart başlık kütüğü (header file)

A Simple C Program (Cont.)

- **`int main(void)`**
  - 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
  - **`void`** indicates that the function takes no arguments

```
/* Hello World! Example */
#include <stdio.h>

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

- Braces (`{` and `}`) indicate a block
  - The bodies of all functions must be contained in braces

A Simple C Program (Cont.)

- **`printf("Hello world!\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

```
/* Hello World! Example */
#include <stdio.h>

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

A Simple C Program (Cont.)

- **`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

```
/* Hello World! Example */
#include <stdio.h>

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

Lexical Elements

- **Token:** the smallest element of a program that is meaningful to the compiler

- Kinds of tokens in C:
  - Keywords (Anahtar Kelimeler)
  - Identifiers (Tanımlayıcılar)
  - Constants/Literals (Sabitler)
  - Operators (İşleçler)
  - Punctuators (Noktalama İşaretleri)

Keywords

- 32 words defined as keywords in C
- have predefined uses and cannot be used for any other purpose in a C program

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

Identifiers (Tanımlayıcılar)

- A sequence of letters, digits, and the special character '_' satisfying:

  identifier = α {α + #}*

  with α = {A,…,Z,a,…,z,_ }, #={0,…,9}, and
  * means "0 or more"

- Case-sensitive
  e.g. `Ali` and `ali` are two different identifiers.

- Identifiers are used for:
  - Variable names (değişken isimleri)
  - Function names (işlev isimleri)
  - Macro names (makro isimleri)

Identifiers (Cont.)

- Sample valid identifiers

```
x
a1
_xyz_33
integer1
Double
```

- Sample invalid identifiers

```
xyz.1
gx^2
114West
int
pi*r*r
```

Variable Declarations

- **Variables:** locations in memory where a value is stored
- Variable declarations must appear before executable statements.
- Variables must be declared before use.
  - a syntax (compile-time) error if these are violated
- Every variable has a name, a type, a size and a value

Basic Datatypes in C

- Integer `int`
- Character `char`
- Floating Point `float`
- Double precision floating point `double`

- Datatype modifiers
  - `signed` / `unsigned` (for `int` and `char`)
  - `short` / `long`

Basic Datatypes in C (Cont.)

Type	**Typical Range of Values**
`signed char (8 bits)`	-127 to +127
`unsigned char`	0 to 255
`short int (16 bits)`	-32,767 to +32,767
`unsigned short int`	0 to 65,535
`int (32 bits)`	-2,147,483,647 to +2,147,483,64
`unsigned int`	0 to 4,294,967,295
`long int (32-64 bits)`	-2,147,483,647 to +2,147,483,647
`unsigned long int`	0 to 4,294,967,295
`float`	$\sim 10^{-37}$ to $\sim 10^{38}$
`double`	$\sim 10^{-307}$ to $\sim 10^{308}$
`long double`	$\sim 10^{-4931}$ to $\sim 10^{4932}$

Variable Declarations (Cont.)

- A declaration consists of a data type name followed by a list of (one or more) variables of that type:

```
char c;
int ali, bora;
float rate;
double trouble;
```

- A variable may be initialized in its declaration.

```
char c = 'a';
int a = 220, b = 448;
float x = 1.23e-6; /*0.00000123*/
double y = 27e3; /*27,000*/
```

- Variables that are not initialized may have garbage values.
- Whenever a new value is placed into a variable, it replaces the previous value
- Reading variables from memory does not change them

Constants

C manipulates various kinds of values.

- integer constants: `0, 37, 2001`
- floating-point constants: `0.8, 199.33, 1.0`
- character constants: `'a', '5', '+'`
- string constants: `"a", "Monday"`

Common Escape Sequences

\a	audible alarm	\b	backspace
\n	newline	\r	carriage return
\t	horizontal tab	\f	form-feed
\\	backslash	\"	double quote

Operators

- Arithmetic operators
- Assignment operator
- Logical operators (later on; in the lecture on selective structure)

Arithmetic Operators

- For arithmetic calculations
  - Use `+` for addition, `-` for substraction, `*` for multiplication and `/` for division
  - Integer division truncates remainder
    - `7 / 5` evaluates to `1`
  - Modulus operator(`%`) returns the remainder
    - `7 % 5` evaluates to `2`

- Arithmetic operators associate left to right.
- Operator precedence
  - Example: Find the average of three variables a, b and c
    - Do not use: a + b + c / 3
    - Use: (a + b + c) / 3

Arithmetic Operators (Cont.)

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

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.

Arithmetic Operators (Cont.)

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`

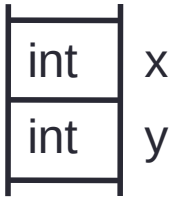
Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

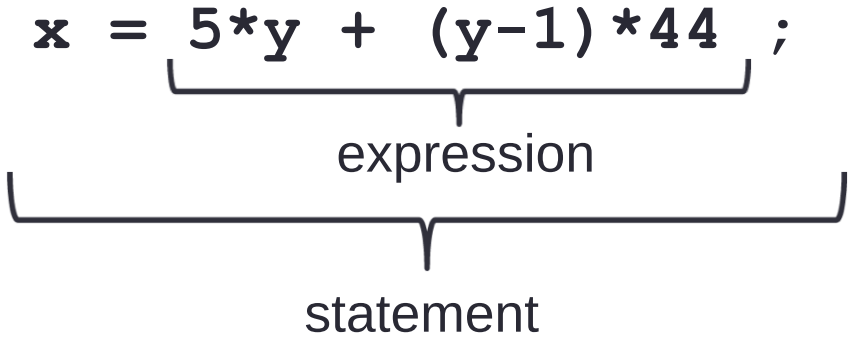
Step 6. `y = 72;` *(Last operation—place 72 in y)*

Assignment Operator

- `int x,y; /* variable declarations */`

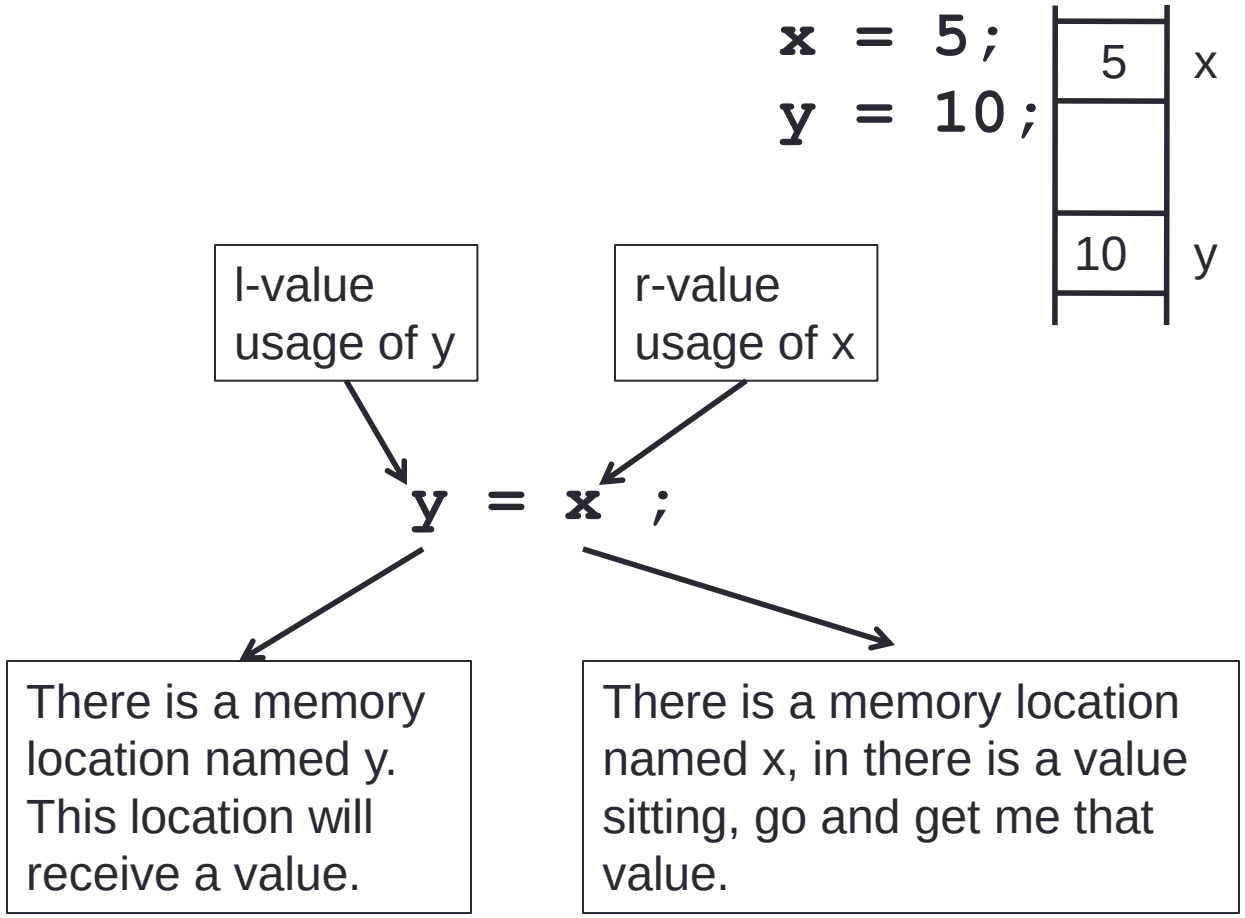


- `variable = expression ;`
- expressions:
  - operations
  - variables
  - constants
  - function calls



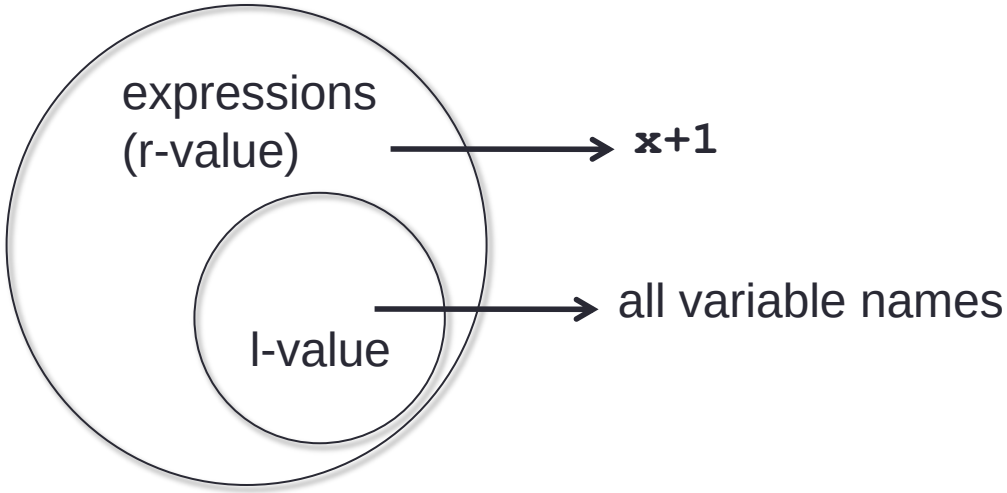
- Precedence of the assignment operator is lower than the arithmetic operators

l-value vs. r-value



l-value vs. r-value (Cont.)

- `x+1 = 3;` invalid l-value expression



- l-value usages MUST refer to a fixed position in the MEMORY.

Addition of two numbers

```
/* This programs adds the two integers that it reads */
#include <stdio.h>

int main (void)
{
  int num1, num2;                /* declarations */
  printf("Enter first integer.\n");   /* prompt */
  scanf("%d", &num1);                 /* read an integer*/

  printf("Enter second integer.\n");  /* prompt */
  scanf("%d", &num2);                 /* read an integer */

  num1 = num1 + num2;                 /* assignment of sum */

  printf("Sum is %d.\n", num1);       /* print sum */
  return 0;                           /* program ended
}                                  successfully */
```

Addition of two numbers – Sample Runs

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

```
Enter first integer.
30
Enter second integer.
12
Sum is 42.
```

Addition of two numbers - Analysis

- **`int num1, num2;`**
  - Variable declarations
  - **`int`** means these variables can hold integers
- **`scanf( "%d", &num1);`**
  - 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
    - **`&`** refers to the address to store the value
    - **`&num1`** - location in memory where the variable **`num1`** is stored
  - When executing the program the user responds to the **`scanf`** statement by typing in a number, then pressing the *enter* (return) key

Addition of two numbers - Analysis

- **`num1 = num1 + num2;`**
  - performs the arithmetic operation and then assigns the resulting value to the variable **`num1`**

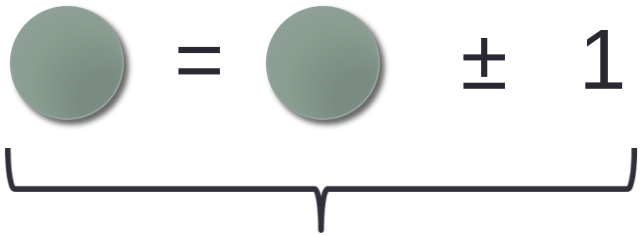
- **`Printf("Sum is %d\n", num1);`**
  - similar to **`scanf`**
    - **`%d`** means decimal integer will be printed
    - **`num1`** specifies what integer will be printed

- Calculations can be performed inside **`printf`** statements

```
printf( "Sum is %d\n", num1+ num2);
```

Increment and Decrement Operators

- `i = i + 1 ;`

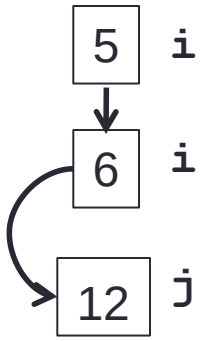


- `i = i + 1` **OR** `++i`
- `++(i-3)` invalid
- `++(++i)` invalid

Increment and Decrement Operators

- Post-incrementation/decrementation exists!

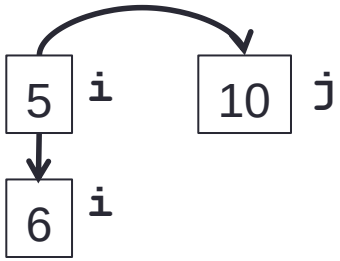
- `i=5;`
  `j=(++i)*2;`
  `(i=i+1)*2`




```
i=5;
j=2*i++ - i;
```

- ambiguous
- compiler-dependent

- `i=5;`
  `j=(i++)*2;`



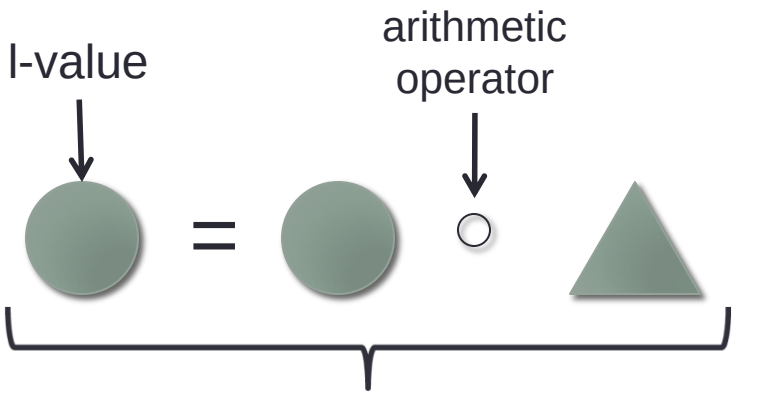
Increment and Decrement Operators

```
/*** increment and decrement expressions ***/
#include <stdio.h>

int main(void)
{
    int a =0 , b = 0, c = 0;

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

```
2 1 1
2 2 2
5 3 3
5 2 2
```

Compound Assignment Operator

- `sum = sum + x ;`



- `sum += x;`

Nested Assignments

- Multiple assignments in one statement.
- Assignment operators are right-associative.

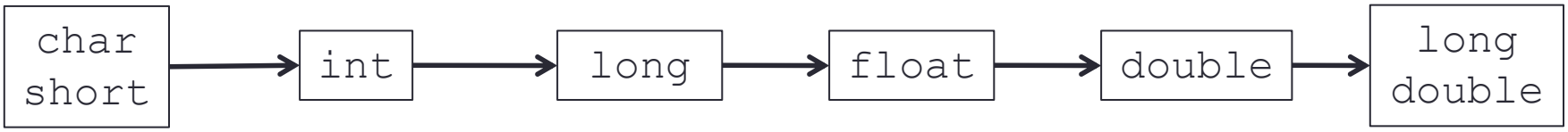
- ```
  x = y = z = 0;
      x = (y = (z = 0));
  ```

- ```
  x -= y = z;
      x -= (y = z);
  ```

- ```
  x = y += z;
      x = (y += z);
  ```
  ```

Input and Output

- `printf`(format string, arg1, arg2, …);

- The format string is composed of zero or more directives:
  - ordinary characters (not %), which are copied unchanged to the output stream
  - **e.g.** `printf("Hello World!\n");`
  - conversion specifications, each of which results in fetching zero or more subsequent arguments.
    - `%c`: the argument is taken to be a single character
    - `%d`: the argument is taken to be an integer
    - `%f`: the argument is taken to be a floating point (float or double)
      `%s`: the argument is taken to be a string

Input and Output

- **Examples:**

```
int i = 2;
printf("%d \n", i);
printf("%d \n", 20*i);


float c = 20;
printf("%f centigrade = %f %s\n",
            c, 1.8*c+32,"Fahrenheit");
```

Input and Output

- `scanf` (format string, arg1, arg2, …);
- The format string consists of a sequence of directives which describe how to process the sequence of input characters.
- A directive is one of the following:
  - A sequence of white-space characters
  - An ordinary character (i.e., one other than white space or '%')
  - A conversion specification, which commences with a '%' (percent) character
    - `%c`: a single character is expected in the input
    - `%d`: an integer is expected in the input
    - `%f`: a floating point is expected in the input
    - ..
- Each argument must point to the variable where the results of input are to be stored.

Input and Output

- **Example:**

```
#include<stdio.h>
int main(void)
{
  float principal, rate, interest;
  int years;

  printf("principal, rate, and years? ");
  scanf("%f %f %d", &principal, &rate, &years);

  rate /= 100;
  interest = principal * rate * years;
  printf("interest = %f\n", interest);
  return 0;
}
```

Type Conversion and Casting

- In an operation, if operands are of mixed data types, the compiler will convert one operand to agree with the other using the following hierarchy structure:

  `long double` (highest)

  `double`

  `float` `long`

  `int`

  `char/short` (lowest)



Type Conversion and Casting (Cont.)

- implicit (automatic) type conversion
  - done automatically by the compiler whenever data from different types is intermixed.

  - ```
    int i;
    double x = 17.7;
    i = x;
    ```

    i=17

  - ```
    float x;
    int i = 17;
    x = i;
    ```

    x=17.0

Type Conversion and Casting (Cont.)

- **Casting:** explicit type conversion

(type) any r-value

```
#include <stdio.h>

int main(void)
{
  int total_score, num_students;
  float average;

  printf("Enter sum of scores: ");
  scanf("%d",&total_score);
  printf("Enter number of students: ");
  scanf("%d",&num_students);

  average=total_score/num_students;
  printf("Average score (no type casting) is %.2f\n", average);

  average=(float)total_score/(float)num_students;
  printf("Average score (with type casting) is %.2f\n", average);
  return 0;
}
```

```
Enter sum of scores: 333
Enter number of students: 4
Average score (no type casting) is 83.00
Average score (with type casting) is 83.25
```

Simple Macros

- C provides a **`#define`** directive to define symbolic names for constants:

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

  int main(void)
  {
        float radius;
        scanf("%f", &radius);
        printf("area = %f\n", PI * radius * radius);
        printf("circumference = %f\n", 2 * PI * radius);
        return 0;
  }
  ```
  ```

Exercises

- Given

```
int x=4, y=5, z=6;
```

  Evaluate the following:

  - `x -= y = z;`
  - `y *= z/x;`
  - `z += -++x - y++;`

- Given

```
int a=12, b=8, c=2;
```

  Evaluate the following:

```
b %= (a %= b) * c * c-b;
```

Exercises

- Given

```
int x=3;
```

What is the result of the following?

```
x = x*5/2.0 + 3/2;
```

- Given

```
double x = 3;
```

What is the value of `x` after the following statements are executed?

- `x = x*5/2;`
- `x = x*5/(int)2;`
- `x = (int)x*5/2;`

Exercises

- Write a program that reads 3-digit 2 positive integer number. Firstly, you will find the sum of individual digits of the first number. After that, you will find the multiplication of individual digits of the second number. Finally, you will sum of these two results as an output.

  For example:

  inputs: 157 218

  157 → 1+5+7=13

  218 → 2*1*8=16

  output: 13+16=29