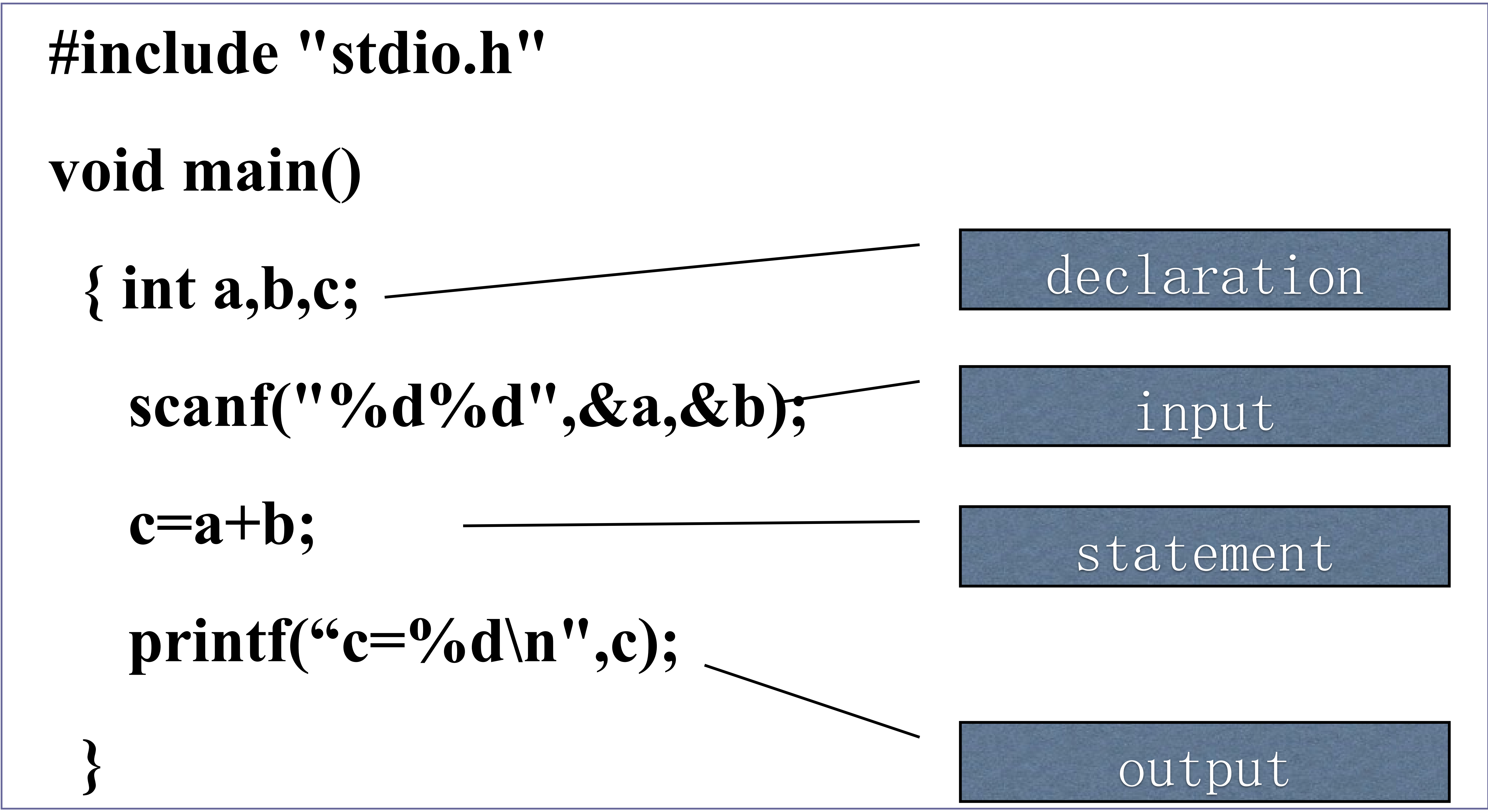
C Programming

03: Expression Statements

Program Structure

- A C program needs:
  - **an entry point (where it starts)**
    - the beginning of main()
  - **zero or more statements**
    - the contents of main()
  - **one or more exit points (where it ends)**
    - the return keyword

Usual Structure



```
#include "stdio.h"
void main()
 { int a,b,c;
   scanf("%d%d",&a,&b);
   c=a+b;
   printf(“c=%d\n",c);
 }
```

Usual Structure

- An imperative program almost always has:
  - Storage (for the data) - i.e. declarations
  - Access to useful functions - i.e. libraries
  - Input from a file, a device, or the terminal
    - in C, <stdio.h> or argc/argv (later)
  - Output to a file, a device, or the terminal
    - in C, <stdio.h>

Types of Statements

- C has four basic types of **statement**
  - Expression Statements
  - Compound Statements
  - Control Statements
  - Functions

  Expression statements are the simplest

  And the most common

Expression Statements

- An expression followed by ;
- (Nearly) everything in C is an expression
  - Even assignment (set A equal to B)
- An empty expression is still an expression
  - So this is valid (but silly & bug-prone):

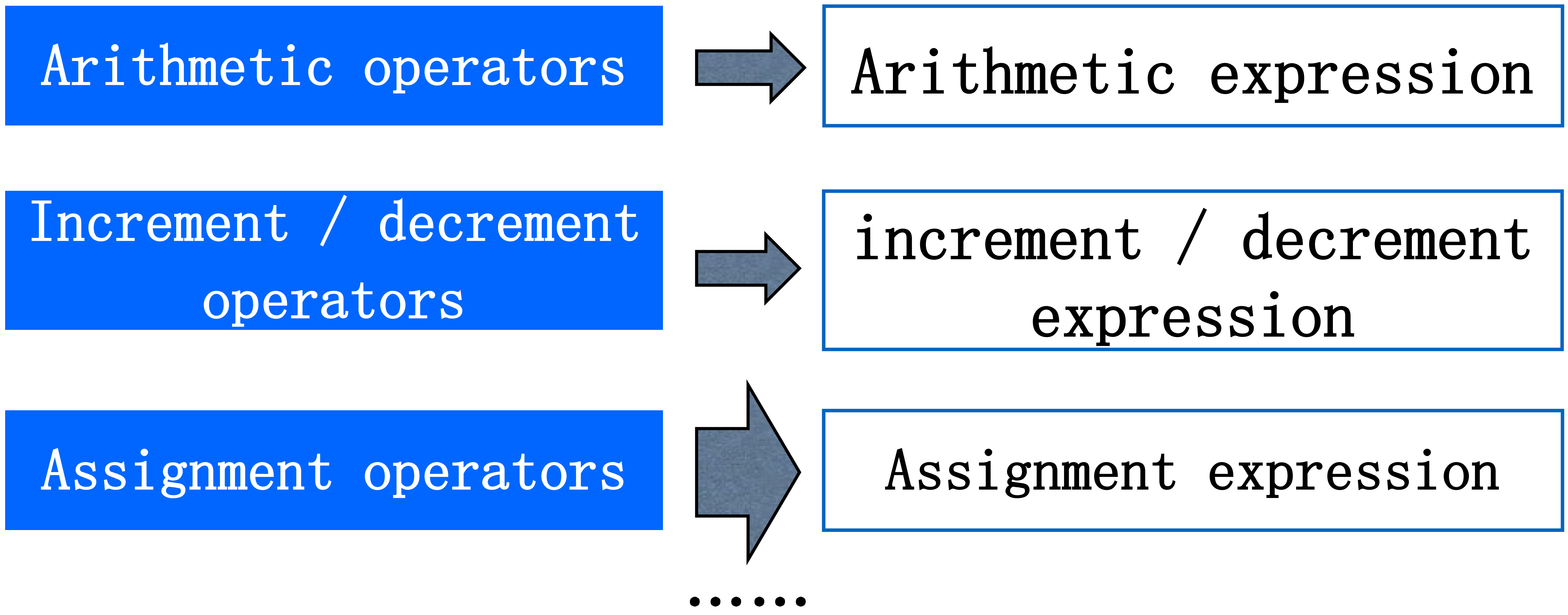
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Expression Rules

- An operand is a value (a noun)
- An operator is what you do with it (a verb)
  - They can change variables if they want
  - Typically, they change an operand
  - E.g.   the = (assignment) operator

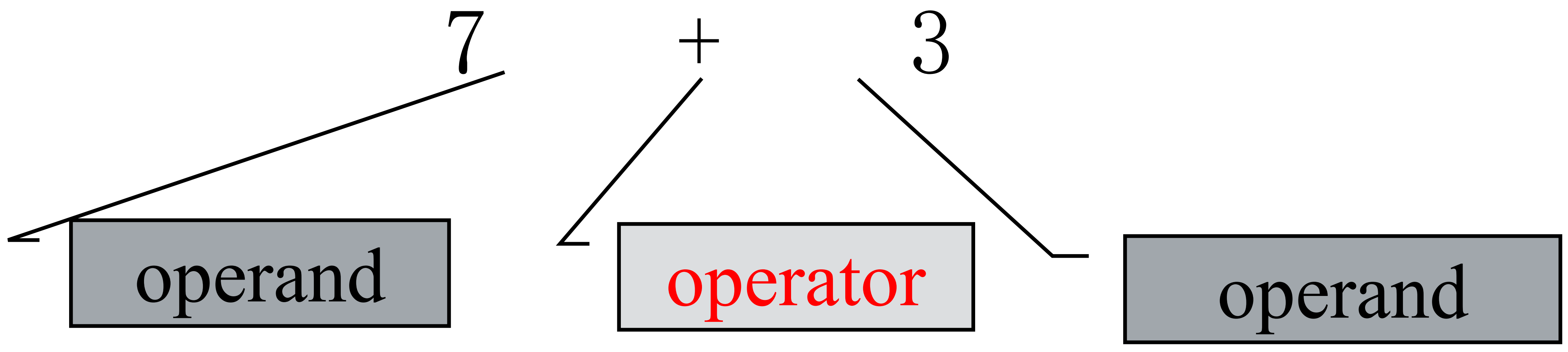
Expression Statements

- An expression is a combination of values and operations which evaluates to a value.



Expression

Example

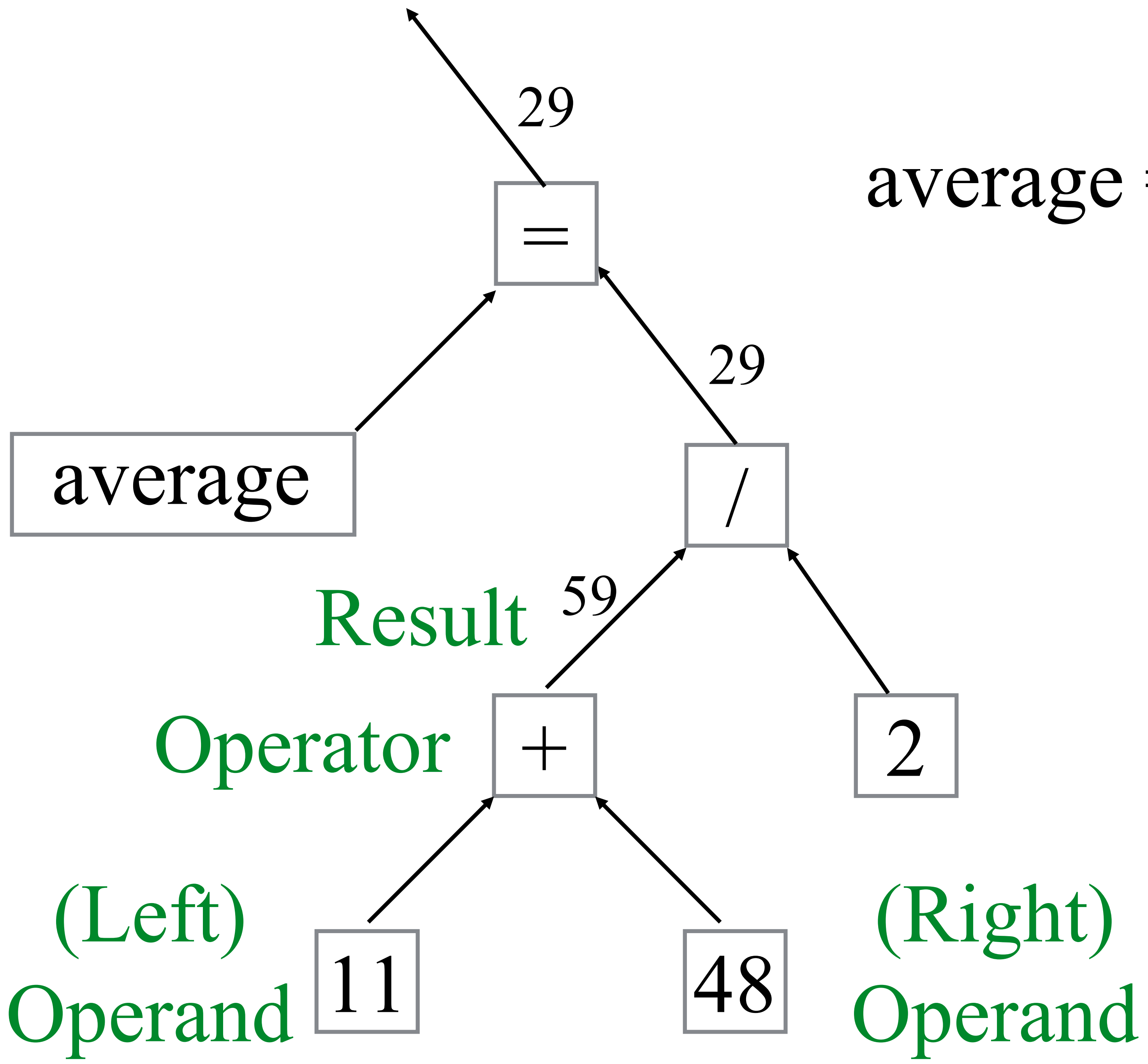


$4 * 6 + 9 * 3$

Operator & Operands

- An operator can have up to three operands
  - Unary: -1 +1
  - Binary: 1 - 2
  - Ternary: (later)
- An expression can be used as an operand
  - Which is how we build up expressions
  - Often shown as expression trees

An expression tree



average = (11+48) /2

Arithmetic operators

Unary:	
+A	Plus: Does nothing, returns A
-A	Minus: Returns the negative of A

Binary:	
A + B	Plus: returns A + B
A - B	Minus: returns A - B
A * B	Multiply: returns A * B
A / B	Divide: returns A / B
A % B	Modulus: returns remainder of A / B

Arithmetic operators

Tips:

1/2 = 0   1.0/2 = 0.5   7/2=**3**

5%3=2   1%2=0

Float vs. Integer

- Try not to mix floats and integers
- You can get confusing results
  - Especially for division
- When in doubt, use the **cast operator**
  - Just a type name in parentheses
- This tells the compiler what type you want

```
average = (float) sum / howMany;
```

Assignment Operators

- Left side variable is assigned the result, which is passed on
  - This makes x = y = z = 1 possible

A = B Sets A to B, returns A

Shorthand

- These shorthands do not introduce any new behaviors. Instead,
- they just provide a shorter way to write common patterns of existing things we have seen.

A += B	Sets A to A+B, returns A
A -= B	Sets A to A-B, returns A
A *= B	Sets A to A*B, returns A
A /= B	Sets A to A/B, returns A
A %= B	Sets A to A%B, returns A

Shorthand

Tips:

A += B	⟺	A = A + B
A -= B	⟺	A = A - B
A *= B	⟺	A = A * B
A /= B	⟺	A = A / B
A %= B	⟺	A = A % B

a*=b+c

B-=b*c

Increment Operators

- Incredibly convenient short forms:
  - Prefix increment / decrement:

    ++A **--A**

    - Equivalent to A += 1 and A -= 1
  - Suffix increment / decrement:

    A++ **A--**

    - these return the original version of A

Prefix / Suffix

- 70s CPUs had special increment operations
  - Faster than regular addition / subtraction
  - Now the same speed, but convenient
- ++A (prefix) is safer to learn
  - so text prefers it
- A++ (suffix) is arguably more useful
  - *very common idiom* for pointers

Prefix / Suffix

e.g. 1) int a=2, b;

b=a--;

Equivalent to-> b=a; a--;

result-> a= __1__, b= __2__ 。

2) int a=2, b;

b=++a;

Equivalent to-> ++a; b=a;

result-> a= __3__, b= __3__ 。

3) int a=2, b=1, c;

c=a++-b--;

Equivalent to-> c=a-b; a++; b--;

result-> a= __3__, b= __0__ , c= __1__ 。

Operator Priority

- We evaluate operators in a fixed order:
  - Parentheses () but not [] or {}
  - Exponents (not available in C)
  - Division / Multiplication
  - Addition / Subtraction
- Same as in school, but gets very complex
- When in doubt, use parentheses to be sure

Operator Priority

Priority	Operator
1	[ ] ( ) . –>
2	~ ! sizeof & * +(Unary) –(Unary)
3	(typename)
4	* / %
5	+ –
6	<< >>
7	> < >= <=
8	= = !=
9	&
10	^
11	|
12	&&
13	||
14	? :
15	= *= /= %= += –= <<= &= ^= |=
16	,

Input & Output

- output a letter

int putchar(int ch);

- input a letter

int getchar(void);

- #include "stdio.h"

e.g.

```
char ch;
ch=getchar();
putchar(ch);
```

Input & Output

- input a letter

int getche(void); int getch(void);

- #include “conio.h ”
- all the three functions can read a single character from a given input stream and returns the corresponding integer value (typically ASCII value of read character) on success

Input & Output

☐ Difference

getchar

- reads a single character from the keyboard and displays immediately on output screen after the enter key pressed

getche

- reads a single character from the keyboard and displays immediately on output screen without waiting for the enter key

getch

- the entered character is immediately returned without waiting for the enter key

Output with printf()

- printf() prints output based on a format string
- A string with conversion specifiers inside it

Syntax:

printf(“format”,expression,expression2…)

- Conversion specifiers use **%**
- Escape sequences Just like \ in character literals
- Expressions are substituted into them

printf(“2× %d is %d\n”, 15, 2 * 15);

Output with printf()

E.g.

- printf(“2× %d = %d\n”, 15, 2 * 15);

  Output: 2×15=30

- printf("We are students.\n");

  Output: We are students.

- printf("\n");

  Output:

  (just a newline)

Escape sequences

Escape sequences	function
\n	newline
\t	tabulator(TAB)
\v	vertical tab
\b	back space
\r	return
\f	form feed
\\	……
\'	……
\ddd	……
\xhh	……

Conversion Specification

%d:	**decimal integer (signed)**
%o:	**unsigned octal**
%x:	**hexadecimal**
%f:	**float**
%lf:	**double**
%c:	**character**
%s:	**string**
%%:	**the % character itself**

Conversion Prefixes

%ld:	decimal long
%lld:	decimal long long
%-f:	left-justified float
%8d:	decimal, padded to 8 characters
%-8d:	decimal, left-justified & padded
%8.3f:	float, 8 characters, 3 decimals

- And there are more
  - But that's what manuals are for!

Reading Numbers

- printf() has a counterpart called **scanf()**
  - it uses the same conversion specifiers
  - but variables need to have an **&** attached
  - we'll see why later

Syntax:

scanf("format",address1, address2…)

E.g.

scanf("%d %d", &a, &b);

- breaks down if input badly formatted

Conversion Specification

%d:	decimal integer (signed)
%o:	unsigned octal
%x:	hexadecimal
%f:	float
%lf:	double
%c:	character
%s:	string

Worked Example

Example for Printf ➡

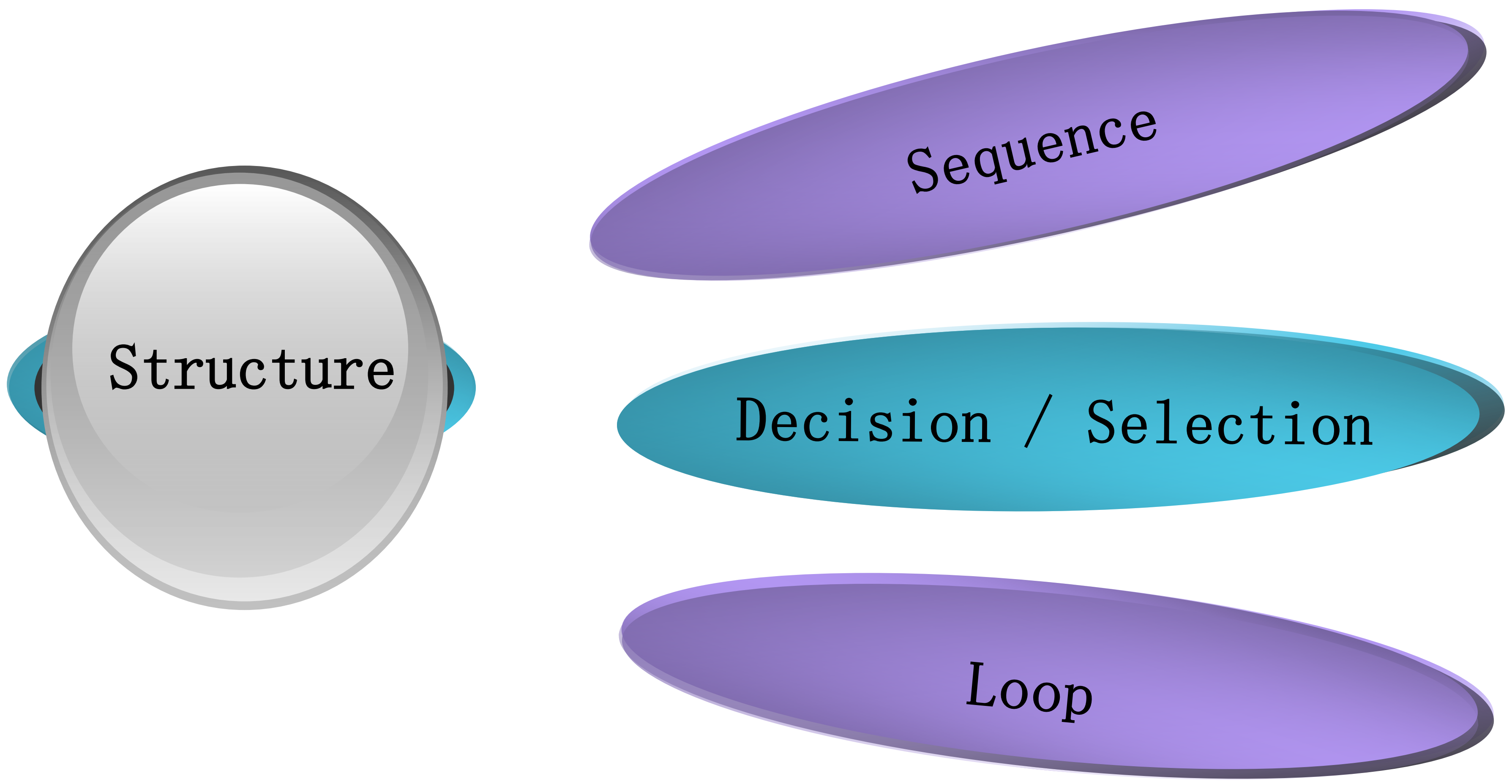
Programming Structure

1
- Sequence structure

2
- Selection structure

3
- Loop structure

Introduction of three structure



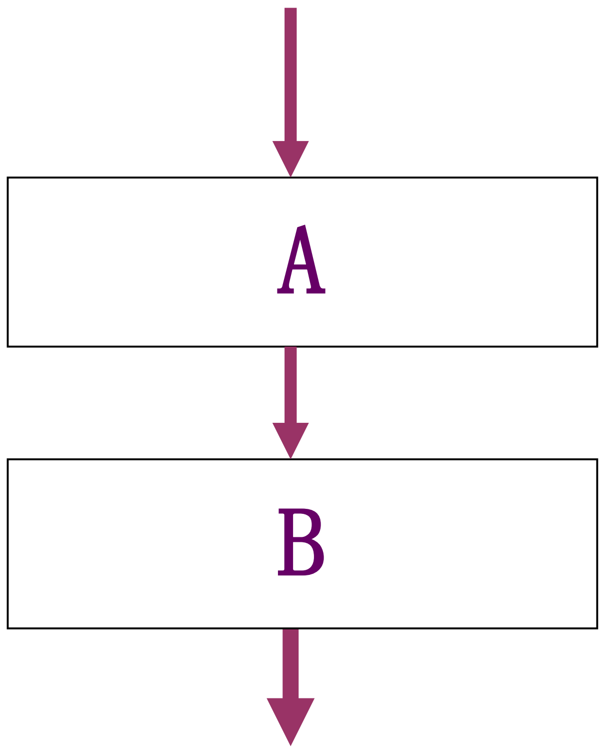
Loops have as purpose, to repeat a statement a certain number of times or while a condition is fulfilled

The sequence structure [illegible] leads to the next ordered [illegible] branches the [illegible]

a different [illegible] a condition predetermined order [illegible]

Introduction of three structure

- Sequence Structure
  - an action, or event, leads to the next ordered action in a predetermined order

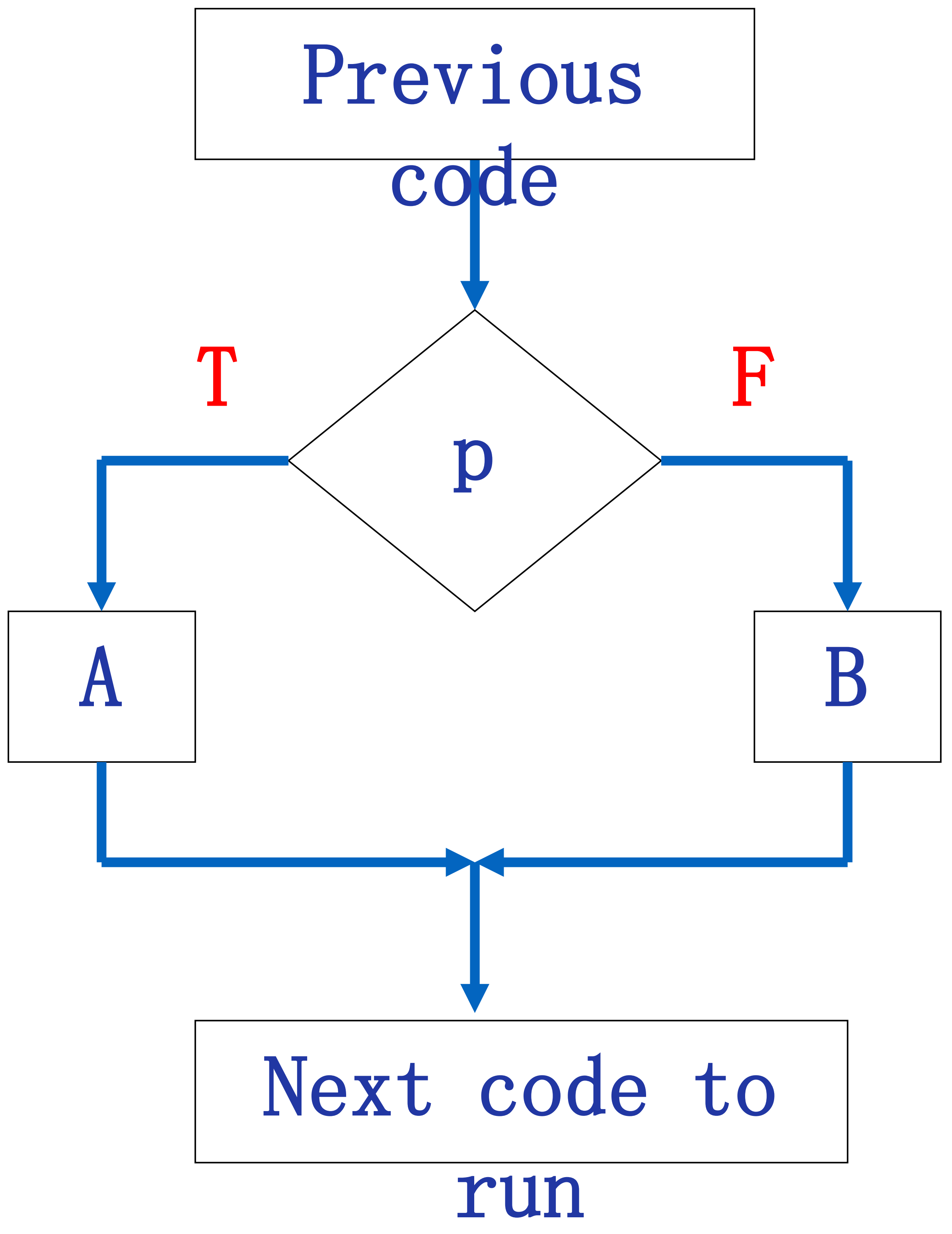


Introduction of three structure

- selection structure
  - In a selection structure, a question is asked, and depending on the answer, the program takes one of two courses of action, after which the program moves on to the next event.

Introduction of three structure

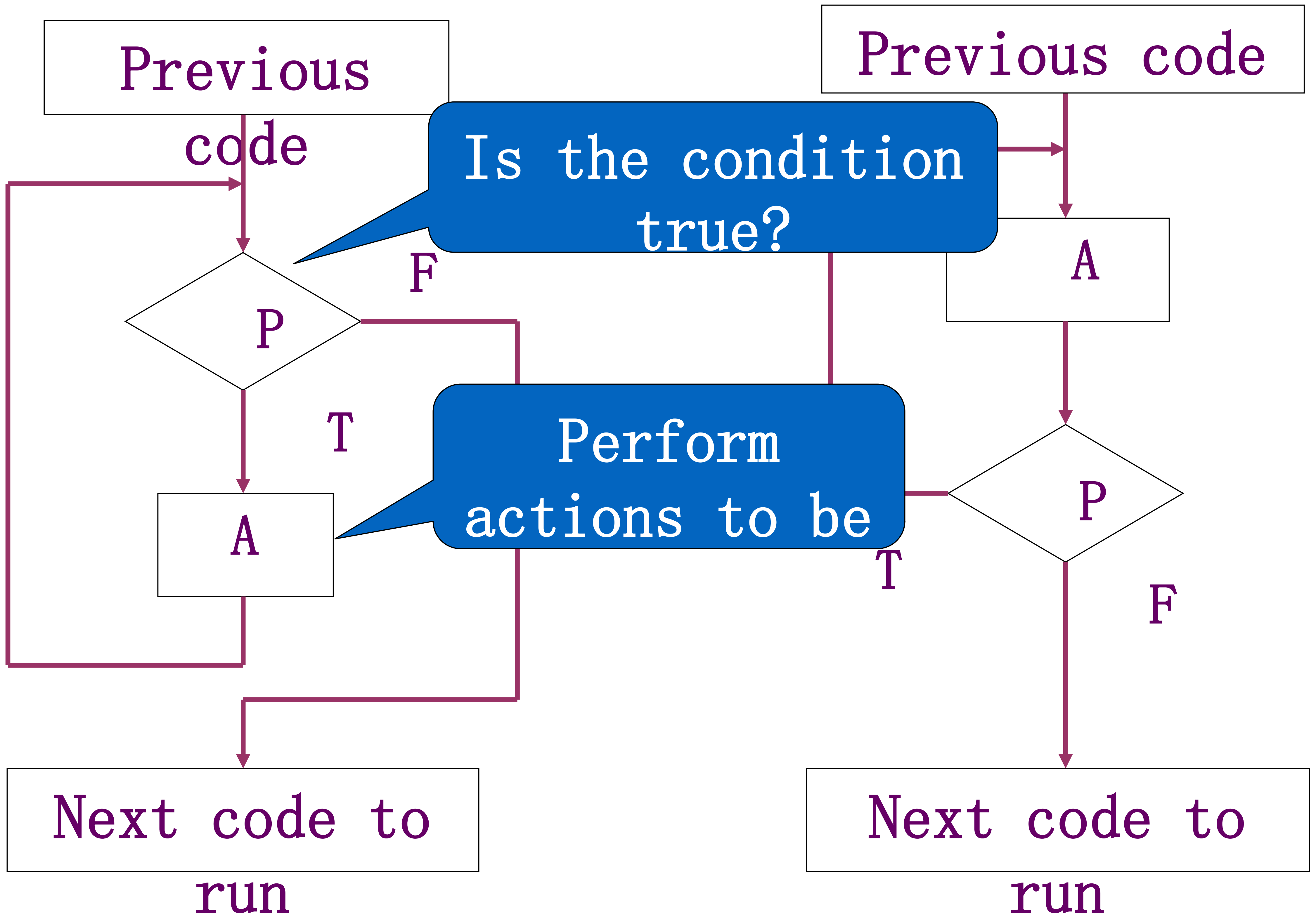
selection structure



Introduction of three structure

- Loop structure
- The Loop has two parts:
  - a condition that is tested for true or false value;
  - a statement or set of statements that is repeated as long as the condition is true.

Introduction of three structure



Sequence structure

1

- The Sequence structure

2

- Example

Sequence structure

[ex.1]

```
#include "stdio.h"
void main()
  {  int a,b;
     scanf("%d%d",&a,&b);
     printf("a=%d\n",a);
  }
```

Define the variable

Input the data

Output the data

Sequence structure

A sequence structure program usually includes four steps:

1. Define the data structure
2. Input data
3. Data calculating or processing
4. Output the result

Sequence structure

- 1. Define the data structure
- 2. Input data
- 3. Data calculating or processing
- 4. Output the result

3 data type: int, float (double), char

putchar(), getchar(), scanf(), printf()

+ - * / % ++ -- && || ^ ~ << >> [illegible]

int a; float b; char d;

scanf("%d",&a); printf("%lf",[illegible]);

---> expression

Sequence structure

[ex.2]

A circle with its radius which is input by user， compute the perimeter and the area.

```
main()
 { float  r, l, s,PI=3.14;   /* define the variables */
   scanf("%f",&r);          /* input the radium; */
   l=2*PI*r;                /* compute the perimeter */
   s=PI*r*r;                /* compute the area */
   printf("l =%.2f,  s=%.2f\n",l,s);
                /*output the perimeter  and the area */
  }
```

```
Input: 3
Output:
l=18.84, s=28.
26
```

Sequence structure

[ex.3]

C Centigrade , *F* Fahrenheit . Input a temperature in Fahrenheit, then convert it into a Centigrade.

$$C = \frac{5}{9}(F - 32)$$ (Conversion Formula)

```
#include ″stdio.h”
void main()
   { float C,F;
     scanf( “%f″,&F);
     C=5.0/9*(F–32);
     printf(″C=%.3f\n″,C);
   }
```

```
Input: 80
Output:
C=26.667
```

Exercise

- Input a three-digit number, to output each digit of the number.

  input 156   output 1 5 6

- Enter an uppercase letter， convert it to lowercase and output the letter.

  input A   output a

Exercise

- Read ： Microsoft Visual C++ （Text book）

Exercise

- 有三种货物A,B,C,单价分别是3元、4元和5元；分别输入三种货物的数量，计算总价并输出。
- 输入a,b,c三个参数值，求一元二次方程的解。（用if语句或if else语句实现）