Topic 3. C Basics (2). Theory

Data Types, Operators, Decision and Repetition Statements (2)

COMP ENG 2SH4

Principles of Programming

McMaster University, 2015

Instructor: Sorina Dumitrescu

Data Types

- Basic data types
  - **char** (for characters) – size: **1 byte** (8bits)
  - **int** (integers) – either 16 or 32 bits – implementation defined.
  - **short int (short), long int (long)**
  - **float** (single-precision floating points)
  - **double** (double-precision floating points)
  - **long double** (extended-precision floating point)
  - **signed/unsigned** -- apply to **char, int, short, long**
  - **unsigned** types - only positive values
  - **signed** types – positive and negative values

Data Types

- The storage size of integer and floating point types is implementation defined (each compiler can choose sizes appropriate for the hardware) subject to some constraints:
  - For **int** either 16 or 32 bits
  - For **short** at least 16 bits and no larger than for **int**
  - For **long** at least 32 bits and no smaller than for **int**
  - For unsigned integer types with storage size **b** bits, the possible values are in the range: $0 \sim 2^{b}-1$
  - For signed int types with storage size b bits, the possible values are in the range: $-(2^{b-1}-1) \sim 2^{b-1}-1$
  - C99 specifies additional types: **long long int**, **_Bool.**

Range of value for integer types, depending on the size in bits

Type Name	Size(bits)	Range of Values
short int (short)	16	-32,767 to 32,767
int	32	-2,147,483,647 to 2,147,483,647
long int (long)	32	-2,147,483,647 to 2,147,483,647
unsigned short int (unsigned short)	16	0 to 65,535
unsigned int	32	0 to 4,294,967,295
unsigned long int (unsigned long)	32	0 to 4,294,967,295

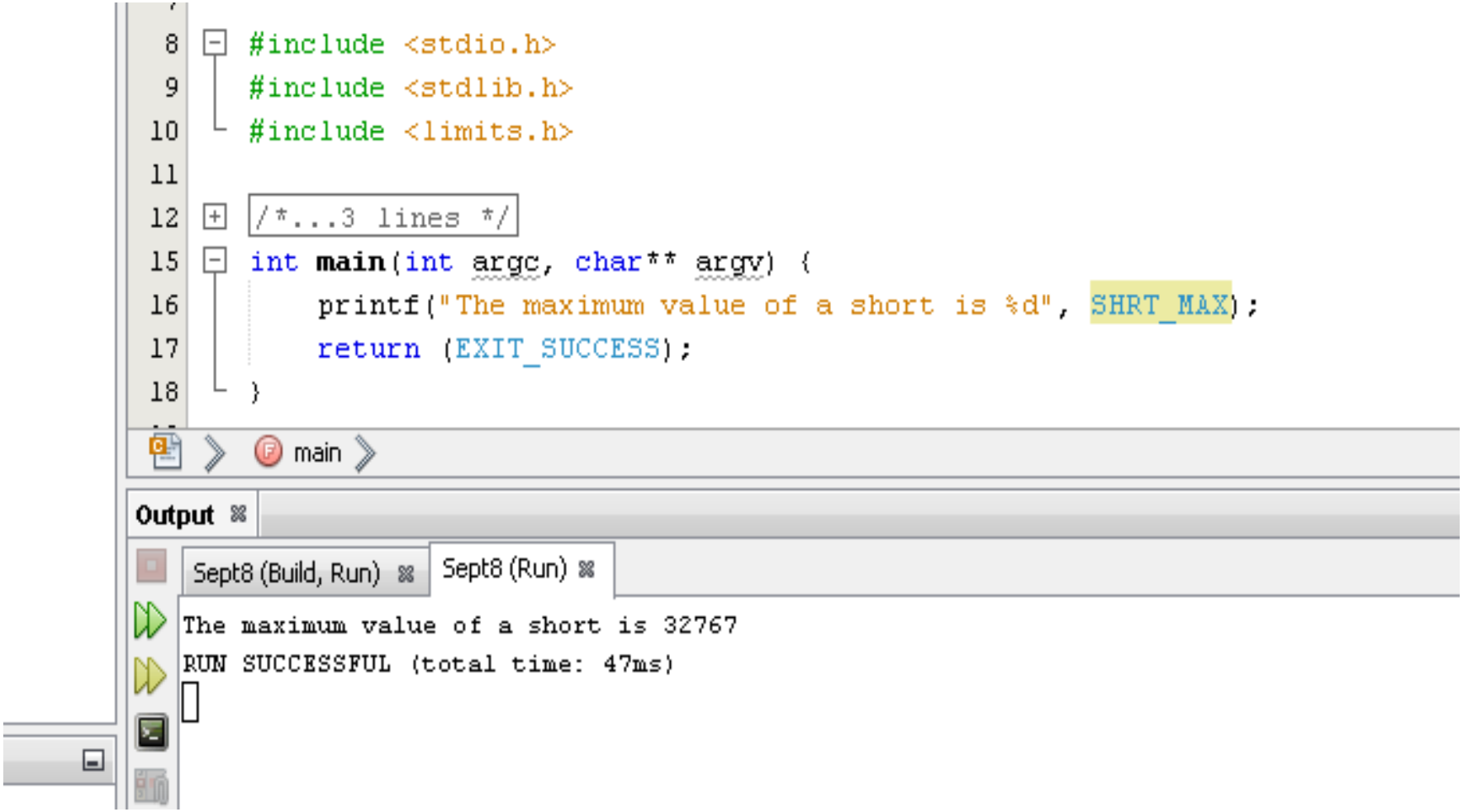
Floating Point Types

- Type *float*: 32bits (IEEE standard)
  - Range of values: 3.4E-38 ~ 3.4E+38
  - precision: 6 digits
- Type *double*: 64 bits (IEEE standard)
  - Range of values:1.7E-308 ~1.7E+308
  - precision: 15 digits

Implementation Defined Limits

- **limits.h** and **float.h** contain the implementation specifics for your computer
- **limits.h** defines constants for the sizes of integral types. Ex:
  - INT_MAX – maximum value of int
  - SHRT_MIN – min value of short
  - ULONG_MAX – max value of unsigned long
- **float.h** - constants related to floating point arithmetic. Ex:
  - FLT_DIG – decimal digits of precision for float
  - DBL_MAX – maximum value for double

Using limits.h



Integer Constants

- Integer numbers written in decimal -- type **int**
  - 3988, -98791, 9771121, 0
- Suffix l or L → type *long int*
  - 3988L
- Suffix u or U → type *unsigned int*
  - 3988U
- Suffix ul or UL → type *unsigned long int*
  - 3988UL

Floating Point Constants

- Constants of type **double**: numbers written
  - with a decimal point: 0.1, .45
  - or with an exponent: 3e-7 (3×10^{-7}), 24E2
  - or with both: 4.6e-9, -23.E4
- Suffix f or F → type *float*
- Suffix l or L → type *long double*

Type **char**

- Size: 1 byte memory (8 bits)
- Variables and constants of type **char** **have integer values**.
- Range of values is implementation-dependent:
  - -128 ~ 127
  - 0 ~ 255
- Printable characters: 0 – 127.
- American Standard Code for Information Interchange (ASCII) character set: 0 ~ 127.
- Conversion specifier for char is %c.
- When %d is used in printf for an expression of type char, its integer value is printed.
- printf("%c", 's');
- printf("%d", 's');

ASCII character set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. D.1 ASCII Character Set.

The digits at the left of the table are the left digits of the decimal equivalent (0-127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for "F" is 70, and the character code for "&" is 38.

Characters

- Unicode characters may be represented using

"*wide characters*" : **wchar_t**

- Size of **wchar_t** is implementation dependent (16 or 32 bits)
- The newer C standards: **char16_t, char32_t**

Checking the Range of chars

```
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <limits.h>
11
12 /*...3 lines */
15 int main(int argc, char** argv) {
       printf("The min value of a char is %d.\nThe max value of a char is %d.",
17             CHAR_MIN, CHAR_MAX);
18     return (EXIT_SUCCESS);
```

main > printf("The min value of a char is %d.\nThe max value of a char is %d.",

Output

Sept8 (Build, Run) | Sept8 (Run)

```
The min value of a char is -128.
The max value of a char is 127.
RUN SUCCESSFUL (total time: 47ms)
```

Character Constants

- A constant of type *char*: a single character within single quotes: ‘X‘, ‘a’, ‘3’
- Its value is an integer.
  - ASCII: value of 'A' is 65, value of 'a' is 97.
- Some characters are written as escape sequences:
  - \n (newline), \t (tab), \v(vertical tab), \b (backspace)
  - corresponding constants of type char: ‘\n’, ‘\t’, ‘\v’, ‘\b’
  - Table of escape sequences: pp. 403

Automatic Type Conversions in Expressions

- The value of the operands of type **char** or **short** in arithmetic expressions is converted to **int.**
- ‘A’+’a’ -- the result is of type **int** (65+97=162)
- **int** x = ‘A’+’a’;

```
int main(void){
char x1='a'+3;  /* x1 is declared and assigned a value */
char x2 = x1+1;
printf( "%c%c\n\n", x1,x2);
return 0;}
```

What is the output?

Automatic Arithmetic Conversions

- If two operands of an arithmetic operator have different types, then the value of "**narrower**" type is converted to the "**wider**" type.

- **int→long→float→double→long double**

- 12 / 5.0 -- type?, value?
- Answer:
- 10.0 + 1/2 -- type?, value?
- Answer:

Assignment Operator "="

- Assignment expression*:* **var** = **expr$_1$**
- Evaluate **expr$_1$** and assign its value to **var**.
- **Every expression has a value and a type.**
- The **value** of the assignment expression is that of **expr1.**
- The **type** of the assignment expression is that of **var.**
- "=" has second lowest precedence level, and associates from **right-to-left**.
- Example: int x; x=7*9;

Type Conversions across Assignment

- Consider the Assignment expression*:* **var** = **expr**
- If the type of the right hand side is different than the type of left hand side a type conversion occurs
- The value of **expr** is converted to the type of **var**, and the new value is assigned to **var**
  - Ex: **int** k;

    k = 2.5;

    2.5 is of type **double.** 2.5 is converted to an **int** by truncating the decimal part

    Thus k is assigned the value 2.

Type Conversions across Assignment

- The value of the right side is converted to the type of the left, which is the type of the result.
- Recall that "=" associates from right to left
- **int** j; **double** x;
  - j = x = 2.45;

    x=?, j=?
  - x = j = 2.45;

    x=?, j=?

Data Type Cast Operator

- Cast operation: converting from one type to another.
- *(**type**) expression*
  - (**float**)17/4
  - (**float**)(17/4)
- Same level of precedence as the other unary operators (higher than binary operators).

Relational and Equality Operators

- Relational operators: >, <, >=, <=.
- Equality operators: == (equal), != (nonequal).
- The **type** of a relational or equality expression is **int**.
- The **value** is

  1, if the condition is TRUE

  0, if the condition is FALSE

- What are the **type** and **value**?
- 100 + (100 != 200)

Logical Operators

- Logical negation operator: “!”
- Can be applied to any expression with a numerical value
- A nonzero numerical value means TRUE
- Zero means FALSE
- !**var** or !(**expr**):
  - !(nonzero) has **value** 0 (**type** **int**)
  - !(0) has **value** 1 (**type** **int**)
- !(75.9) type?, value?
- !(3==4) type?, value?

Logical Operators &&, ||

- && (logical ***and***)
- || (logical ***or***)
- In ASCII the lower case letters have consecutive values according to alphabetical order. Likewise for the uppercase letters.
- Condition that a **char c** is a lowercase letter:

  c>='a' && c <= 'z'

- The **value** of a logical expression **can be only 0 or 1**, and has **type int**.

Truth Tables for &&, ||

X	Y	X&&Y	X||Y
T (nonzero)	T (nonzero)	T (1)	T (1)
T (nonzero)	F (0)	F (0)	T (1)
F (0)	T (nonzero)	F (0)	T (1)
F (0)	F	F (0)	F (0)

Logical Operators &&, ||. Short-circuit Evaluation

- The operands of “&&” and “||” are evaluated left to right.
- Evaluation stops as soon as the **truth** (value 1) or **falsehood** (0) is determined (shortcircuit evaluation).

```
int main(void)
{       int a = 0, b = 1, c,d;
        c = a++ && b++;
        d = a++ || b++;
        printf("a=%d\nb=%d\nc=%d\nd=%d\n",a,b,c,d);
        return 0;
}
```

Output:

Comma Operator “,”

- **$expr_1$, $expr_2$**
- **$expr_1$** is evaluated, then **$expr_2$**
- The **value** of the comma expression and its **type** are those of **$expr_2$**.
- “,” associates from **left to right**.
- Lowest precedence among C operators.
- Ex: s of type *double* , t of type *int*
  - s = (t=2, t+3)
  - s = t = 2, t+3
- Comma **is not an operator** in list of function arguments, or in list of initializers.

Precedence and Associativity Table

Operators						Associativity	Type
()						left to right	parentheses
++ preincrement	**--** predecrement	**+**	**-**	**!**	**(type)**	right to left	unary
*****	**/**	**%**				left to right	multiplicative
+	**-**					left to right	additive
<	**<=**	**>**	**>=**			left to right	relational
==	**!=**					left to right	equality
&&						left to right	logical AND
||						left to right	logical OR
=	**+=**	**-=**	***=**	**/=**	**%=**	right to left	assignment
,						left to right	comma operator

On “condition” in if…else and Loops

- *The **condition*** in *an **if … else*** statement *or in loops* can be **any expression of numerical type**.
- Nonzero value -- TRUE
- Zero value -- FALSE

-
```
if (1/2)              // condition is false
    printf(“yes”);
else
   printf(“no”);
```

Output:

-
```
if (1/2.0)       // condition is true
    printf(“yes”);
else
   printf(“no”);
```

Output:

Confusing "==" with "="

- **double** x=9.5;

```
if (x == 7)
   printf("yes");
else
   printf("no");
```

  Output: **no**

- If you accidentally type "**=**" instead of "**==**" in the condition, a logic error occurs:
- **double** x=9.5;

```
if (x = 7)     /* correct syntax */
   printf("yes");
else
   printf("no");
```

- x=7 is an assignment expression and its **value** is the value of x, i.e., 7, which means TRUE.
- Output: **yes**

Confusing "==" with "="

- To prevent the logic error:
- double x=9.5;

```
if (7 == x)
    printf("yes");
else
   printf("no");
```

- double x=9.5;

```
if (7 = x)      /*syntax error */
    printf("yes");
else
   printf("no");
```

Conditional Operator

- **cond** ? **expr1** : **expr2**
  - evaluate **cond**
  - evaluate **expr1** if **cond** is **true**
  - evaluate **expr2** if **cond** is **false**
  - Precedence level immediately higher than assignment.
- `min = x > y ? y : x;`
- `gr>=60?printf("Passed\n"):printf("Failed\n");`
- ✗ `gr>=60?printf("Passed\n");:printf("Failed\n");`

Dangling else

```
//A
1.   if( exp1 )
2.       if( exp2 )
3.           statement1
4.   else
5.       statement2
```

```
//B
1.   if( exp1 )
2.   {
3.       if( exp2 )
4.           statement1
5.       else
6.           statement2
7.   }
```

Variant A is equivalent to variant B.

```
//C
1.   if( exp1 )
2.   {
3.       if( exp2 )
4.           statement1
5.   }
6.   else
7.       statement2
```

break Statement

break;

- It can be used inside the body of
  - **for**
  - **while**
  - **do…while**
  - **switch**
- Control exits immediately from the statement.
- Execution continues from the next statement.

continue Statement

continue*;*

- It can be used inside the body of
  - **for**
  - **while**
  - **do…while**
- The remaining statements in the body are skipped.
- Execution proceeds with the next iteration of the loop (with “for”, *expression2* is evaluated first).

break versus continue

```
int i = 0;
while( i < 10 )
{
        i++;
        if( i % 3 == 0 )
            break;
        printf("%d\n", i );
}
```

```
1
2
```

```
int i = 0;
while( i < 10 )
{
        i++;
        if( i % 3 == 0 )
            continue;
        printf("%d\n", i );
}
```

```
1
2
4
5
7
8
10
```

```
int i=3,j;
for( j = 1; j <= 5; j++ )
    {

        printf("*");
    }
```

```
int i=3,j;
for( j = 1; j <= 5; j++ )
    {
        if(j == i)
            break;
        printf("*");
    }
```

```
int i=3,j;
for( j = 1; j <= 5; j++ )
    {
        if(j == i)
            continue;
        printf("*");
     }
```

```
int i,j;
for( i = 1; i <= 5; i++ )
{
    for( j = 1; j <= 10; j++ )
    {
        if(j == i)
           break;
        printf("*");
    }
    printf("\n");
}
```

```
**********
**********
**********
**********
**********
```

```
*
**
***
****
```

```
int i,j;
for( i = 1; i <= 5; i++ )
{
    for( j = 1; j <= 10; j++ )
    {
        if(j == i)
           continue;
        printf("*");
     }
     printf("\n");
}
```

Multiple Selection

- We can use nested **if …else** when we need to test for multiple cases

```
if (score > 80)
    printf("Excellent!");
else
{   if (score > 70)
       printf("Good!");
    else
    {   if (score > 50)
            printf("Sufficient!");
        else
            printf("INSUFFICIENT!");
    }
}
```

Multiple Selection

- We can use nested **if …else** for multiple selection

```
if (score > 80)
    printf("Excellent!");
else
    if (score > 70)
       printf("Good!");
    else
        if (score > 50)
             printf("Sufficient!");
        else
             printf("INSUFFICIENT!");
```

Multiple Selection

```
if (score > 80)
      printf("Excellent!");
else
      if (score > 70)
          printf("Good!");
      else
           if (score > 50)
                  printf("Sufficient!");
           else
                  printf("INSUFFICIENT!");
```

Alternative way of writing the above

```
if (score > 80)
      printf("Excellent!");
else if (score > 70)
      printf("Good!");
else if (score > 50)
      printf("Sufficient!");
else
      printf("INSUFFICIENT!");
```

Switch Statement

```
switch ( integer expression )
{
    case constant1: statement_list_1
    case constant2: statement_list_2
    case constant3: statement_list_3
    case constant4: statement_list_4
    default: statement_list_default
}
```

- First the integer expression is evaluated. Then control goes to the case matching the value of expression.
- If no case matches, control goes to default. If no default, control exits switch.
- To use switch for multiple selection insert **break;** after the statement list for each case.
- **break;** causes control to **exit the switch statement**.

Multiple Selection with *switch*

- with **break;**

```
1.    switch(grade)
2.    {
3.          case 'A': printf("85-100\n");
4.                    break;
5.          case 'B': printf("70-84\n");
6.                    break;
7.          case 'C': printf("60-69\n");
8.                    break;
9.          case 'D': printf(" < 60\n");
10.                   break;
11.         default: printf("60-69\n");
12.   }
```

- when `grade` is 'B', the output is:

70-84

Switch Statement

- without **break; (logic error)**

```
1.   switch(grade)
2.   {
3.        case 'A': printf("85-100\n");
4.        case 'B': printf("70-84\n");
5.        case 'C': printf("60-69\n");
6.        case 'D': printf(" < 60\n");
7.        default: printf("60-69\n");
8.   }
```

- when `grade` is 'B', the output is:

```
70-84
60-69
 < 60
60-69
```

Multiple cases can share the same entrance.

```
1.    switch(grade)
2.    {
3.         default: break;
4.         case 'A':
5.         case 'a':
6.            printf(“case A and case a\n”);
7.            break;
8.         case 'B':
9.         case 'b':
10.           printf(“case B and case b\n”);
11.           break;
12.        case 'C':
13.        case 'c':
14.           printf(“case C and case c\n”);
15.           break;
16.   }
```

- The two case labels have the same value → syntax error

```
1.    switch(grade)
2.    {
3.          case 'A':
4.             printf("case A and case a\n");
5.             break;
6.          case 65:
7.             printf("case B and case b\n");
8.             break;
9.    }
```

Notes (*switch*)

- The type of the expression should be integral (this includes **char**).
- ***default*** is optional.
- Case labels should be distinct (distinct values).
- Multiple cases can share the same entrance.

Input and Output

- #include <stdio.h>
- Formatted input and output
  - printf( ) (prints on the screen)
  - scanf( )   (reads input from the keyboard)
- character input and output
  - putchar( ) (outputs on the screen)
  - getchar( ) (reads input from the keyboard)

Conversion Specifiers

- printf() and scanf() use conversion specifiers
- int: %d
- char: %c
- float: %f
- double: %f (printf); %lf (scanf)

Formatted output

- printf (format-control-string, argument-list);
- Ex: **printf("sum=%d, average=%f", 2+3, (2+3.0)/2);**
- **Format-control- string** (FCS): ordinary characters and conversion specifiers enclosed within " "
- **Argument list** (AL) : Expressions separated by commas.
- The # of expressions in AL must equal the # of conversion specifiers in FCS.
- EFFECT: Ordinary characters in FCS are printed, in place of each conversion specifier the value of the corresponding expression is printed.
- For earlier ex: sum=5, average=2.50

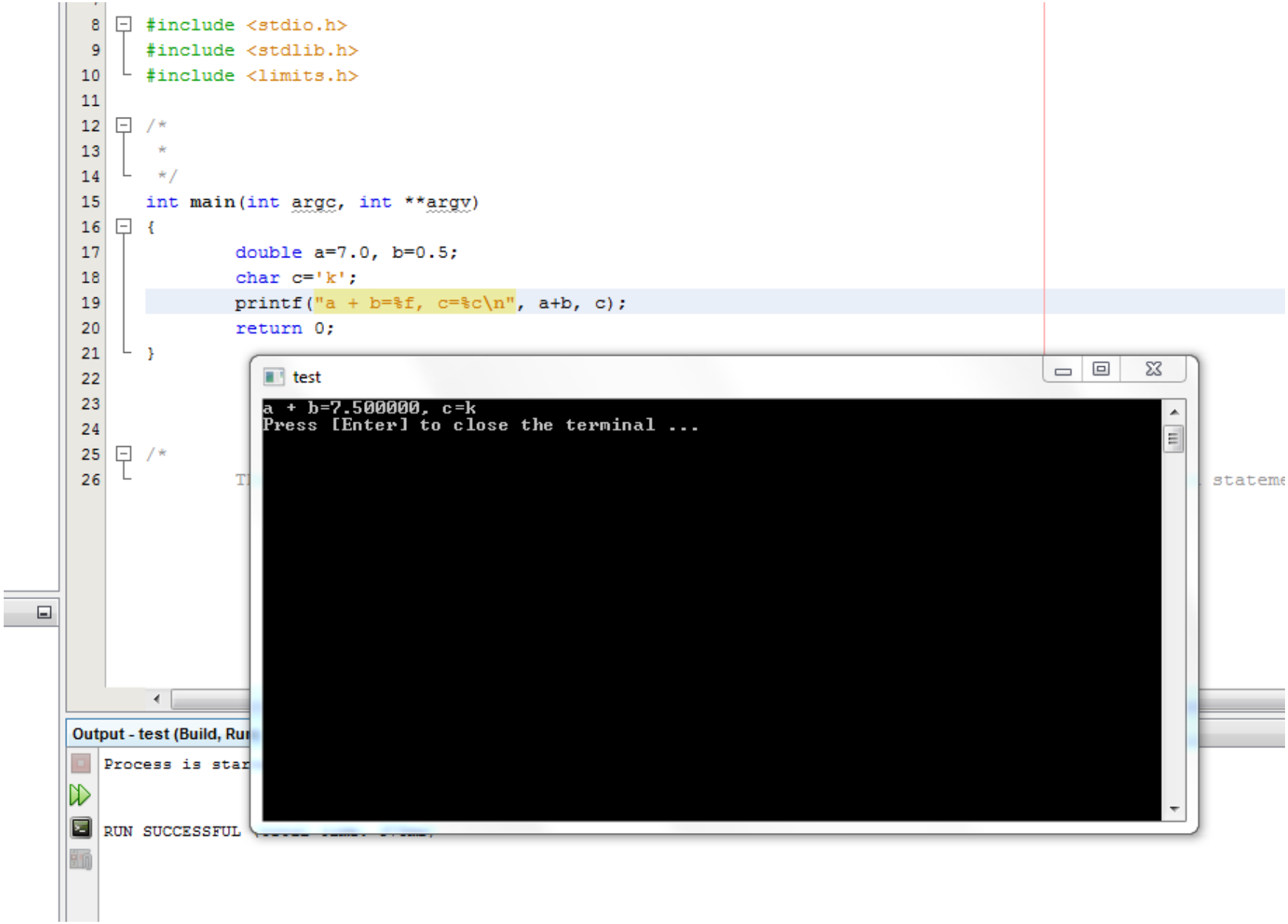
Formatted output

- printf (format-control-string, argument-list);

```
int main(void){
   double a=7.0, b=0.5;
   char c='k';
   printf("a + b=%f, c=%c\n", a+b, c);
   return 0;}
```

Output:

a+b=7.500000, c=k


#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
/*
*
*/
int main(int argc, int **argv)
{
double a=7.0, b=0.5;
char c='k';
printf("a + b=%f, c=%c\n", a+b, c);
return 0;
}
/*
test
a + b=7.500000, c=k
Press [Enter] to close the terminal ...
Output - test (Build, Ru
Process is star
RUN SUCCESSFUL

Formatted output

- The size of the field (field width) used to print a value can be controlled:
- printf(“%4d%5c”, 34, ‘b’);
- Output: _ _ 34_ _ _ _ b
- The precision can be specified (default precision is 6):
- printf(“%8.3f, %.2f”, 1.23678, 7.238);
- Output: _ _ _ 1. 237, 7.24

Formatted input

- scanf (format-control-string, argument-list);
- Example: **scanf("%d%lf%c%c",&v1, &v2,&v3,&v4);**
- & is the address operator
- Each of **%d,%lf** indicate that **the next sequence of characters until the first white space is read** from the input and converted to a value of corresponding type, which is assigned to the corresponding variable.
- **%c** : **next character is read (this could be a whitespace)**

Formatted input

- scanf (format-control-string, argument-list);
- Example: **scanf("%d%lf%c%c",&v1, &v2,&v3,&v4);**
- Input: **-567 45.8 gf**
- v1=-567, v2=45.8, v3=' ', v4='g'
- Notice that after the input sequence 45.8 is read, the next character in the input stream is ' ' (space). The first %c causes the space to be read and stored in v3. The second %c causes **g** to be read and stored in v4.
- To read the **next non-white space character** include a white space in front of %c:
- **scanf("%d%lf %c%c",&v1, &v2,&v3,&v4);**
- Input: **-567 45.8 gf**
- v1=-567, v2=45.8, v3='g', v4='f'

getchar() and putchar()

- #include <stdio.h>
- **getchar()** (with no argument) **reads the next character from the standard input stream** (keyboard)
- Assume **ch** is of type **char**
  - **ch = getchar();** (reads the next character from the standard input stream and assigns it to variable ch)
- The above statement has the same effect as:
  - **scanf("%c", &ch);**
- **putchar(ch);** has the same effect as
- **printf("%c", ch);**
- putchar('\n'); (moves the cursor on the screen to the beginning of the next line)