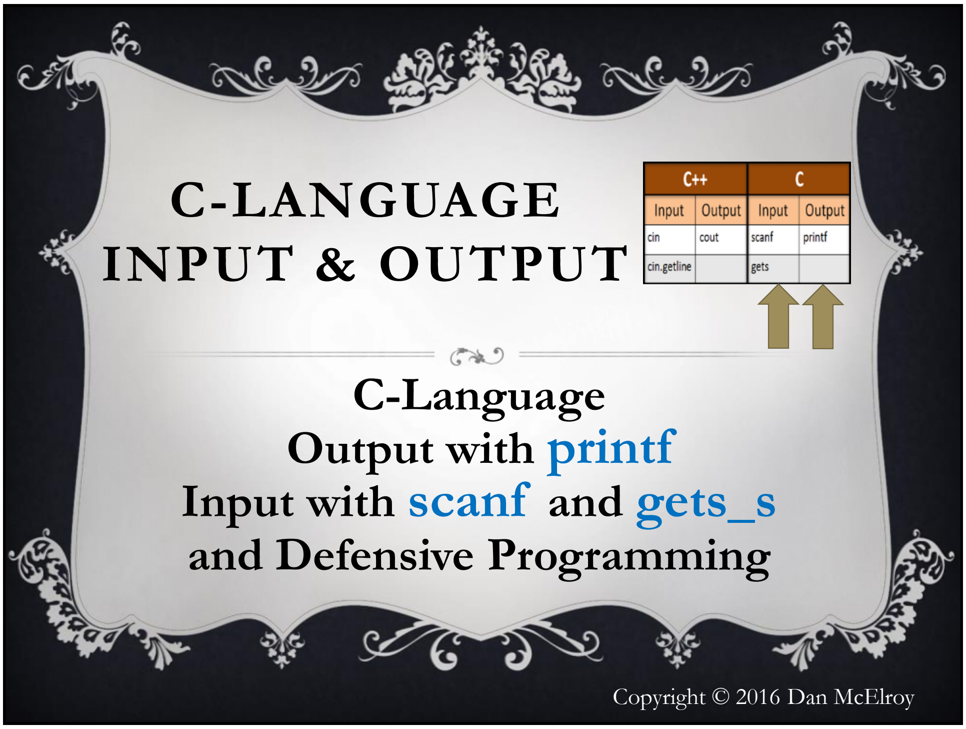
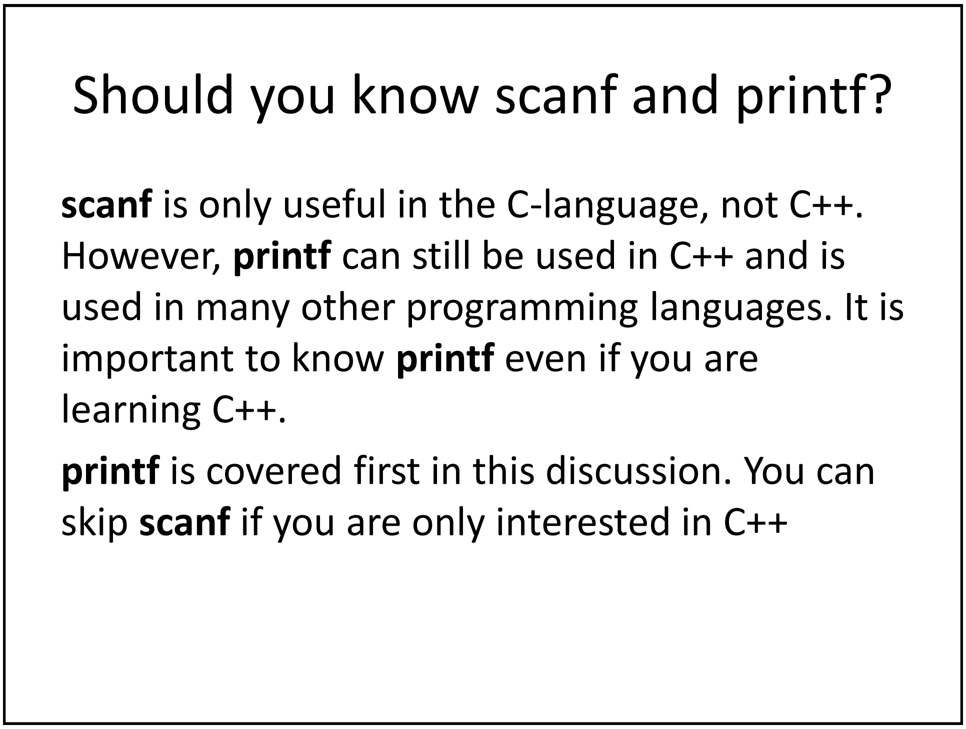


C-LANGUAGE INPUT & OUTPUT

C++		C	
Input	Output	Input	Output
cin	cout	scanf	printf
cin.getline		gets	

C-Language
Output with printf
Input with scanf and gets_s
and Defensive Programming

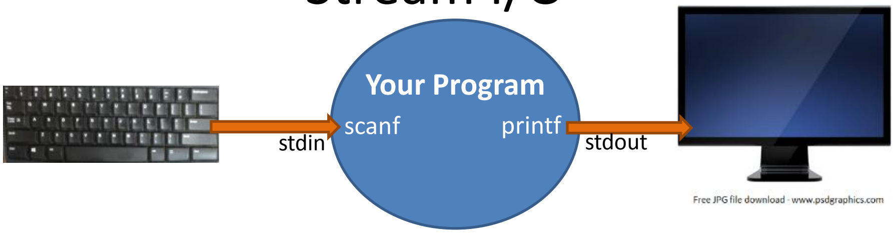


Should you know scanf and printf?

scanf is only useful in the C-language, not C++. However, **printf** can still be used in C++ and is used in many other programming languages. It is important to know **printf** even if you are learning C++.

printf is covered first in this discussion. You can skip **scanf** if you are only interested in C++

Stream I/O



Free JPG file download - www.psdgraphics.com

The most common way for a C program to input from a keyboard and output to the display is to use **scanf** and **printf** which stand for scan-formatted and print-formatted.

```
int a;
printf ("Enter a number: ");
scanf_s ("%d", &a);
printf ("The number squared is %d\n", a*a);
```

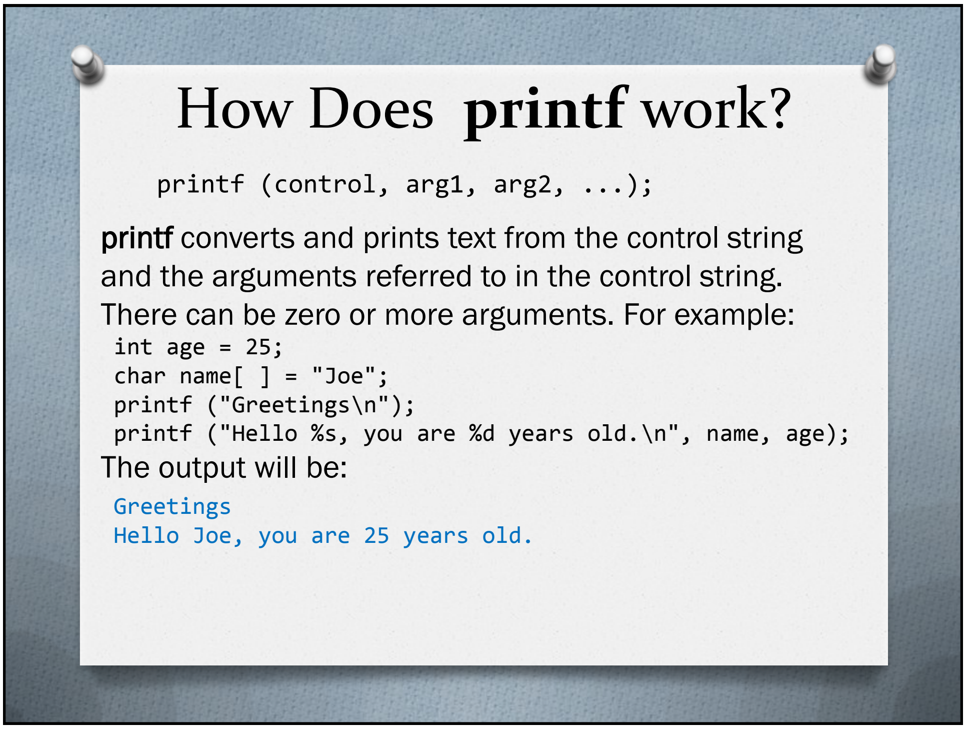
Use **\n** in C to move the cursor to the next line on the display. The backslash character \ is called the 'escape' character. It gives the next character a special meaning. Make sure you enter the backslash \ and not the slash /

NOTE: Operating systems like Linux can use pipes and redirection to cause stdin and stdout to use other devices or even other programs

scanf and **printf** Don't Know the Data Type

There are many data types that are used in the C-language such as: int, float, double, char, char* etc.

In order to properly work with these different data types, a "control string" is passed to these functions to identify the type of data and how it is to be processed. scanf and printf can work with multiple pieces of data, each with different data types each time a request is made to them.



How Does **printf** work?

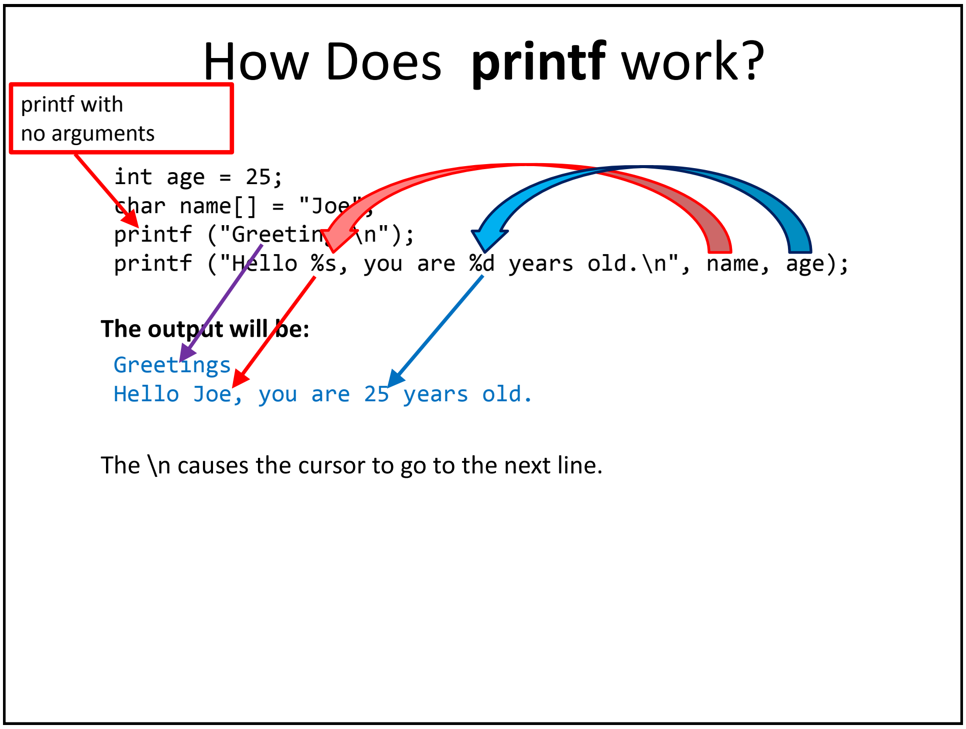
```
printf (control, arg1, arg2, ...);
```

printf converts and prints text from the control string and the arguments referred to in the control string. There can be zero or more arguments. For example:

```
int age = 25;
char name[ ] = "Joe";
printf ("Greetings\n");
printf ("Hello %s, you are %d years old.\n", name, age);
```

The output will be:

```
Greetings
Hello Joe, you are 25 years old.
```



How Does **printf** work?

printf with
no arguments

```
int age = 25;
char name[] = "Joe";
printf ("Greetings\n");
printf ("Hello %s, you are %d years old.\n", name, age);
```

The output will be:

```
Greetings
Hello Joe, you are 25 years old.
```

The \n causes the cursor to go to the next line.

The **printf** Format Specifiers

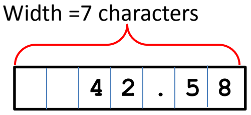
Control	Description
%d	Decimal integer. The argument should be an integer.
%o	Octal integer. The argument should be an integer.
%x	Hexadecimal integer. The argument should be an integer.
%X	Hexadecimal integer. A-F is displayed in upper case
%c	The argument should be the address of a character.
%s	Character string. The argument should a character array
%e	Floating point number in engineering format. The argument should be a float. Example 5632 displays as 5.632E3
%f	Floating point number. The argument should be a float.
%lf	Long-float. The argument should be a double.
%g	Use %e or %f which ever is shorter. Non-significant zeros are not printed
%%	If the character after the % is not a control character, print it. %% prints %

printf Field Width and Precision

The printf control specifiers can have an optional field width and/or precision listed. Examples:

```
double length = 42.578;
printf ("%7.2lf", length);
```

small-L

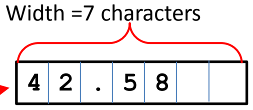


		4	2	.	5	8

printf uses a total of 7 character positions with 2 digits past the decimal and is right-justified

```
double length = 42.578;
printf ("%-7.2lf", length);
```

Left-justified



4	2	.	5	8		

printf Field Width and Precision

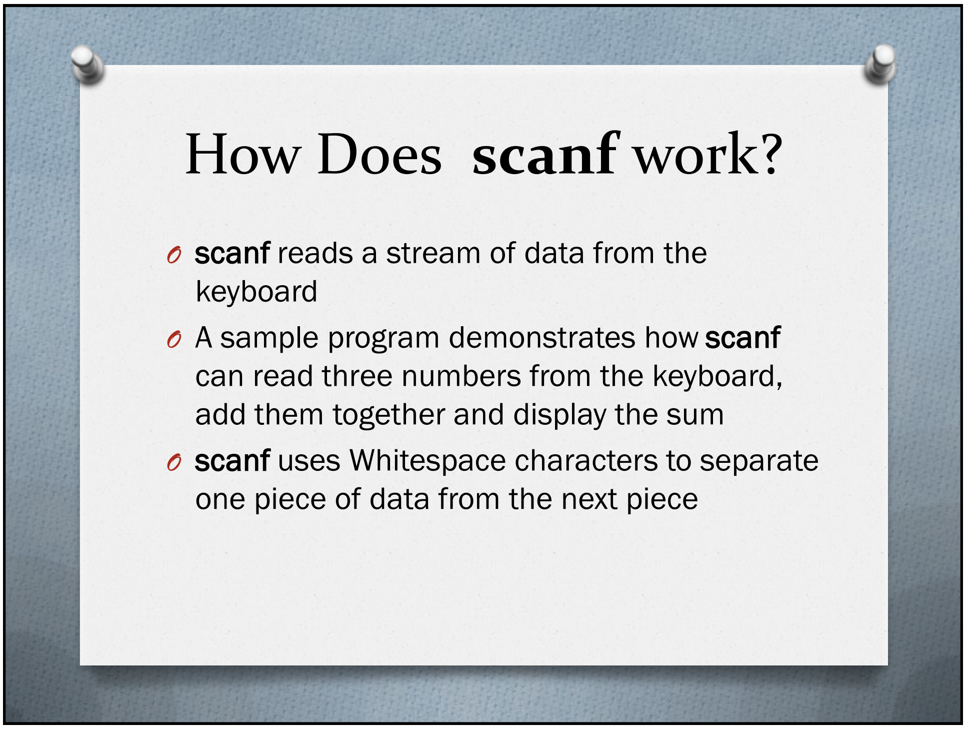
Examples with %s:

```
char msg[ ] = "Hello world!";
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`printf ("%s", msg);`	H	e	l	l	o		w	o	r	l	d	!		
`printf ("%14s", msg);`			H	e	l	l	o		w	o	r	l	d	!
`printf ("%-14s", msg);`	H	e	l	l	o		w	o	r	l	d	!		
`printf ("%7.10s", msg);`	H	e	l	l	o		w							
`printf ("%-7.10s", msg);`				H	e	l	l	o		w				

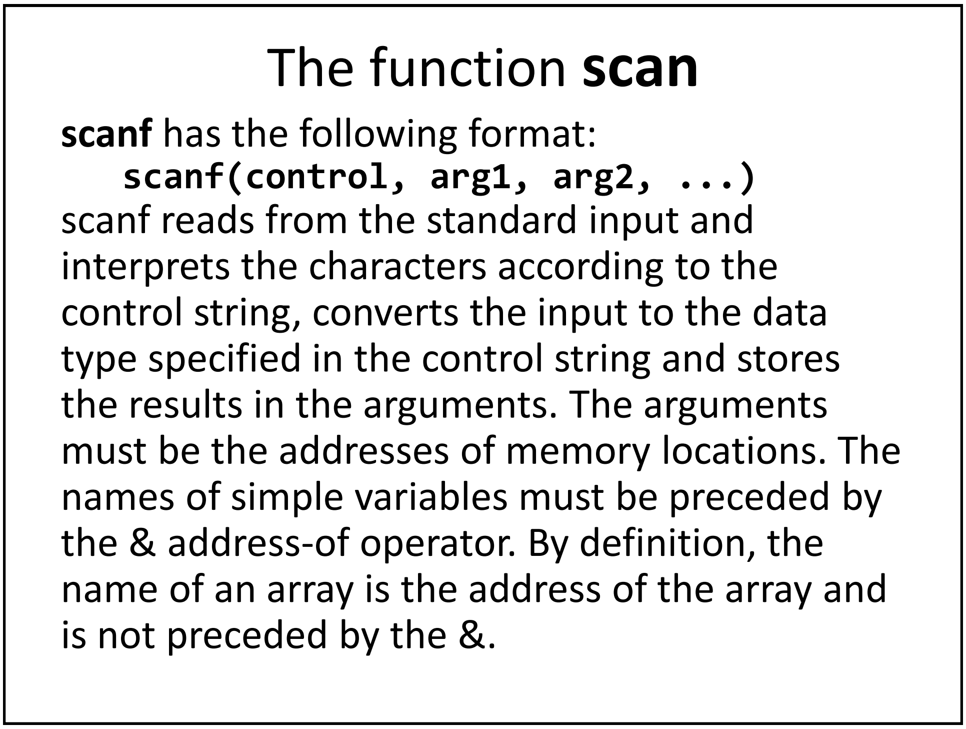
WARNING

printf uses the control string to determine the number and data type for the arguments that follow. printf gets confused and prints nonsense answers if there are not enough arguments, they are the wrong type, or the arguments are not listed in the same order as the control specifiers!



How Does **scanf** work?

- **scanf** reads a stream of data from the keyboard
- A sample program demonstrates how **scanf** can read three numbers from the keyboard, add them together and display the sum
- **scanf** uses Whitespace characters to separate one piece of data from the next piece



The function **scan**

scanf has the following format:

```
   scanf(control, arg1, arg2, ...)
```

scanf reads from the standard input and interprets the characters according to the control string, converts the input to the data type specified in the control string and stores the results in the arguments. The arguments must be the addresses of memory locations. The names of simple variables must be preceded by the & address-of operator. By definition, the name of an array is the address of the array and is not preceded by the &.

The **scanf** Control String

The control string contains one or more specifications that tell scanf how to interpret the input data. Blanks and tabs are ignored, ordinary characters (not %) which are expected to match the input data and conversion specifiers which start with %, contain an optional field width and a conversion control character.

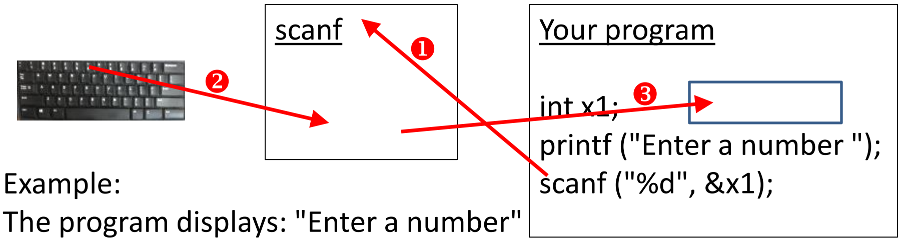
scanf Field Width

The control specifiers can also have a count of the number of characters to process. For example, **%3d** causes scanf to read three characters from the input and convert them into a decimal integer. **%10s** causes scanf to read up to 10 characters and store them into a character array.

The **scanf** Conversion Specifiers

Control	Description
%d	Decimal integer. The argument should be the address of an integer.
%o	Octal integer. The argument should be the address of an integer.
%x	Hexadecimal integer. The argument should be the address of an integer.
%c	Input the next character, even if it a whitespace character. The argument should be the address of a character. To skip over the whitespace and read the next character, use %1s.
%s	Character string. The argument should a character array that is large enough to hold the string. A NULL byte is placed at the end of the string.
%f	Floating point number. The argument should be the address of a float.
%lf	Long-float. The argument should be the address of a double.

Why the **address-of** operator is needed

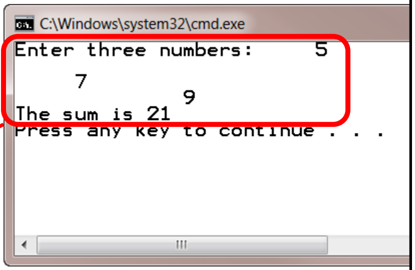


```
int x1;
printf ("Enter a number ");
scanf ("%d", &x1);
```

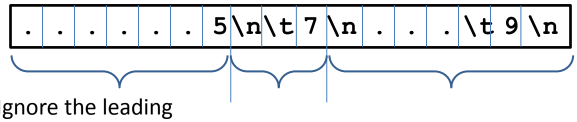
Example:
The program displays: "Enter a number" and then ❶ calls **scanf** to read a decimal number from the keyboard. **"%d"** in the control string indicates that **scanf** is to read a decimal number. ❷ If the user would press the [5] [2] and [1] keys, **scanf** takes those individual keys and convert them into the decimal value of 521. Since your program is in a different part of memory than scanf, **scanf** needs to know where to save the data. ❸ The address of the variable x1 is passed to **scanf** as a parameter &x1**.** **scanf** now knows where to place the data.

Whitespace

```
/* input */
printf ("Enter three numbers: ");
count = scanf_s ("%d %d %d", &a, &b, &c);
/* process */
sum = a + b + c;
/* output */
printf ("The sum is %d\n", sum);
```



```
Enter three numbers:      5
     7
                9
The sum is 21
Press any key to continue . . .
```



.	.	.	.	.	.	5	\n	\t	7	\n	.	.	.	\t	9	\n

Ignore the leading spaces then read the 5 into the variable **a**

Ignore the Enter and Tab keys then read the 7 into the variable **b**

Ignore the Enter spaces and Tab then read the 9 into the variable **c**

. represent the space bar was pressed

\n represent the Enter key was pressed

\t represent the Tab key was pressed

Sample Program – Add 3 Numbers

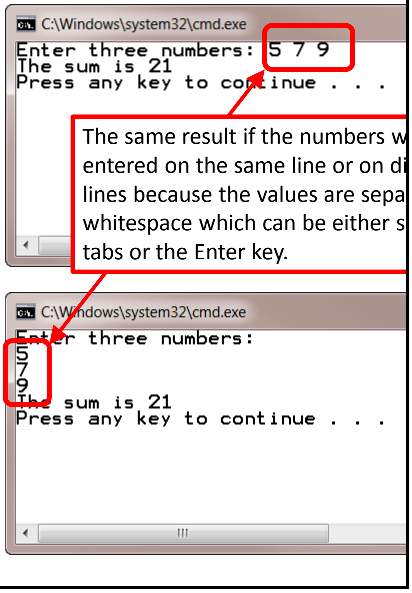
```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a;
    int b;
    int c;
    int sum;

    /* input */
    printf ("Enter three numbers: ");
    scanf_s ("%d %d %d", &a, &b, &c);
    /* output */
    printf ("The sum is %d\n", sum);
    return 0;
}
```

stdio.h is a header file, thus the **.h** It has all the information needed to compile **scanf**, **printf** and other routines.

Inside the parentheses for **scanf** is the control string enclosed in quotes " and the address of the variables that will receive the data read by scanf. **%d** indicates that a decimal value is to be read. The & character is the **address-of** operator. Because **scanf** is in one part of memory and your variables are in another part, **scanf** needs the address of each of the variables so that it will know where to place the data. **printf** does not need the & because the program is giving data not receiving.

Sample Program – Add 3 Numbers

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a;
    int b;
    int c;
    int sum;

    /* input */
    printf ("Enter three numbers: ");
    scanf_s ("%d %d %d", &a, &b, &c);
    /* output */
    printf ("The sum is %d\n", sum);
    return 0;
}
```



```
Enter three numbers: 5 7 9
The sum is 21
Press any key to continue . . .
```

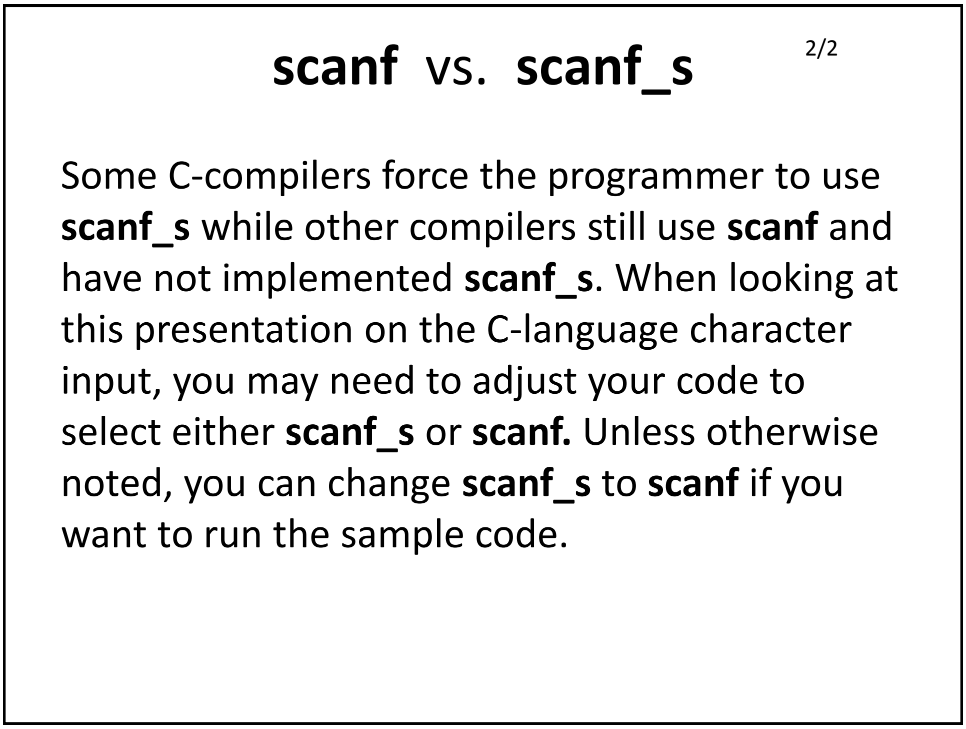
The same result if the numbers w
entered on the same line or on di
lines because the values are sepa
whitespace which can be either s
tabs or the Enter key.

```
Enter three numbers:
5
7
9
The sum is 21
Press any key to continue . . .
```

scanf vs. **scanf_s**

1/2

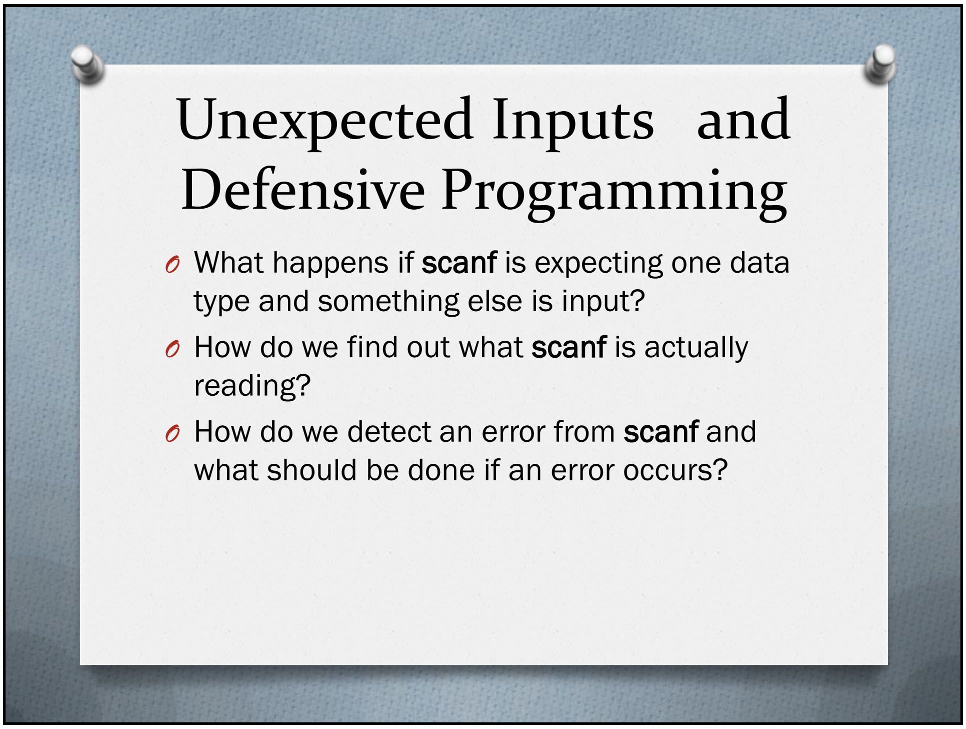
scanf is the original scan-formatted routine for the C-language. As C became more popular, it was found that users could enter more characters than the program was expecting and could cause the program to crash, or worse. The updated version of **scanf** is called **scanf_s**, or scan-formatted-secure. The size of an array that receives a character string must be specified in **scanf_s** to prevent a buffer overrun.



scanf vs. **scanf_s**

2/2

Some C-compilers force the programmer to use **scanf_s** while other compilers still use **scanf** and have not implemented **scanf_s**. When looking at this presentation on the C-language character input, you may need to adjust your code to select either **scanf_s** or **scanf.** Unless otherwise noted, you can change **scanf_s** to **scanf** if you want to run the sample code.

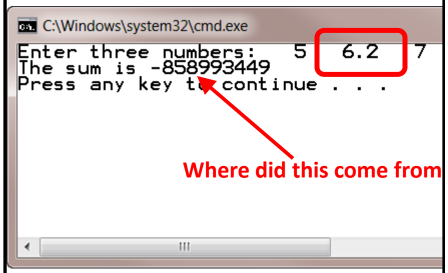


Unexpected Inputs and Defensive Programming

- What happens if **scanf** is expecting one data type and something else is input?
- How do we find out what **scanf** is actually reading?
- How do we detect an error from **scanf** and what should be done if an error occurs?

Unexpected Data

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a;
    int b;
    int c;
    int sum;

    /* input */
    printf ("Enter three numbers: ");
    scanf_s ("%d %d %d", &a, &b, &c);
    /* output */
    printf ("The sum is %d\n", su
    return 0;
}
```



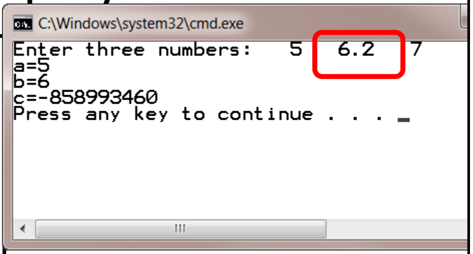
```
Enter three numbers:   5   6.2   7
The sum is -858993449
Press any key to continue . . .
```

Where did this come from

scanf is expecting to input **integers** into a b c

a) Use code to display the values

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a;
    int b;
    int c;

    /* input */
    printf ("Enter three numbers: ");
    /* output */
    printf ("a = %d\n", a);
    printf ("b = %d\n", b);
    printf ("c = %d\n", c);

    return 0;
}
```

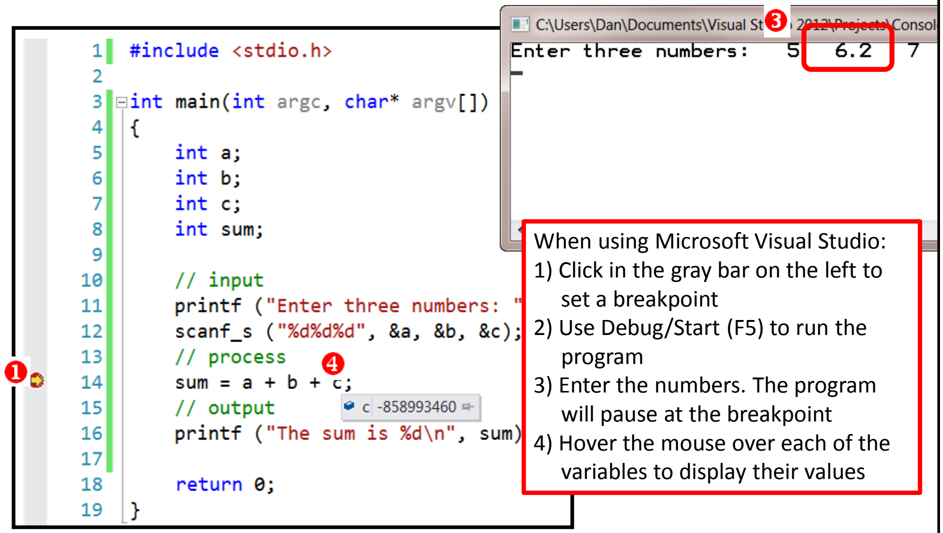


```
Enter three numbers:   5   6.2   7
a=5
b=6
c=-858993460
Press any key to continue . . . _
```

scanf is expecting to input integers into
a b c

scanf reads the 5 into the variable **a**, then reads the 6 into the variable **b**, then when **scanf** went to read into the variable **c** it saw the decimal point. Integers are whole numbers. No decimals allowed. Since the variable **c** was not initialized, whatever garbage was in its memory location is what was used.

b) Use Debug to display the values



```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a;
    int b;
    int c;
    int sum;

    // input
    printf ("Enter three numbers: "
    scanf_s ("%d%d%d", &a, &b, &c);
    // process
    sum = a + b + c;
    // output
    printf ("The sum is %d\n", sum)

    return 0;
}
```

```
Enter three numbers:  5  6.2  7
_
```

When using Microsoft Visual Studio:

1) Click in the gray bar on the left to set a breakpoint
2) Use Debug/Start (F5) to run the program
3) Enter the numbers. The program will pause at the breakpoint
4) Hover the mouse over each of the variables to display their values

Possible Solutions

Solution #1 – Initialize the variables to 0 to prevent weird numbers from showing, but this does not stop wrong answers from being displayed.

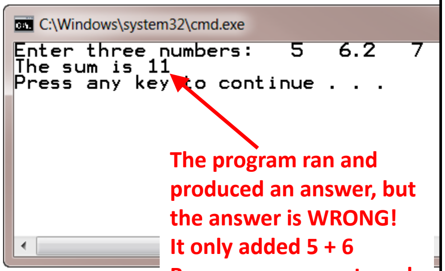
Solution #2 – change the definition of the variables from type **int** to type **double**. This will allow the **6.2** to be read without an error, but the program will still fail if the user inputs a non-numeric character such as **X**. This is NOT a complete solution.

Solution #3 – Test **scanf** and compare the number of data items that were expected to be read by scanf and the number that were actually read.

Solution #1 - Initialize the Variables

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int a = 0;
    int b = 0;
    int c = 0;
    int sum;

    /* input */
    printf ("Enter three numbers: ");
    scanf_s ("%d %d %d", &a, &b, &c);
    /* output */
    printf ("The sum is %d\n", sum);
    return 0;
}
```

scanf is expecting to input integers



```
Enter three numbers:   5   6.2   7
The sum is 11
Press any key to continue . . .
```

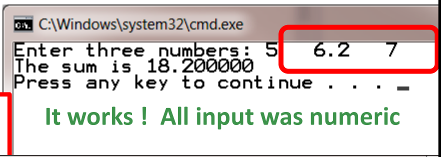
The program ran and produced an answer, but the answer is WRONG! It only added 5 + 6 Because c was not read. scanf stopped trying to read c when it saw the decimal point.

Not a good solution. The answer is WRONG

Solution #2 – **double** Data Type

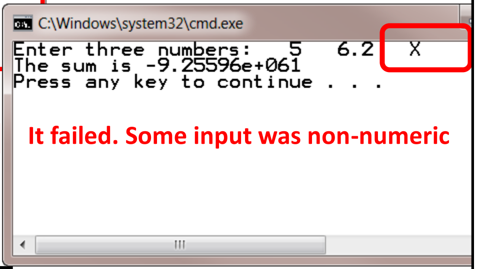
```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    double a;
    double b;
    double c;
    double sum;

    /* input */
    printf ("Enter three numbers: ");
    scanf_s ("%lf %lf %lf", &a, &b, &c);
    /* output */
    printf ("The sum is %lf\n", sum);
    return 0;
}
```

scanf is expecting to input floating point numbers **%lf** is for long-float



```
Enter three numbers: 5   6.2   7
The sum is 18.200000
Press any key to continue . . . _
```

It works ! All input was numeric



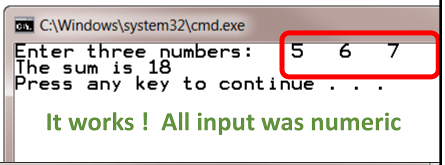
```
Enter three numbers:   5   6.2   X
The sum is -9.25596e+061
Press any key to continue . . .
```

It failed. Some input was non-numeric

Not a good solution.
It still does not detect illegal inputs.

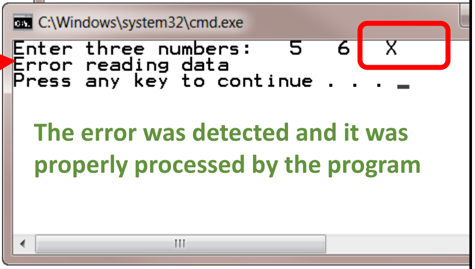
Solution #3 – Test count of items

```
int a;
int b;
int c;
int sum;
int count;

/* input */
printf ("Enter three numbers: ");
count = scanf_s ("%d %d %d", &a, &b, &c);
/* check for errors */
if (count != 3) /* test for not equal 3 */
{
    printf ("Error reading data\n");
    return 1;
}
else
{
    /* process */
    sum = a + b + c;
    /* output */
    printf ("The sum is %d\n", sum);
}
return 0;
```



```
Enter three numbers:  5   6   7
The sum is 18
Press any key to continue . . .
```

It works ! All input was numeric



```
Enter three numbers:  5   6   X
Error reading data
Press any key to continue . . .
```

The error was detected and it was properly processed by the program

This is a good solution

More on **scanf** vs. **scanf_s**

scanf_s was developed to prevent users from entering more text data as a string of characters into arrays than there was room. **%s** identifies a string. Example:

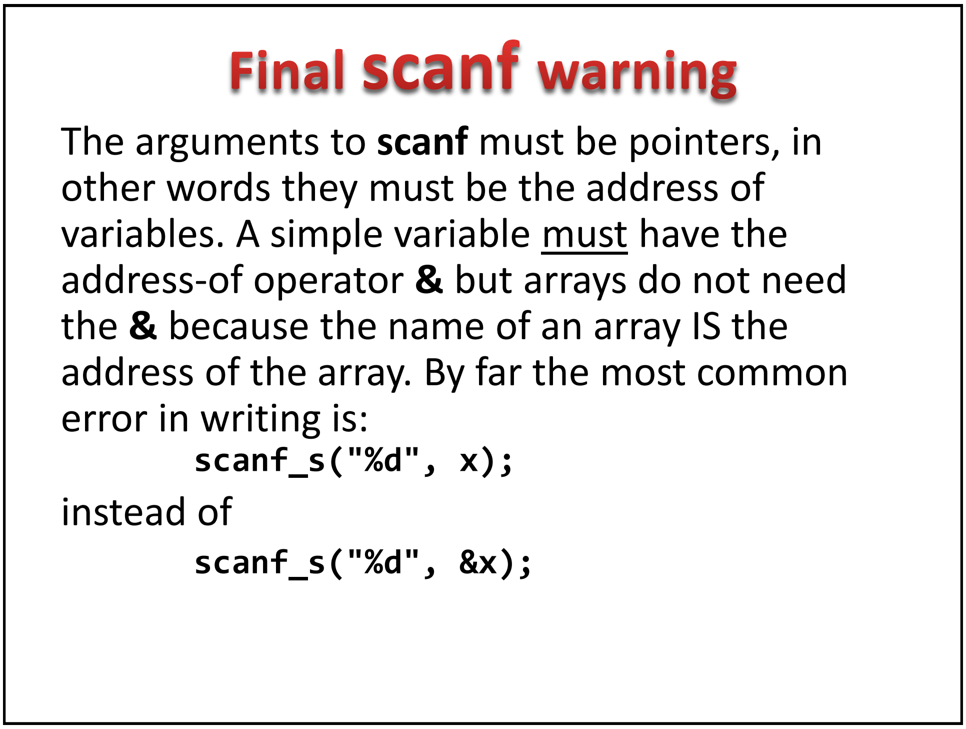
```
char name[20];
scanf ("%s", name);  // read characters into name
```

There is only room for 19 characters plus one more for the string terminator. If the user were to enter more than 19 characters an undetected buffer overrun would occur.

scanf_s uses one more parameter to identify the size of the array. Example:

```
char name[20];
scanf_s ("%s", name, 100);  // array size = 100
```

If **scanf_s** is not implemented on your compiler then you need to use **scanf** and not include the array size



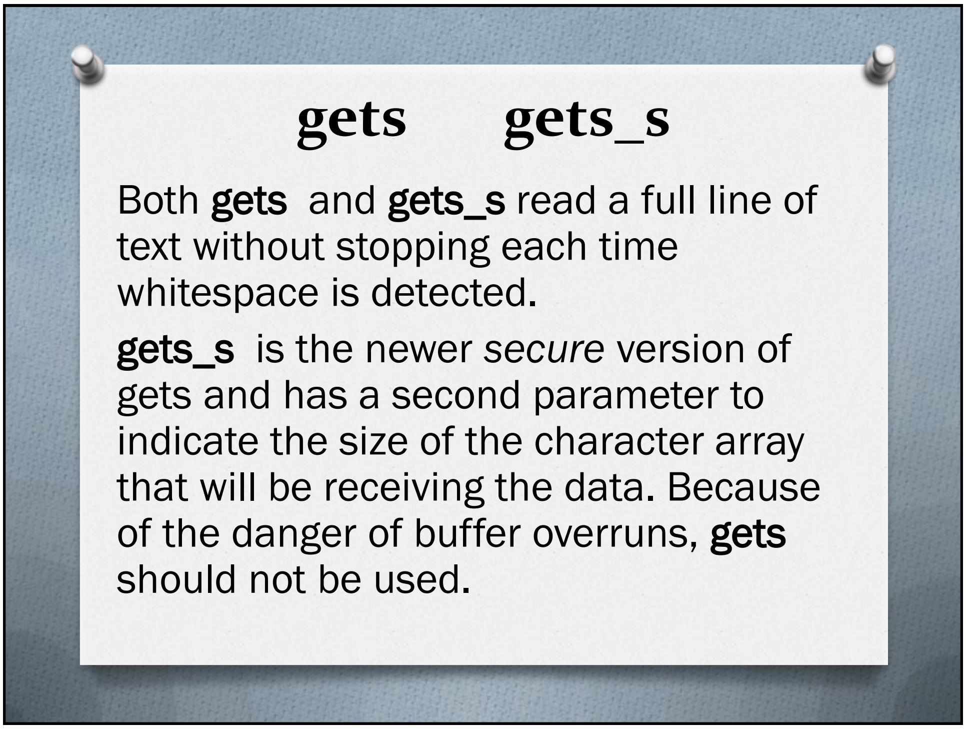
Final scanf warning

The arguments to **scanf** must be pointers, in other words they must be the address of variables. A simple variable must have the address-of operator **&** but arrays do not need the **&** because the name of an array IS the address of the array. By far the most common error in writing is:

```
        scanf_s("%d", x);
```

instead of

```
        scanf_s("%d", &x);
```



gets   gets_s

Both **gets** and **gets_s** read a full line of text without stopping each time whitespace is detected.

gets_s is the newer *secure* version of gets and has a second parameter to indicate the size of the character array that will be receiving the data. Because of the danger of buffer overruns, **gets** should not be used.

gets gets_s

The format for these two functions is:

```
gets (char *);
gets_s (char *,  int size);
```

where:

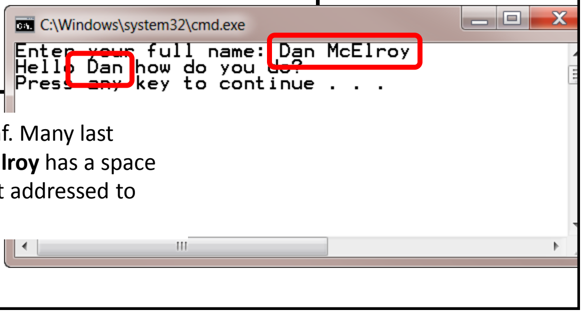
char * is the address of an array of characters that will receive the characters

int size is the size of the array

WARNING The use of **gets** is a common cause of buffer overruns and program crashed. However some C compilers do not have the **gets_s** function and **gets** must still be used.

scanf stops at whitespace

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    // declare the variables
    char fullName[101];   // room for 100 characters

    printf ("Enter your full name: ");
    scanf_s ("%s", fullName, 100);
    printf ("Hello %s\n", fullName);

    return 0;
}
```

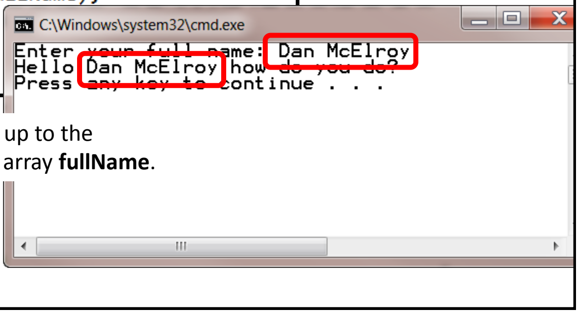


```
Enter your full name: Dan McElroy
Hello Dan how do you do?
Press any key to continue . . .
```

Anything after a space is lost by scanf. Many last names have spaces. Sometimes **McElroy** has a space and becomes **Mc Elroy** so letters get addressed to **Mr. Elroy** instead of **Mr. Mc Elroy**

scanf stops at whitespace
gets_s does not (gets = get string)

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    // declare the variables
    char fullName[101];  // room for 100 characters

    printf ("Enter your full name: ");
    gets_s (fullName, 100);
    printf ("Hello %s\n", fullName);

    return 0;
}
```



```
Enter your full name: Dan McElroy
Hello Dan McElroy how do you do?
Press any key to continue . . .
```

By using **gets_s** all of the characters up to the Enter key are read into the character array **fullName**.