Input Output Functions

What Exactly Is Program Input/Output?

- A C program keeps data in random access memory (RAM) while executing.
- This data is in the form of variables, structures, and arrays that have been declared by the program.
- Where did this data come from, and what can the program do with it?

Contd….

- Data can come from some location external to the program. Data moved from an external location into RAM, where the program can access it, is called input.
- The keyboard and disk files are the most common sources of program input.
- Data can also be sent to a location external to the program; this is called output. The most common destinations for output are the screen, a printer, and disk files.

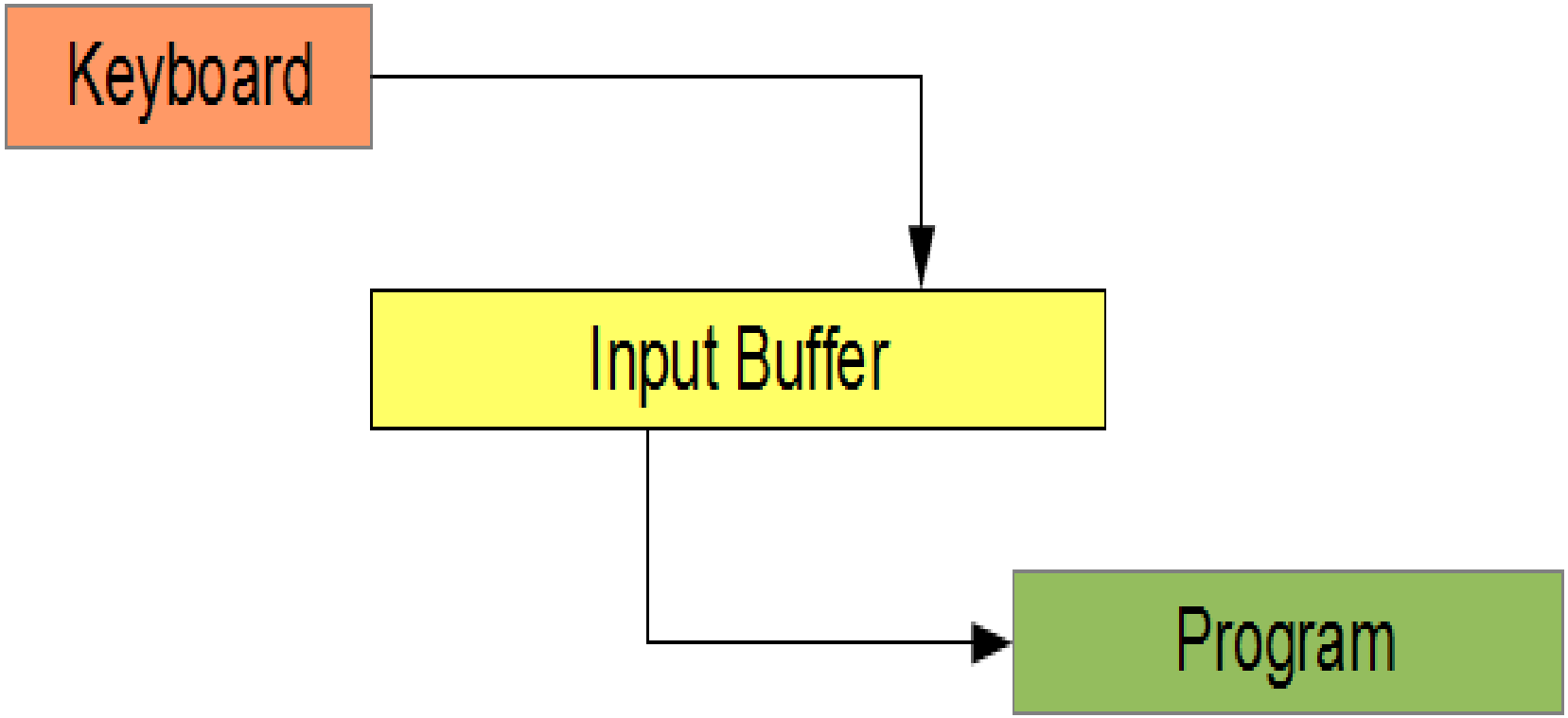
Input Functions

- Some programming languages leave input and output support to the libraries developed for the languages.
- For instance, the core C language does not include input and output specifications. These facilities are available in a set of functions, which are defined in the **stdio** module.
- This module ships with the C compiler. Its name stands for standard input and output.
- Typically, standard input refers to the system keyboard and standard output refers to the system display.
- The system header file that contains the prototypes for the functions in this module is **<stdio.h>**.

Buffer

- Temporary storage area is called buffer.
- All standard input and output devices contain input and output buffer.
- In standard C, streams are buffered, for example in case of standard input, when we press the key on keyboard, it isn’t sent to your program, rather it is buffered by operating system till the time is allotted to the program.

Buffer



Accepting keyboard input

- The character input functions read input from a stream one character at a time.
- When called, each of these functions return the next character in the stream, or EOF if the end of the file has been reached or an error has occurred.
- EOF is a symbolic constant defined in stdio.h as -1.

What Is a Stream?

- A stream is a sequence of characters.
- More exactly, it is a sequence of bytes of data.
- A sequence of bytes flowing into a program is an input stream; a sequence of bytes flowing out of a program is an output stream.

Accepting keyboard input

- Most C programs require some form of input from the keyboard (that is, from the stdin stream).
- Input functions are divided into a hierarchy of three levels: character input, line input, and formatted input.

Accepting keyboard input

- Character input functions differ in terms of buffering .
- Some character input functions are buffered. This means that the operating system holds all characters in a temporary storage space until you press Enter, and then the system sends the characters to the stdin stream.
- Others are unbuffered, meaning that each character is sent to stdin as soon as the key is pressed.

Buffered input

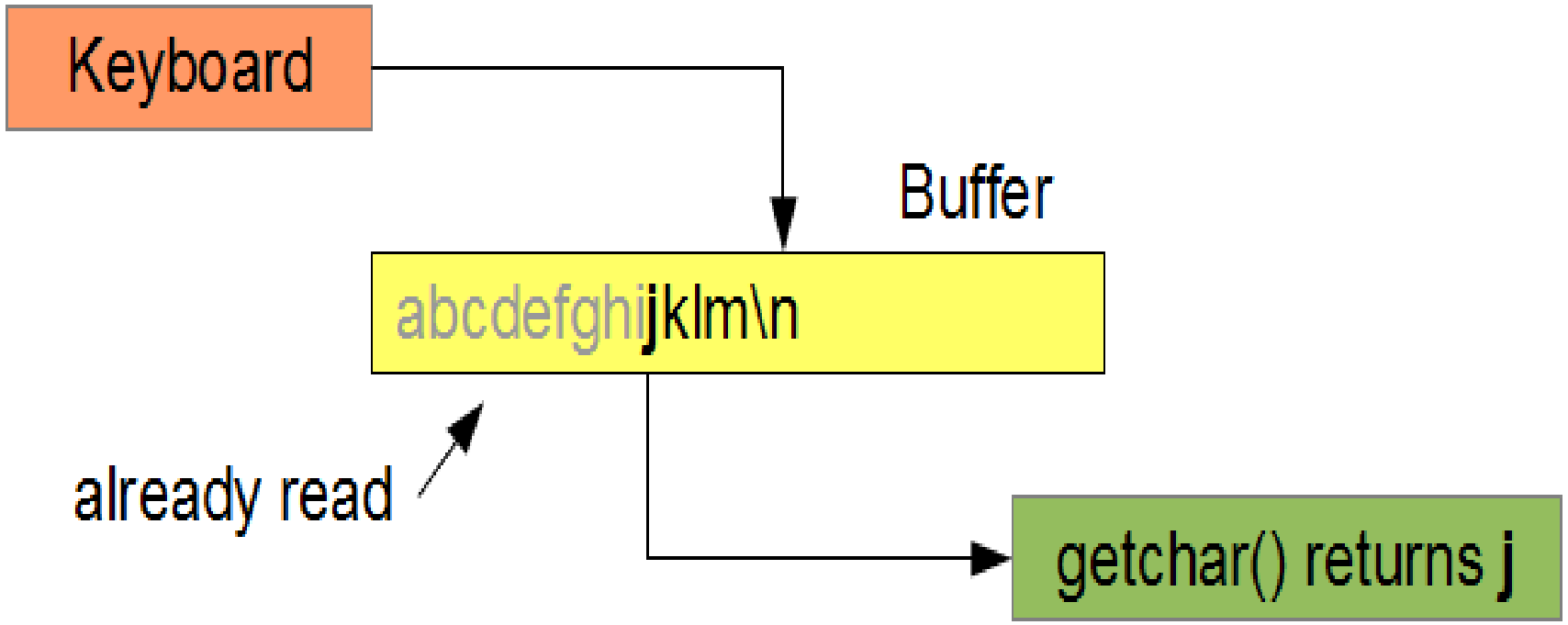
- Two functions accept buffered input from the keyboard (the standard input device):
- **getchar() -** unformatted input
- **scanf() -** formatted input

Unformatted Input

- The function **getchar()** retrieves the next unread character from the input buffer.
- It provides buffered character input, and its prototype is

```
int getchar(void);
```

Contd….


Keyboard
Buffer
abcdefghijklm\n
already read
getchar() returns j

Contd…

- **getchar()** returns either

  the character code for the retrieved character

  EOF
- The character code is the code from the collating sequence of the host computer. If the next character in the buffer waiting to be read is **'j'** and the collating sequence is ASCII, then the value returned by **getchar()** is 106.
- **EOF** is the symbolic name for end of data.

Contd….

- **“ch = getchar( )”** will read a character from the keyboard and copy it into memory area which is identified by the variable ch.
- Once the character is entered from the keyboard, the user has to press Enter key.

Example

```
main()
{
 char ch;
 clrscr();
 ch = getchar();
 putchar(ch);
 getch();
}
```

Formatted input

- The input functions covered up to this point have simply taken one or more characters from an input stream and put them somewhere in memory.
- No interpretation or formatting of the input has been done, and you still have no way to input numeric variables.
- For example, how would you input the value 12.86 from the keyboard and assign it to a type float variable?

Formatted input

- The scanf() function reads formatted input from the keyboard.
- The **scanf()** function retrieves the next set of unread characters from the input buffer and translates them according to the conversion(s) specified in the format string.

The scanf() Function's Arguments

- The scanf() function takes a variable number of arguments; it requires a minimum of two.
- The first argument is a format string that uses special characters to tell scanf() how to interpret the input.
- The second and additional arguments are the addresses of the variable(s) to which the input data is assigned.

Contd….

- Here’s an example: scanf(“%d”, &x);
- The first argument, “%d”, is the format string.
- In this case, %d tells scanf() to look for one integer value.
- The second argument uses the address-of operator (&) to tell scanf() to assign the input value to the variable x.

Contd…

- The only required part of the format string is the conversion specifications.
- Each conversion specification begins with the % character and contains optional and required components in a certain order.
- The scanf() function applies the conversion specifications in the format string, in order, to the input fields.

Contd…

Specifier	Input Text	Convert to Type …	Most Common
`%c`	character	`char`	*
`%d`	decimal	`char, int, short, long, long long`	*
`%o`	octal	`int, char, short, long, long long`	
`%x`	hexadecimal	`int, char, short, long, long long`	
`%f`	floating-point	`float, double, long double`	*

Example

```
#include <stdio.h>
int main()
{
int testInteger;
printf("Enter an integer: ");
scanf("%d",&testInteger);
printf("Number = %d",testInteger);
return 0;
}
```

Contd…

- When user enters an integer, it is stored in variable testInteger.
- Note the '&' sign
  before testInteger; &testInteger gets the address of testInteger and the value is stored in that address.

Handling Extra Characters

- Input from scanf() is buffered; no characters are actually received from stdin until the user presses Enter.
- The entire line of characters then “arrives” from stdin, and is processed, in order, by scanf().
- Execution returns from scanf() only when enough input has been received to match the specifications in the format string.
- Also, scanf() processes only enough characters from stdin to satisfy its format string.

Contd…

- Extra, unneeded characters, if any, remain waiting in stdin.
- These characters can cause problems.

Contd…

- When a call to scanf() is executed and the user has entered a single line, you can have three situations.
- For these examples, assume that
- scanf(“%d %d”, &x, &y); is being executed; in other words, scanf() is expecting two decimal integers.
- Here are the possibilities:

Contd…

- The line the user inputs matches the format string. For example, suppose the user enters 12 14 followed by Enter. In this case, there are no problems.
- scanf() is satisfied, and no characters are left over in stdin.
- The line that the user inputs has too few elements to match the format string. For example, suppose the user enters 12 followed by Enter.
- In this case, scanf() continues to wait for the missing input. Once the input is received, execution continues, and no characters are left over in stdin.

Contd…

- The line that the user enters has more elements than required by the format string. For example, suppose the user enters 12 14 16 followed by Enter.
- In this case, scanf() reads the 12 and the 14 and then returns. The extra characters, the 1 and the 6, are left waiting in stdin.

Contd…

- It is this third situation (specifically, those leftover characters) that can cause problems.
- They remain waiting for as long as your program is running, until the next time the program reads input from stdin.
- Then the leftover characters are the first ones read, ahead of any input the user makes at the time. It’s clear how this could cause errors.
- For example, the following code asks the user to input an integer and then a string:
- puts(“Enter your age.”);

  scanf(“%d”, &age);

  puts(“Enter your first name.”);

  scanf(“%s”, name);

Contd…

- Say, for example, that in response to the first prompt, the user decides to be precise and enters 29.00 and then presses Enter.
- The first call to scanf() is looking for an integer, so it reads the characters 29 from stdin and assigns the value 29 to the variable age.
- The characters .00 are left waiting in stdin. The next call to scanf() is looking for a string. It goes to stdin for input and finds .00 waiting there.
- The result is that the string .00 is assigned to name.

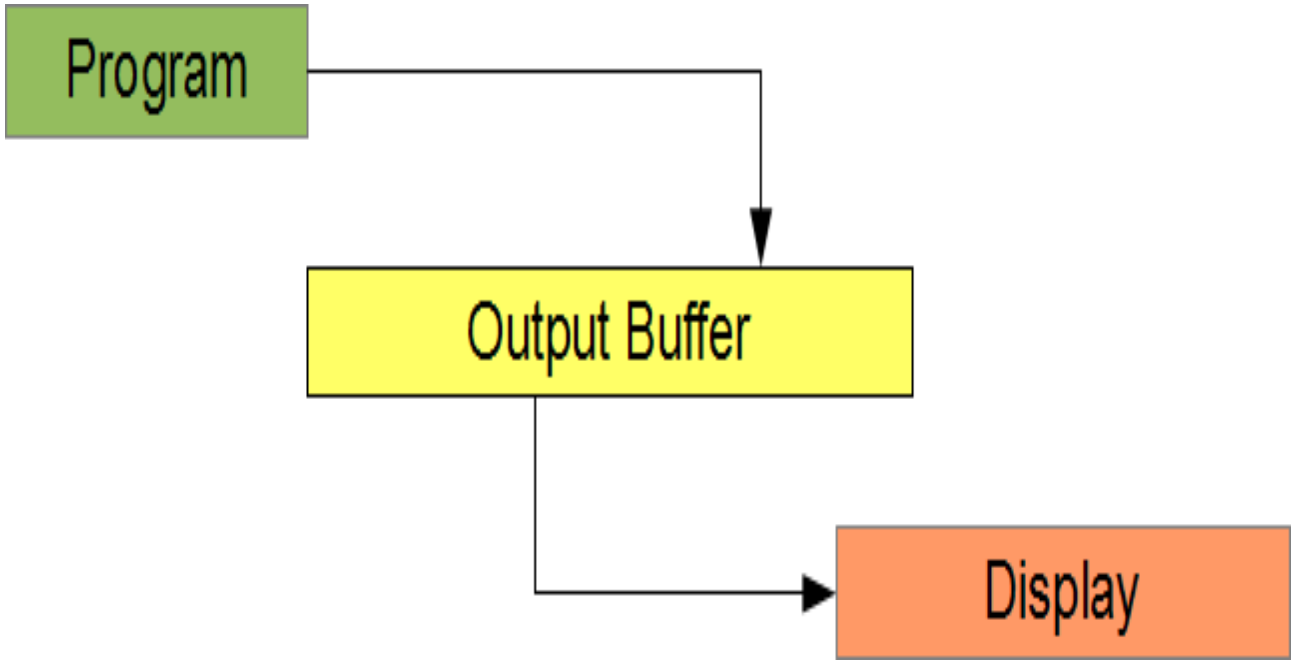
Controlling output to the screen

- The output facilities of a programming language convert the data in memory into a stream of characters that is read by the user.
- The **stdio** module of the C language provides such facilities.
- We will describes two functions in the **stdio** module that provide formatted and unformatted buffered support for streaming output data to the user and demonstrates in detail how to format output for a user friendly interface.

Contd…

- Standard output is line buffered. A program outputs its data to a buffer.
- That buffer empties to the standard output device separately. When it empties, we say that the buffer *flushes*.

Contd…



Contd…

- Output buffering lets a program continue executing without having to wait for the output device to finish displaying the characters it has received.
- The output buffer flushes if
- it is full
- it receives a newline (**\n**) character
- the program terminates

Contd…

- Two functions in the **stdio** module that send characters to the output buffer are
- **putchar()** - unformatted
- **printf()** - formatted

Unformatted output

- The prototype for putchar, which is located in stdio.h, is as follows:
- int putchar(int c);

Contd…

- This function writes the character stored in c to stdout.
- The function returns the character that was just written, or EOF if an error has occurred.

Example

```
#include <stdio.h>
int main(void)
{
putchar('a');
return 0;
}
```

Formatted Output

- So far, the output functions have displayed characters and strings only. What about numbers?
- To display numbers, you must use the C library's formatted output functions, printf().
- These functions can also display strings and characters.

Contd…

- The printf() function takes a variable number of arguments, with a minimum of one.
- The first and only required argument is the format string, which tells printf() how to format the output.
- The optional arguments are variables and expressions whose values you want to display.

Examples

- The statement printf(“Hello, world.”); displays the message Hello, world. on-screen.
- This is an example of using printf() with only one argument, the format string.
- In this case, the format string contains only a literal string to be displayed on-screen.

Contd…

- The statement printf(“%d”, i); displays the value of the integer variable i on screen.
- The format string contains only the format specifier %d, which tells printf() to display a single decimal integer.
- The second argument i is the name of the variable whose value is to be displayed.

Contd…

- The statement printf(“%d plus %d equals %d.”, a, b, a+b); displays 2 plus 3 equals 5 on-screen (assuming that a and b are integer variables with the values of 2 and 3, respectively).
- This use of printf() has four arguments: a format string that contains literal text as well as format specifiers, and two variables and an expression whose values are to be displayed.

Conversion Specifiers

pecifier	Format As	Use With Type ...	Common
`%c`	character	`char`	*
`%d`	decimal	`char, int, short, long, long long`	*
`%o`	octal	`char, int, short, long, long long`	
`%x`	hexadecimal	`char, int, short, long, long long`	
`%f`	floating-point	`float, double, long double`	*
`%g`	general	`float, double, long double`	
`%e`	exponential	`float, double, long double`	

Library functions

- The standard libraries that support programming languages perform many common tasks.
- C compilers ship with the libraries that include functions for mathematical calculations, generation of random events and manipulation and analysis of character data.
- To access any function within any library we simply include the appropriate header file for that library and call the function in our source code.

Mathematical functions

- The mathematics related libraries that contain the more common mathematical functions are:

  stdlib - the standard library

  math - the math library

Standard Library

- The header file **<stdlib.h>** contains prototypes for the functions that perform the more general mathematical calculations.
- These calculations include absolute values of integers and random number generation.

Integer Absolute Value

- abs(), labs(), llabs() return the absolute value of the argument. Their prototypes are

- int abs(int);
- long labs(long);
- long long llabs(long long);

Random Numbers

- **rand()** returns a pseudo-random integer in the range **0** to **RAND_MAX**. **RAND_MAX** is implementation dependent but no less than **32767**. The prototype for **rand()** is

```
int rand(void);
```

Example

- ```
  #include <stdlib.h>
  #include <stdio.h>
  int main(void) {
  int i;
  for (i = 0; i < 10 ; i++)
      printf("Random number %d is %d\n", i+1, rand());
   return 0;
   }
  ```
  ```

Math.h

- The C standard library contains a variety of functions that perform mathematical operations.
- Prototypes for the mathematical functions are in the header file math.h.

Floating-Point Absolute Value

- **fabs()**, **fabsf()**, **fabsl()** return the absolute value of the argument.
- Their prototypes are

```
double fabs(double);
float fabsf(float);
long double fabsl(long double);
```

Floor

- **floor()**, **floorf()**, **floorl()** return the largest integer value not greater than the argument.
- For example, **floor(16.3)** returns a value of **16.0**.

Ceiling

- **ceil()**, **ceilf()**, **ceill()** return the smallest integer value not less than the argument.
- For example, **ceil(16.3)** has a value of **17.0**.

Rounding

- **round()**, **roundf()**, **roundl()** return the integer value closest to the argument.
- For example, **round(16.3)** has a value of **16.0**, while **round(-16.3)** returns a value of **-16.0**.

Truncating

- **trunc()**, **truncf()**, **truncl()** return the integer part of the argument.
- For example, **trunc(16.7)** has a value of **16.0**, while **trunc(-16.7)** returns a value of **-16.0**.

Square root

- **sqrt()**, **sqrtf()**, **sqrtl()** return the square root of the argument.
- For example, **sqrt(16.0)** returns a value of **4.0**.

Powers

- **pow()**, **powf()**, **powl()** return the result of the first argument raised to the power of the second argument.

Logarithms

- **log()**, **logf()**, **logl()** return the natural logarithm of the argument.
- For example, **log(2.7182818284590 45)** returns a value of **1.0**.

Time functions

- The term time refers to the date as well as hours, minutes, and seconds.
- To obtain the current time as set on your system's internal clock, use the time() function.
- The prototype is time_t time(time_t *timeptr)
- Remember, time_t is defined in time.h.

Character

- The ctype library contains functions for character manipulation and analysis.
- Their prototypes are listed in <ctype.h>.

tolower

- **tolower()** returns the lower case of the character received, if possible; otherwise the character received unchanged.
- The prototype isint tolower(int);
- For example, **tolower('D')** returns **'d'**, while **tolower(';')** returns **';'**.

toupper

- **toupper()** returns the upper case of the character received, if possible; otherwise the character received unchanged.
- The prototype is int toupper(int);
- For example, **toupper('d')** returns **'D'**, while **toupper(';')** returns **';'**.