Input and output

So far:

- all programs have generated and printed something
- they have not read any input
- instead, the input has been encoded into the program
- a program which doesn't read input has limited usefulness

Now we will look at:

- how to read some input from the keyboard or a file
- how to write output to the screen or a file

A program is like a function from its input to its output

Input and output are called I/O for short

Input and output issues

Be careful not to confuse I/O with function calling

- a function has parameters which are like its "input"
- a function returns a result which is like its "output"

But the words input and output are reserved for *external* operations

- a program reads input from somewhere outside itself
- a program writes output to somewhere outside itself
- a function *might* perform I/O
- I/O is a global activity

Types of input and output

Input may be:

- from a device connected to the computer
  - text from a keyboard, positions and movements from a mouse
  - graphical from a web-cam, or audio from a microphone
  - requests or responses from the network
- from a file, for persistence between program runs
  - text or database or 'binary' (standard or custom formats)

Output may be

- immediate interaction with the user or other computers
  - text on screen, graphics, audio, network requests/responses
- to a file
  - text, database, standard or custom binary format

Text and binary are supported by the language of standard libraries
Graphics is usually supported by non-standard libraries (except Java)

Output in C

We have seen output already:

```
printf( "%d students, %d staff\n", stud, staff ) ;
```

`printf` prints its first argument, which must be a string, then the **`%`** items are substituted:

- **`%d`** next parameter as a decimal integer
- **`%f`** next parameter as a floating point number
- **`%s`** next parameter as a string
- **`%c`** next parameter as a character
- **`%%`** prints a single **`%`** character
- **`%6d`** next parameter as integer in 6 places (right aligned)
- **`%7.2f`** next parameter as float, 7 total places, 2 decimal places

To find out about the 100's of other possibilities, type **`man printf`** or search for **`printf`** on Google

Input in C: characters

`getchar` returns the next character that is typed

```
int main( void ) {
  char c ;
  while( 1 ) {
    c = getchar() ;
    if( c == '\n' ) { break ; }
    if( c >= 'a' && c <= 'z' ) {
      putchar( c - 'a' + 'A' ) ;
    } else {
      putchar( c ) ;
    }
  }
  putchar( '\n' ) ;
  return 0 ;
}
```

Input in C: characters

```
#include <ctype.h>                /* standard header file */
int main( void ) {
  char c ;
  while( 1 ) {
    c = getchar() ;
    if( c == '\n' ) { break ; }
    if( islower(c) ) {             /* standard function */
      putchar( toupper(c) ) ;/* standard function */
    } else {
      putchar( c ) ;
    }
  }
  putchar( '\n' ) ;
  return 0 ;
}
```

Input in C: characters

```
#include <ctype.h>
int main( void ) {
   char c ;
   while( 1 ) {
      c = getchar() ;
      ...
   }
   putchar( '\n' ) ;
   return 0 ;
}
```

The **`#include`** is not essential (though there can be bugs otherwise):
the standard library is linked anyway, but the compiler now knows the types of the functions

The first **`getchar`** doesn't execute until you finish typing a line
the operating system allows you to edit before hitting enter

Return value of getchar

`getchar` actually returns an integer

- it is the character code of the next character, if there is one or
- the number EOF (End Of File) if there are no more

The constant EOF is not a character (it is usually -1)
You can convert the value of `getchar()` to a character once you know it is not EOF (though it usually isn't necessary)

```
#include <stdio.h> ...                  /* defines EOF */
int i ;                                 /* not char */
while( (i=getchar()) != EOF ) {
   char c = i ; ...                     /* convert */
}
```

When does `getchar` return EOF? When you terminate the input stream to the program by typing ^D (^Z) instead of the next line

Reading integers and floats

```
int stud, staff ;
double money ;
scanf( "%d %d %lf\n", &stud, &staff, &money ) ;
printf( "%d %d %f\n", stud, staff, money ) ;
```

scanf, like **printf**, uses its first argument to decide what to input:

- **%d** read a decimal integer and store it via an integer pointer
- **%lf** read a double (long float) and store via a double pointer
- **%lf** not needed in **printf**, because arguments are converted

do not forget the & to pass the address of a variable
this code doesn't work well for interactive input
the \n discards the newline (necessary, otherwise it will be the first character read in by the next **scanf** or even the next program to run)
but the \n actually reads "any number of whitespace characters", so it looks at the first character of the next line, delaying the **printf**

Reading integers and floats

```
int stud, staff ;
double money ;
char nl ;
scanf( "%d %d %lf%c", &stud, &staff, &money, &nl ) ;
printf( "%d %d %f\n", stud, staff, money ) ;
```

one solution is to read a single character with `%c` into a char variable

Reading integers and floats

```
int stud, staff ;
double money ;
scanf( "%d %d %lf", &stud, &staff, &money ) ;
getchar() ;
printf( "%d %d %f\n", stud, staff, money ) ;
```

another is to use **`getchar`** to read and discard the newline

Reading integers and floats

```
int stud, staff ;
double money ;
char line[100] ;
fgets(line, 100, stdin) ;
sscanf( line, "%d %d %lf", &stud, &staff, &money ) ;
printf( "%d %d %f\n", stud, staff, money ) ;
```

another is to use **fgets** to read the whole line in (including newline) from **stdin** ('standard input', usually keyboard) then use **sscanf** instead of **scanf** to extract the data from the line

in general, this is the best, because validation of the line length and formatting can be added (e.g. check all the return values)

Reading strings

A lazy programmer might write

```
char text[20] ;
scanf( "%s", text ) ;          /* where is the &? */
```

What's wrong?

- `%s` reads in everything up to the next whitespace character
  so it won't read in strings containing spaces
- it doesn't discard the newline (use one of the previous techniques)
- the user may type in a string with more than 19 characters
  then excess characters will go in `text[20]`, `text[21]`, ...
  which overwrites some other variable(s) or even code
  and this was the security loophole used in the 'Internet Worm'
  so never read a string with `%s` unless you can prove it will fit

Other ways to read a string

Limit `scanf` to a maximum field (spaces/newlines are still problems)

```
char text[20] ;                 /* 19 chars and '\0' */
scanf( "%19s", text ) ;
getchar() ;
```

Use `fgets`

```
char text[20] ;
fgets( text, 20, stdin ) ;
text[strlen(text)-1] = '\0'; /* discard newline */
```

Write your own loop in a function:

- pass an array and size to your function, read characters into it
- read characters into a local array, return `strdup`ed version

File I/O in C

File I/O consists of three phases:

- You have to *open* a file, getting a file descriptor
- You may then perform a number of I/O operations
- Finally you have to close a file

```
#include <stdio.h>          /* standard I/O library */
int main( void ) {
   FILE *fd ;

   fd = fopen( "Somefile", "w" ) ;
   fprintf( fd, "Bla %s %d %d\n", "World", 1, 13 ) ;
   fclose( fd ) ;
   return 0 ;
}
```

without **`fclose`**, the last part-full buffer of text may not be written out

File I/O in C

File I/O consists of three phases:

- You have to *open* a file, getting a file descriptor
- You may then perform a number of I/O operations
- Finally you have to close a file

```
#include <stdio.h>          /* standard I/O library */
int main( void ) {
   FILE *fd ;
   int i, j ;
   fd = fopen( "Somefile", "r" ) ;
   fscanf( fd, "%d %d\n", &i, &j ) ;
   fclose( fd ) ;
   return 0 ;
}
```

there are fewer problems with **`fscanf`** than with **`scanf`**

Peculiarities of I/O in C

Output:

- Output in C is buffered (i.e. characters not printed until a newline) so include `\n` in `printf("...\n", ...)` when debugging
- There is a `fflush` function to override this, e.g. after a prompt

Input:

- Input in C is often line buffered (i.e. getchar does not read until you hit return, when the whole line is delivered to your program)
- You need & on variables in `scanf` or `sscanf` or `fscanf`
- You need `"%lf"` when scanning a variable of type double
- Use e.g. `"%19s"` not `"%s"` unless you can prove correctness
- Don't use `gets` unless you can prove correctness
- `fgets` includes the newline, which you can use to check whether the whole line has been read in, but which you may want to discard

A complication

```
int main( void ) {


  if( getchar() > getchar() ) {
    printf("Yes!");
  }
  return 0;
}
```

Run the program with input 06. Will it print 'Yes'?

A complication

```
int main( void ) {
  char c, d ;
  c = getchar() ; d = getchar() ;
  if( c          > d          ) {
    printf("Yes!");
  }
  return 0;
}
```

Run the program with input 06. Will it print 'Yes'?

A complication

```
int main( void ) {
  char c, d ;
  c = getchar() ; d = getchar() ;
  if( c          > d          ) {
    printf("Yes!");
  }
  return 0;
}
```

Run the program with input 06. Will it print 'Yes'?
The order of evaluation of C subexpressions is not fully defined.
So multiple I/O operations mustn't be used in such situations.
Compare with Haskell where I/O operations can't be used like this.
I/O in Haskell is more complicated, but unambiguous.