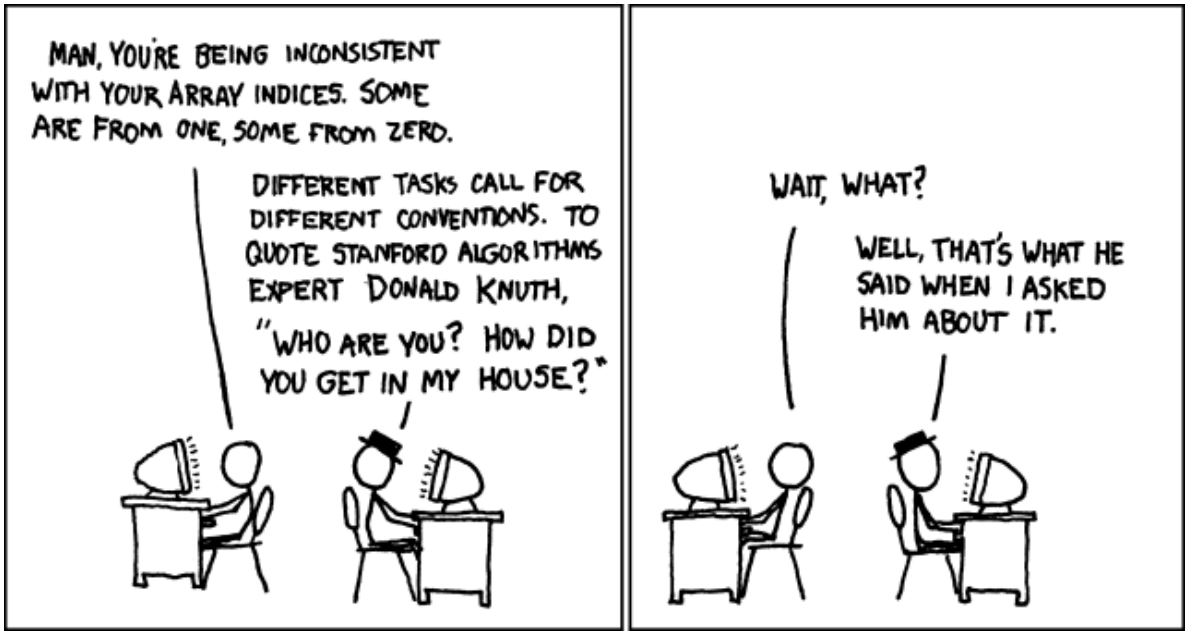
Week 4: Functions and Pointers and Characters

CS211: Programming and Operating Systems

Wednesday and Thursday, 05+06 Feb 2020



This week, in CS211:

1. Output: print()
   - plain text
   - Escape Characters
   - Conversion characters
   - Other output functions
2. Input: scanf()
   - Note:
3. Input Checking
4. Functions
   - Examples
   - `void` functions
   - `return` values
5. Call-by-value
6. Pointers
7. Characters
   - `ASCII.c`
   - Important functions

Output: print()

Part of the `standard input/output` library, the `printf()` function is the most commonly used mechanism for sending ***formatted*** output to the screen.
It is unusual because it many actually take an arbitrary number of arguments:

- a format string,
- followed by zero or more variables,

The format string may include

- plain text, to be sent to `stdout`
- ***escape*** characters,
- conversion characters, to tell the system how variables whose values will be displayed. These are actually a bit complicated, and so we won't be able to describe them in full detail.

Output: print() plain text

To print a simple message, pass you text as the first argument , encapsulated in double quotes:

```
1 printf("This is not a very interesting example");
```

However, usually this first string argument includes ***escape characters*** and ***conversion characters***

The format string in C may contain a number of "*escape characters*". These are represented with a *backslash*, followed by a single letter, and allow `printf` to "display" commonly used characters, but that don't have easy keyboard representations. The most important ones are:

- \a Produces a beep or flash (useful when debugging)
- \b Moves the cursor to the last column of the previous line. (Not that useful).
- \f Moves the cursor to start of next page. (not very useful)
- \n New line. The ***most used***
- \r Carriage Return
- \t Horizontal Tab (quite useful when displaying tables of data).
- \v Vertical Tab (not very useful)
- \\ Prints single \
- \" quotation
- %% Prints %.

A ***Conversion character*** is a letter that follows a % (percent symbol) and tells `printf` to display the value stored in the variable that is next in its argument list. The most common ones are

- `%c` Single **char**acter (i.e., variable of type `char`,
- `%d` **d**ecimal integer (`int`)
- `%e` floating-point value in `E` ("scientific") notation
- `%f` floating-point value (`float`)
- `%g` Same as `%e` or `%f` format, whichever is shorter
- `%o` octal (base 8) integer
- `%s` String of text (`char` array)
- `%u` Unsigned `int`
- `%x` hexadecimal (base 16) integer

These can also take flags that modify their behaviour.

flags

1. Width specifiers
2. Precision specifiers
3. Input-size modifiers

Examples:

Although `printf` is the most versatile function, there are others for displaying output:

- `putchar`
- `putc`
- `puts`

Input: scanf()

The `scanf()` function is analogous to `printf()`: it will

- read input from standard input,
- format it, as directed by a ***conversion character*** and
- store it in a specified address.

```
int i;
char s;
printf("Enter an integer and a char: ");
scanf("%d %c", &i, &s);

printf("The int is %d, char is %c\n",i,s);
```

Input: scanf()

Example

Write a short C programme that reads a single integer from the keyboard, and checks that it's an even number between 1 and 49 (inclusive).

```
int i;
printf("Enter a positive, even integer less than 50: ");
scanf("%d", &i);

printf("You entered %d", i);
if ((i<=0) || (i>=50) )
  printf(", which is *not* between 1 and 49.\n");
else if ( (i%2) != 0)
  printf(", which is in [1, 49], but is *not* even.\n");
else
  printf(". Thank you.\n ");
```

Input: scanf()

Note:

Some other things about `scanf`:

- We usually call the `scanf` function as if its return value is `void`, but it actually returns an `int`eger equal to the number of successful conversions made.
- It has friends `fscanf` that we'll use for reading from files (in fact `scanf` is really just `fscanf` in disguise but with the keyboard as the input "file"), and `sscanf` used for extracting from strings.
- There are other very useful functions for reading from the standard input stream: `getchar`, `gets`

Input Checking

In the last example, we checked that the user inputted that data that was asked for. If we don't include such checks...

NoInputCheck.c

```
int n, i, list[30];
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
for (i=0; i<n; i++)
  list[i] = rand()%40;
```

While this is OK, it can lead to strange results if the user enters a number less than 1 or greater than 30.
So we should check that the user inputs the data correctly...

Input Checking

We could use an `if` statement to improve this:

IfInputCheck.c

```
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
if ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  return(1);
}
```

although it would be better if the user had a chance to enter the data correctly...

Input Checking

So we could ask the user the try entering the data again:

IfInputCheckAgain.c

```
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
if ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
}
```

but this only allows the user to make one mistake. Where we have a persistently dumb user, we need to let them try again, and again, and again...

Input Checking

That is easily achieved by using a `while` loop instead of the `if` expression:

WhileInputCheck.c

```
printf("Enter a number between 1 and 30: ");
scanf("%d", &n);
while ( (n<1) || (n>30) )
{
  printf("\aError:  number not between 1 and 30\n");
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
}
```

Now the programme will keep asking the user to enter the number `until` they get it right.

Input Checking

And as described in out previous lecture, we could also use a `do... while` loop. This lets the loop run once, ***before*** checking that the input was correct. If its not, it repeats the loop.

DoWhileInputCheck.c

```
do
{
  printf("Enter a number between 1 and 30: ");
  scanf("%d", &n);
} while ( (n<1) || (n>30) );
```

Functions

A good understanding of **functions**, and their uses, is of prime importance.
Some functions return/compute a single value. However, many important functions return more than one value, or modify one of its own arguments. In these cases we need to know how to use pointers.

Functions

Every C program has at least one function: `main()`

Example

```
#include <stdio.h>

int main(void )
{
   /* Stuff goes here */
   return(0);
}
```

Functions

Each function consists of two main parts:

- Function "header" or ***prototype*** which gives the function's
  - return value data type, or `void` if there is none, and
  - parameter list data types or `void` if there are none.

  The prototype is often given near the start of the file, before the ***main()*** section.
- ***Function definition***. Begins with the function name, parameter list and return type, followed by the body of the function contained within curly brackets.

Example: averages

```
float average(float a, float b)
{
  return( (a+b)/2);
}
```

Example: a `factorial` function

```
int factorial(int n)   /* Defination */
{
  int i, fac=1;

  for (i=1; i<=n; i++)
    fac = fac*i;

  return(fac);
}
```

Calling the `factorial` function

```
#include <stdio.h>

int factorial( int); /* Declaration */

int main(void )
{
  int i;
  printf("Enter an integer: ");
  scanf("%d", &i);
  printf("The %d! = %d\n", i, factorial(i)); // Call
  return 0;
}
```

We say a function has a `void` argument list if it requires no inputs.
Its return value is `void` if it has no return statement.

Example:

```
#include <stdio.h>
void Banner(void);

int main(void )
{
  /* ...  */
  Banner();
  /* ... */
}
```

```
void Banner(void )
{
  printf("\nThis is intro.c\n'');
  printf("%s%s\n");
    "It prints this message",
    "when the program starts");
}
```

However, most functions to compute some value(s) and need to communicate that with their parent function.

01gcd.c

```
#include <stdio.h>
#include <stdlib.h>

int gcd(int a, int b);

int main(void)
{
  int a, b;
  printf("Enter a and b: ");
  scanf("%d", &a);
  scanf("%d", &b);
  printf("gcd(a,b)=%d\n", gcd(a,b));
  return(EXIT_SUCCESS);
}
```

01gcd.c

```
22 int gcd(int a, int b)
   {
24   int x=a, y=b, r;

26   while(y != 0)
     {
28     r = x%y;
       x=y;
30     y=r;
     }
32   return(x);
   }
```

Call-by-value

In C, it is ***very*** important to distinguish between

- a variable
- the value stored in it.

A good example is as follows: write a C function as follows:

- the function is called `Swap()`
- takes two integer inputs `a` and `b`
- after calling the function, the values of `a` and `b` are swapped.

Call-by-value

Call-By-Value.c

```
void Swap(int i, int j);

int main(void )
{
   int i, j;

   printf("Enter an integer: "); scanf("%d", &i);
   printf("Enter an integer: "); scanf("%d", &j);

   printf("i=%2d and j=%2d\n",i,j);
   printf("Swapping...\n");
   Swap(i,j);
   printf("i=%2d and j=%2d\n",i,j);
```

Call-by-value

```
void Swap(int a, int b)
{
  int tmp;

  tmp=a;
  a=b;
  b=tmp;
}
```

This won't work! We will only have passed the *values stored in the variables i and j*. even if these are swapped in the function, they remained unchanged in the calling function.
What we really wanted to do here was to use ***Call-By-Reference*** where we modify the contents of the memory space referred to by `i` and `j`.

Pointers

A variable has a location in memory. The value of the variable is stored at that location. Example:

```
int i=10;
```

tells the system to allocate a location in memory for storing integers can be referred to as `i`. Furthermore, the value `10` should be stored there.

One of the distinguishing features of C is that we can manipulate the address of the variable almost as easily as changing its value.

Pointers

The important concepts are

- if `i` is a variable, then `&i` is its location in memory.
- The declaration `int *p` creates a variable called `p` that can store the memory address of an integer.
- If a memory address is stored in the variable `p`, then `*p` is the value at that address.

The correct version of the `Swap` function and program is now:

Pointers

Swap_by_Reference

```
void Swap_by_Reference(int *i, int *j)
{
  int tmp;

  tmp=*i;    *i=*j;    *j=tmp;
}
```

This is called as follows

From main

```
   printf("i=%2d and j=%2d\n",i,j);
   printf("Swapping...\n");
   Swap_by_Reference(&i,&j);
   printf("i=%2d and j=%2d\n",i,j);
```

Characters

In C, a `char`atacer is just an unsigned integer; it is how you use it that matters. Each character corresponds to an integer between 0 and 127.

What's so special about 127?

For example, the line

```
printf("%c == %c \n", 'a', 97);
```

will yield the output: `a == a`

Some ASCII codes are given below

32	48	57	65	90	97	122
space	0	9	A	Z	a	z

For more codes: see `ASCII.c`

02ASCII.c

```
#include <stdio.h>

int main(void ) {
  int i, start, step=16;

  for (start=32; start < 127; start+=step)
    {
      printf("\n%12s", "Code:");
      for (i=start; i < start+step; i++)
        printf("%4i", i);

      printf("\n%12s", "Character:");
      for (i=start; i < start+step; i++)
        printf("%4c", i);
      printf("\n");
    }
  printf("\n");
  return(0);
}
```

- `printf("%c", c);` will send the character stored in `c` to the screen.
- `putchar(c);` same as above.
- `scanf("%c", &c);` will take a character form the keyboard input and stored it in `c`.
- `c = getchar();` ditto.

Example: Write a function that takes an character as input and, if that character is lower case, return the corresponding upper case character.

03uppitty.c

```
10 #include <stdio.h>

12 char upify(char c);

14 int main(void ) {
     char c;
16   while( (c=getchar()) != '\n')
         printf("upify( %c ) = %c \n", c, upify(c));
18   return(0);
   }

   char upify(char a)
22 {
     if ((a >= 'a') && (a <= 'z'))
24     return(a - 'a' + 'A');
     else
26     return(a);
   }
```

Strings in C

Next week, we will study ***string***s in C.
Usually **strings** are thought of a collection of letters/characters that make up a word or a line of text.

The C language **does not actually have a `string` data type**.
Instead, it uses arrays of type `char`.

If you make a declaration like:
```
char greeting[20]="Hello.  How are you?";
```
the system stores each character as an element of the array `greeting[]`.

We'll learn more about this next week (or maybe now???).

Strings in C

Some Important Points:

1. In the above example we declared the array to be of length 20. Even though the string contains 19 characters, an extra ***string termination character*** `\0` (backslash zero) is added to show where the end of the string is.
2. Spaces or even new-line characters do not terminate a string. They are treated just like other characters.
3. Declarations are the only time we can use an "equals" to assign a value to a string. At all other times, we can modify the string one character at a time:
   `greeting[0]='H'; greeting[1]='e';` ...
4. Better still use `strcpy()` – the "string copy" function:
   `strcpy(greeting, "Not too bad");`

Strings in C

The `strcpy()` is one of a collection of functions for dealing with strings. Its definition is to be found in the `string.h` header file. More of this later...

Example: *Write a function that determines the length of a string.*

Strings in C

04StrLength.c

```
#include <string.h>
int length(char *);

int main(void )
{
  char greeting[20];

  strcpy(greeting, "Hello. How are you?");

  printf("%s\n", greeting);

  printf("That message was %d chars long.\n",
            length(greeting) );

  return(0);
}
```

Strings in C

04StrLength.c

```
int length(char *str)
{
  int i, length=0;
  for (i=0; str[i] != '\0'; i++)
    length++;
  return(length);
}
```

string.h

Useful functions defined in `string.h` include:

strncpy

```
char *strncpy(char *dest, const char *source, int n);
```

Copies at most `n` character from the string in `source` to `dest`. The advantage is that we won't copy more characters to `dest` than is allowed

Example

```
char Code[6], Name[20]="Operating Systems";
strcpy(Code, Name); // Bad!  Unexpected Results
strncpy(Code, Name, 6); // OK.
```

strcat

`strcat()`: Con**cat**enate two strings, i.e., append one string onto the end of another. E.g,

```
char message1[30]="Hello.";
char message2[30]=" How are you?";
strcat(message1, message2);
```

Now `message1` contains ”Hello. How are you?”;

strcmp

`strcmp(char *s1, char *s2)`: **Comp**ares two stings. It returns an integer:

- 0 if they are the same,
- negative if s_1 is the first alphabetically
- positive if s_2 comes first

Example

```
char Name0[20], Name1[20], First[20];
strncpy(Name0, "Richie", 20);\\
strncpy(Name1, "Dennis", 20);\\

if ( strcmp(Name0, Name1) > 0)
    strncpy(First, Name1, 20);
```

strlen

`strlen` Takes a single (pointer to) a string as its argument and returns an integer equal to its **len**gth minus 1. (**Why -1?**).

strstr

```
char *strstr( char *haystack, char *needle);
```

Searches for the first occurrence of the string `needle` in `haystack`.
It returns a pointer to the start of the matching substring.
Moreover, if `needle` is ***not*** found in `haystack` it returns NULL.

Example:

```
if (strstr(Code, "CS") != NULL)
   printf("%s is a CS course\n", Code);
```

String Output

You all know how to use `printf()` with strings:

```
printf("%s%s\n", "Good morning ", name);
```

or

```
printf("%s%8s\n", "Good morning ", name);
```

In the second example the *field width* specifier is given. This causes the second string to be "padded" so that it takes up a total of 8 spaces. This is useful for tabulated output.

One could also use `puts()`: this prints the contents of a string followed by a new-line character.

String Input

Input is a more complicated issue, but there are three basic methods:

- `scanf("%s", name);` reads a the next "word" from the input buffer (usually the key board) and stores it in the array `name[]`. A word is a sequence of characters that does not include a space, tab or newline character.
- to get more control of the input, you could use `getchar()` within a loop:

```
printf("What is your name? ");
for (i=0;
     (myname[i] = getchar()) != '\n';
     i++);
myname[i]='\0';
```

String Input

- `gets(string)`: this reads a line a input and stores it all (except the `'\n'`) in the array pointed to by `string`. This would be very useful, except that `gets()` is known to be buggy and is best avoided.

From the Linux manual page from `gets()`:

```
BUGS
Never use gets().  Because it is impossible to tell
without knowing the data in advance how many chars
gets() will read, and because gets() will continue to
store characters past the end of the buffer, it is
extremely dangerous to use.  It has been used to
break computer security.  Use fgets() instead.
```

String Input

- `fgets(string, n, stdin)`: reads in a line of text from the keyboard (standard input) and stores at most `n` characters in array `sting`. The new line charater is stored.

Which ever you use is a matter of choise. My preference is always to write functions that use `getchar()` and related functions, particularly if reading from a file.

Exercises

Exercise (Exer 4.1)

Write a short C programme that prompts the user to input an integer, and then uses `scanf` to read that integer.
The program should output the value that the user entered and that `scanf` returns.
Run the program to check what `scanf` will return when

(i) *the user enters an integer;*

(ii) *the user enters a float (with decimal part);*

(iii) *the user enters non-digit character.*

Exercises

Exercise (Exer 4.2)

Write a short C programme that prompts the user to input an integer, i, such that $10 \leqslant i \leqslant 30$.
Use a `while` (or `do... while`) loop so they are repeatedly prompted for this integer until they enter one that is in this range.
Then the program should output an alternating string of zeros and ones of length i.

Exercises

Exercise (4.3)

The `uppitty` function in `02uppitty.c` is a bit trivial, not least because there is a C function, `toupper`, that already does this. Write a variant as follows:

- *Its argument is a pointer to type character.*
- *the function **changes** the character to lower case.*
- *Write a similar function called `downify()` that converts an upper-case character to lower case, but leave all other characters unchanged.*