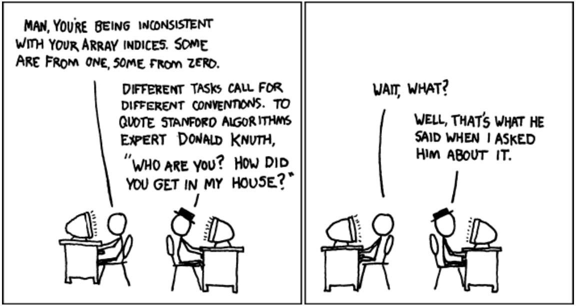
Annotated slides

Week 4: Functions and Pointers and Characters

CS211: Programming and Operating Systems

Wednesday and Thursday, 03+04 March 2021


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

New class times

	Mon	Tue	Wed	Thu	Fri
9 – 10					
10 – 11					
11 – 12					
12 – 1					
1 – 2				***Zoom***	
2 – 3					
3 – 4	**LAB**		***Blackboard***		
4 – 5	**LAB**				

1. The recorded classes on Wednesdays and Thursdays are unchanged.
2. **New lab times: Monday 15:00-17:00**. You aim to attend for an hour. Drop in an out as needed.
3. Little, if any, of the “lab” times will be recorded.
4. Thursday, 4 March: **Will try Zoom:** `https://nuigalway-ie.zoom.us/j/92560272971?pwd=UFlGcHZwN2JkQXdocG1ZOG5HUFYrdz09` (Meeting ID: 925 6027 2971; Passcode: 465580)

This week, in CS211:

1. Part 1: Functions
   - Examples
   - `void`
2. Part 2: Call-by-value, and pointers
   - Pointers

(Wednesday)

3. Part 3: Characters
   - `03ASCII.c`
   - Important functions
4. Part 4: Strings in C
   - `string.h`
5. Part 5: Input and output of strings
   - Output
   - Input

Strings.

(Thursday).

6. Exercises

Part 1: Functions

CS211
Week 4: Functions and Pointers and Characters

Start of ...

PART 1: Functions

Part 1: Functions

A good understanding of **functions**, and their uses, is of prime importance, in C.

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.

All functions return either zero or one values.

Some functions compute more than one value..

Part 1: Functions

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

Example

```
#include <stdio.h>

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

Part 1: Functions

Each function consists of two parts: or declaration

- 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 parameter list can, optionally, include variable names, but these are treated like comments, and ignored.

  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.
  - Unless the return type is `void`, it ends with a `return`.

looks almost the same as the prototype

except – • no semi-colon
• has curly brackets instead.

Part 1: Functions

Examples

We will now look at three examples:

(average) .

- computing the mean of two floats,
- compute the factorial of an int.
- compute the greatest common divisor of two (positive) integers.

Part 1: Functions

Examples

return value.

00mean.c

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

float mean(float, float);    // Prototype

int main(void)
{
  float a, b;
  printf("Enter (floating-point) numbers a and b: ");
  scanf("%f", &a);
  scanf("%f", &b);
  printf("mean(a,b)=%f\n", mean(a,b));
  return(0);
}

float mean(float a, float b)
{
  return( (a+b)/2.0);
}
```

argument list.

function definition.

Notice that the returned value is a float.

Part 1: Functions

01factorial.c

```
int factorial(int n);   // Prototype

int main(void)
{
  int x;
  printf("Enter a positive integer:  ");
  scanf("%d", &x);  // Warning: should do input check
  printf("factorial(%d)=%d\n",
         x, factorial(x));
  return(0);
}

int factorial(int n)   /* Defination */
{
  int i, fac=1;
  for (i=1; i<=n; i++)
    fac = fac*i;
  return(fac);
}
```

"n" is treated as a comment.

$n! = factorial(n) = 1 \times 2 \times 3 \times 4 \ldots \times (n-1) \times (n).$

Part 1: Functions

02gcd.c

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

int gcd(int a, int b); // prototype

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);
}
```

(Also EXIT_FAILURE.

Part 1: Functions

Examples

02gcd.c

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

  while(y != 0)
  {
    r = x%y;
    x=y;
    y=r;
  }
  return(x);
}
```

no semi_colon,

Euclien Alg.

Part 1: Functions

void

In three previous examples, the functions all took one or more arguments, and returned some value.

- Some functions return no values, so the return type is `void`;
- Some functions take no inputs, so the parameter list is `void`;

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");
}
```

Part 1: Functions

`void`

CS211
Week 4: Functions and Pointers and Characters

END OF PART 1

Part 2: Call-by-value, and pointers

CS211
Week 4: Functions and Pointers and Characters

Start of ...

PART 2: Call-by-value and Pointers

(Also : call-by-reference),

Part 2: Call-by-value, and pointers

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.

Part 2: Call-by-value, and pointers

Call-By-Value.c

```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);
```

Part 2: Call-by-value, and pointers

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

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

=2 =3

tmp ← 2

a ← 3

b ← 2

Really Bad:

Eg

=2 =3

```
void Swap(int a, int b)
{
    a=b;   (now a=3)
    b=a;   (b is set to 3)
}
```

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`.

[Pass-by-value]

Reference = "memory address".

Part 2: Call-by-value, and 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.

Part 2: Call-by-value, and 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:

New Operators : & [Gives refence for variable]

*

Part 2: Call-by-value, and pointers

Swap_by_Reference

```
void Swap_by_Reference(int *p, int *q)
{
  int tmp;
  tmp=*p;    *p=*q;    *q=tmp;
}
```

taking pointers as argument.

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);
```

Part 2: Call-by-value, and pointers

If x is a variable, then &x is its memory location.

CS211
Week 4: Functions and Pointers and Characters

END OF PART 2

Finished here Wednesday

If p is a pointer,

- I can store an address in p, eg
  p = &x;
- The value stored at that address is *p.

Part 3: Characters

CS211
Week 4: Functions and Pointers and Characters

Start of ...

PART 3: Characters

Recorder Turns @ 1 pm.

Part 3: Characters

0 or positive (not negative).

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.

so 128 chars $= 2^7$

What's so special about 127?

For example, the line

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

will yield the output: *a* == *a*

So, eg ('A' < 'B') and ('a' > 'A') are true.

Some ASCII codes are given below

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

For more codes: see `03ASCII.c`

To convert, say 'a' to 'A', subtract 32.

Part 3: Characters

03ASCII.c

```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);
}
```

Each row

cols.

Part 3: Characters

Important functions

- `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. (see also getc)

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

Part 3: Characters

03uppitty.c

```c
#include <stdio.h>

char upify(char c);          // Prototype.

int main(void ) {
  char c;
  while( (c=getchar()) != '\n')
      printf("upify( %c ) = %c \n", c, upify(c));
  return(0);
}
//check if a is lower-case...
char upify(char a)                                  Definition.
{
  if ((a >= 'a') && (a <= 'z'))
    return(a - 'a' + 'A');
  else              32
    return(a);
}
```

CS211
Week 4: Functions and Pointers and Characters

END OF PART 3

CS211
Week 4: Functions and Pointers and Characters

Start of ...

PART 4: Strings in C

(Thursday).

→ a list (ie array) of chars

Part 4: Strings in C

Now we will look at **strings**. Usually, these are thought of a collection of letters/characters that make up a word or a line of text.

in order!,

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

greeting[18] = '?'

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[]`.

Part 4: Strings in C

Some Important Points:

Eg, if we set greeting[0]='\0' then it is Empty.

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.

\n

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';` ...

   greeting[2]='l';
4. Better still use `strcpy()` – the "string copy" function:
   `strcpy(greeting, "Not too bad");`

Part 4: 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.*

Part 4: Strings in C

05StringLength.c

```
#include <string.h> // Needed for strcpy

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);
}

int length(char *str)
{
  int i, len=0;

  for (i=0; str[i] != ’\0’; i++)
    len++;

  return(len);
}
```

// Prototyp.

char pointer (so, base address of an array).

String termination char

not equals

Part 4: Strings in C

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.
```

Copy 6 chars, at most, including '\0'.

Part 4: Strings in C

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?”;

Part 4: Strings in C string.h

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

note : can't use == with strings.

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?**).

(Because it does not count the '\0').

Part 4: Strings in C

string.h

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.

"CS211"

Example:

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

Part 4: Strings in C

string.h

To read a string using scanf:

```
char m[10], s[20];

scanf("%s", m),
```

CS211
Week 4: Functions and Pointers and Characters

END OF PART 4

,

Part 5: Input and output of strings

CS211
Week 4: Functions and Pointers and Characters

Start of ...

PART 5: Input and output of strings

Part 5: Input and output of strings

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.

Part 5: Input and output of strings

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’;
```

Part 5: Input and output of strings

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.
```

Part 5: Input and output of strings

- `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 (4.2)

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.*

Exercises

Exercise (4.3)

On Twitter, the satirist John Bull (`@garius`) represents the words of "Gove" using a (seemingly random) mixture of upper- and lower-case text, as in this example from *https://twitter.com/garius/status/1090260422836477952*

> *ThE sNOw gLOwS WhITe oVeR WhitEHall toNIGht*
> *nOT a sTAteSMan tO Be seEn UnITEd KinGDom ISolatED*
> *a PM wHo THInks sHe's a QUEen*

Write a function that takes a `char` array as an argument and "GoVEifies" it by changing letters to upper or lower case at random. You may use the built-in `tolower` and `toupper` functions.