Advanced Pointer & Data Storage

(for ch. 14, 15 18, 19, 26)

18, 19: storage classes
14: Preprocessor & Polymorphism in C)
15 : command line building
26 : stdarg

Contents

PREPROCESSOR & POLYMORPHISM IN C

Read thru Chapter 14 from the Dr. King book.

About token pasting operator

Why:

- Invaluable for generic programming
- Generic customization of variables names in code.
- Argumentably being one way to "confusticate" your source code.

```
#define varName(i)   header##i
#define newName(a,b)  i##a##b

void testVarNames()
{

        for (int i=1; i<5; i++) {
                int varName(i) = i*5;
                printf("%d  ", varName(i));
                //assert( varName(i) < 15);
        }

        int x=10, y = 30;
        int newName(x,y) = x*y;
        printf("%d, %d", newName(x,y), ixy);

}
```

#ERROR

-specified error message at compile time and then terminates the compilation.

```
#if !defined(__cplusplus)
#error C++ compiler required.
#endif
```

STATIC, REGISTER AND EXTERN TYPES

Static: allows data/ information hiding

1) keep a local variable in existence during the life-time of the program instead of being destroyed once outside the scope.
2) The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

The register :
local variables are supposed to be stored in a register instead of RAM, i.e. as it does not have a memory location

Restricted to a register size (usually one word).
cannot used as pointer, reference '&' operator applied to it.

The register should only be used for variables that require quick access such as counters.

CONST AND VOLATILE POINTERS

The const: the pointer cannot be modified after initialization.

The volatile: the value associated with the name that follows can be modified by actions other than those in the user application; such as in share memory shared in multiple processes or global data areas used for communication with ISR (interrupt service routines).

When a name is declared as volatile, the compiler reloads the value from memory each time it is accessed by the program. Thus, the state of an object can change unexpectedly.

Sample 1: To declare the object pointed to by the pointer as const or volatile:

```
const char *cpch;
volatile char *vpch;
```

Sample 2: To declare the value of an a pointer as const or volatile:

```
char *  const  cpch;
char * volatile vpch;

 char ch = 'B';
const char cch = 'A';

const char *pch1 = &cch;  // ok
*pch1 = 'A';                                    // error
pch1 = &ch;                                     // ok

const char *pch2 = &ch;
pch2=  pch1;              // ok
*pch2 = 'A';                                    // error

char *const pch3 =  &ch;    // ok
char *const pch4 =  &cch;   // error
*pch3 = 10;                                     // ok
```

OFFSETOF MACRO

```
size_t offsetof(    structName,     memberName    )    in <stddef.h>
```

The offsetof is a macro, not a function.
It returns the offset in bytes of memberName from the beginning of the structure specified by structName as a value of type size_t.
Example:

```
typedef struct TESTSTRUCT {
        int p1;
        int p2;
                char c1;
                char c2;
}


#define GETAddr(structType, element)  (unsigned long) ( &( ((structType *)NULL)->element ) )

        The address of
int getAddrOfanElement() {

            return GETAddr(TESTSTRUCT, p2);
            // or
            //return offsetof(TESTSTRUCT, c2));

}
```

ASSERT

What is it ?

It is a macro expression for self-checking for any logic violation. This indicates a bug in a program.

Why?

- This is indeed extremely important for robust software.
- well-placed assertion will save you tremendous headache where you are trying to find some difficult bug.
- Guide: assume that assertions only serve the purposes of catching bugs and helping documentation. Particularly useful for bugs that are hard to reproduce.

Where to use it ?

- Exactly what conditions are to be checked by assertions and what by run-time error- checking code is a design issue.
- Assertions are very effective in reusable libraries, because library routines :
    - are supposed to be bug-free,
    - cannot perform error- handling for users, as they do not know in what environment they will be used.

Handy Turning assertions on and off

By default, ANSI C compilers generate code to check assertions at run-time. Assertion-checking can be turned off by defining the NDEBUG flag to your compiler, either by inserting

```
#define NDEBUG
```

in a header file, or by calling your compiler with the -dNDEBUG option:

```
gcc -dNDEBUG ...
```

Syntax & Example

assert (bool expression)

If bool expression == false, assertion message will be triggered, process abort.

Example:

```
#include <assert.h>
...
        assert(x>5);

        like :
        if (  !(x>5) )
        {
                printf("Assertion failed: x>5, file %s, line %d\n", __FILE__, __LINE__);

        }
```

Another example:

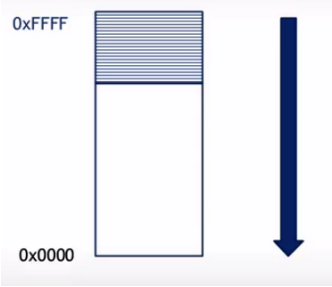
```
        typedef struct
        {
           int  fd1;
           int  fd2;
        } SOME_STRUCT;

        assert(sizeof(SOME_STRUCT) == 8);

        like :

        if (  !(sizeof(SOME_STRUCT) == 8) )
        {
                printf("Assertion failed: size(SOME_STRUCT)==8, file %s, line %d\n", __FILE__,
        __LINE__);

        }
```

STACK AND HEAP MEMORY

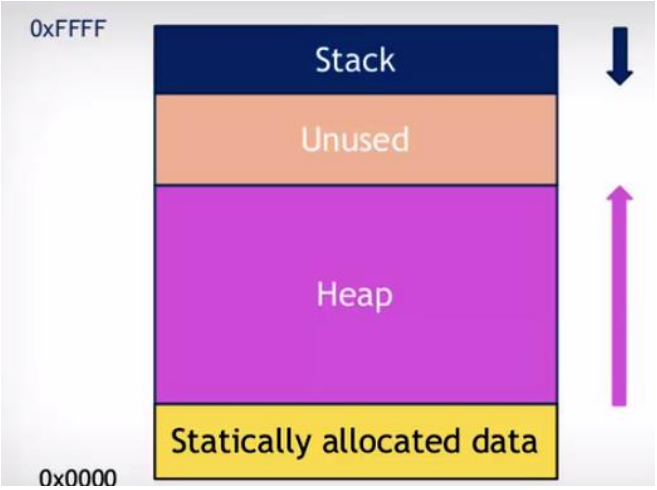
Stack vs. Heap

Stack	Heap
Access via LIFO" (last in, first out) data structure	n/a
the stack grows and shrinks as functions push and pop local variables	Grows and shrinks by malloc() / realloc(), and free() .
Restricted within a local scope, such as { … } or within a single file.	can be accessed globally
Faster than accessing from Heap	Slower than accessing from stack
limit on stack size (can be altered at compilation)	Memory size only limited to the capacity of the cpu
variables cannot be resized. You cannot explicitly de-allocate variables	variables can be resized, reallocated.
All managed by the CPU	Constant allocate and deallocate may cause fragmenation
No pointer structure	Access via pointer
When to use them?	
Relatively short-lived variables	a large block of memory which needs to be kept around a long time, then you should allocate it on the heap.
Memory which remains same	Memory which changes size dynamically

Direction of how stack grows: usually Growing Down



Typical memory block segments



POINTER TO POINTER & FUNCTION POINTER

see the Dr. King Book – chapter 17

VARIABLE LIST OF ARGUMENTS (STDARG)

- define a function which can accept variable number of parameters
- need stdarg.h
- 3 special classes you need to know:
    a. va_list
    b. va_start
    c. va_arg
    d. va_end

```
float average(int num,...) {

  va_list valist;
  float sum = 0.0;
  int i;

  /* initialize valist for num number of arguments */
  va_start(valist, num);

  /* access all the arguments assigned to valist */
  for (i = 0; i < num; i++) {
        sum += va_arg(valist, int);
  }

  /* clean memory reserved for valist */
  va_end(valist);

  return sum/num;
}
```

Ultimately, you can do :

```
function (char *p, ...)
function (float *p, ... )
etc.
```

EXERCISES:

---

1) Write a function which will return all the numbers in a string with the following assumption:
   - all numbers must be delimited by a single character of ',' (i.e. a comma) , except the last one.
   - the may be 0, may be nothing.
   - Each number is separated by a space.

Sample 1:

Input in your code: **char** list[] = "10,20,40,60,100"; Your Output should look like this:

There are 5 numbers. 10 20 60 100.

Note : it should terminate with '.' and each number is separated by a space .

Sample 2:

Input in your code: **char** list[] = "10,0,40,60,,6"; Your Output should look like this:

There are 5 numbers. 10 0 40 60 6.

Sample calling function:

```
Do {
        n = getNext(char **list, ',');
        ... print it out
} while ( .... );
```

---

2) In Arduino Sketch, there is no printf(….) function which will take in variable list of arguments. You will need to write a function to allow you do the following:

Print ("R G B ", x, y, z);

---

3) To take one step further, how about implementing your own version of printf which will be able to perform :

iPrint ("R =? G=? B=? ", x, y, z); // where x, y, z are int type

fPrint ("R =? G=? B=? ", fx, fy, fz); // where fx, fy, fz are float type

---

4) Write a program to reformat an ascii-based csv file to another format. However, you must parse the header and create the variable names based on the header name.
Sample:

Input :

```
fname,lname,team,dob,mentor,league
Amy,Smith,fancy,1/20/2004,Mr. James,Minor
Tod,Grant,fancy,10/5/2006,Mr. James,Minor
James,Stepano,notSoFancy,11/24/2005,Mrs. Logan,Major
Anne,Kim,notSoFancy,6/9/2007,Mrs. Logan,Major
Ben,Cloud,notSoFancy,1/2/2004,Mrs. Logan,Major
```

Output:

```
name,team,league
Amy Smith,fancy,Minor
Tod Grant,fancy,Minor
James Stepano,notSoFancy,Major
Anne Kim,notSoFancy,Major
Ben Cloud,notSoFancy,Major
```