

ELEC3730 Embedded Systems Lecture 6: Memory Management

- Dynamic Memory Allocation
- Register allocation
- Recursion and memory use



Allocating and releasing dynamic memory

```
void *malloc(int bytes) :
```

- Allocates an un-initialized block of memory from the "heap" and returns a pointer to the first byte.
- The pointer is usually recast and saved in a pointer to data of the desired type.
- A **NULL** pointer is returned when insufficient heap space is available.
- **Advantage:** Persistence of static allocation and memory conservation through reuse.
- **Disadvantage:** Programmer's responsibility to release memory; failure to do results in a slow "memory leak" that can be very difficult to debug.

```
void free(void *pointer2block) ;
```

- De-allocates the block and returns it to the heap for reuse.
- Don't use contents of block after release!

Using Dynamic Memory

```
#include <stdlib.h>          /* required to use malloc and free. */
#define NULL ((void) 0)    /* may not be defined in stdlib.h */
...
typedef ... ANYTYPE ;
...
ANYTYPE *p ;
p = (ANYTYPE *) malloc( sizeof(ANYTYPE) ) ; /* object is created */
if (p == NULL) Error("Insufficient heap space");
...
... The object is initialized and used here via the pointer.
...
free(p) ;                    /* object is destroyed */
```



Fragmentation and `alloca()`



- Example: Need 500 bytes
- 836 free bytes **total** available in heap
- Largest single free block is 324 bytes
- Allocation fails.
- Solution (partial) - use `alloca()` :
 - *Location:* Memory is allocated from the stack (like automatic).
 - *Release:* "Automatic" upon return from function that called `alloca`.
- **Advantages:**
 - Very fast: simply adjusts the stack pointer register.
 - Release is automatic (no "memory leaks").
- **Disadvantages:**
 - No indication of allocation failure.
 - No persistence of values (just like automatic).
 - Not a standard feature of C.

Using `alloca()`

```
FILE *OpenFile(char *name, char *ext, char *mode)
{
    int size = strlen(name) + strlen(ext) + 2 ;
    char *filespec = (char *) alloca(size) ;
    FILE *fp ;
    sprintf(filespec, "%s.%s", name, ext) ;
    fp = fopen(filespec, mode) ;
    if (fp != NULL) return fp ;
    printf("Can't open file: %s!\n", filespec) ;
    return NULL ;
}
```

Stack pointer is decreased by "size" bytes.

Stack pointer is automatically restored on any return, releasing the memory.

Recursion & Memory Allocation

```
void PutHex(unsigned n)
{
    static char digits[] =
        "0123456789ABCDEF" ;

    if (n > 0xF) PutHex(n / 16) ;
    putchar(digits[n % 16]) ;
}
```

- Recursive functions call themselves.
- Each call allocates new memory for parameters and local automatics
- Multiple sets of locals and parameters may exist at the same time.

Recursion and Memory Allocation

```
#include<stdio.h>
void puthex(unsigned n)
{
    static char digits[]="0123456789ABCDEF";
    if (n>0xF)
    {
        printf("going in\n");
        puthex(n/16);
    }
    printf("coming out: %d\n",n);
    putchar(digits[n%16]);
    printf("\n");
}

int main(void)
{
    puthex(1234);
}
```

```
csb> gcc -o test3 test3.c
csb> ./test3
going in
going in
coming out: 4
4
coming out: 77
D
coming out: 1234
2
csb>
```

Recursive Calls & Allocation

Enter PutHex(0x123);	0123	?	?
Enter PutHex(0x12);	0123	0012	?
Enter PutHex(0x1);	0123	0012	0001
Exit PutHex(0x1);	0123	0012	released
Exit PutHex(0x12);	0123	released	released
Exit PutHex(0x123);	released	released	released

The register Keyword

- Example:
`register unsigned index ;`
- Outside Functions: Illegal.
- Inside Functions: Attempts to change allocation from automatic (default) to register for faster access; cannot apply "address-of" operator.
- Advantage: Faster Access to Value
- Disadvantage: Knowing when to use register allocation is not obvious.
 - Register allocation is local within each function.
 - When another function is called all register variables must be preserved.
 - This overhead often offsets any performance gain.