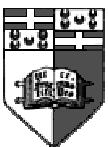




Proper C Programming

- C does not hand hold you
- Segmentation faults and core dumps are common in every program
- Kernel is God, you just abide
- Yet proper programming can make wonders
- First consider coding the problem, then make it more efficient
- System programming is making powerful tools, good system programming is making it efficient

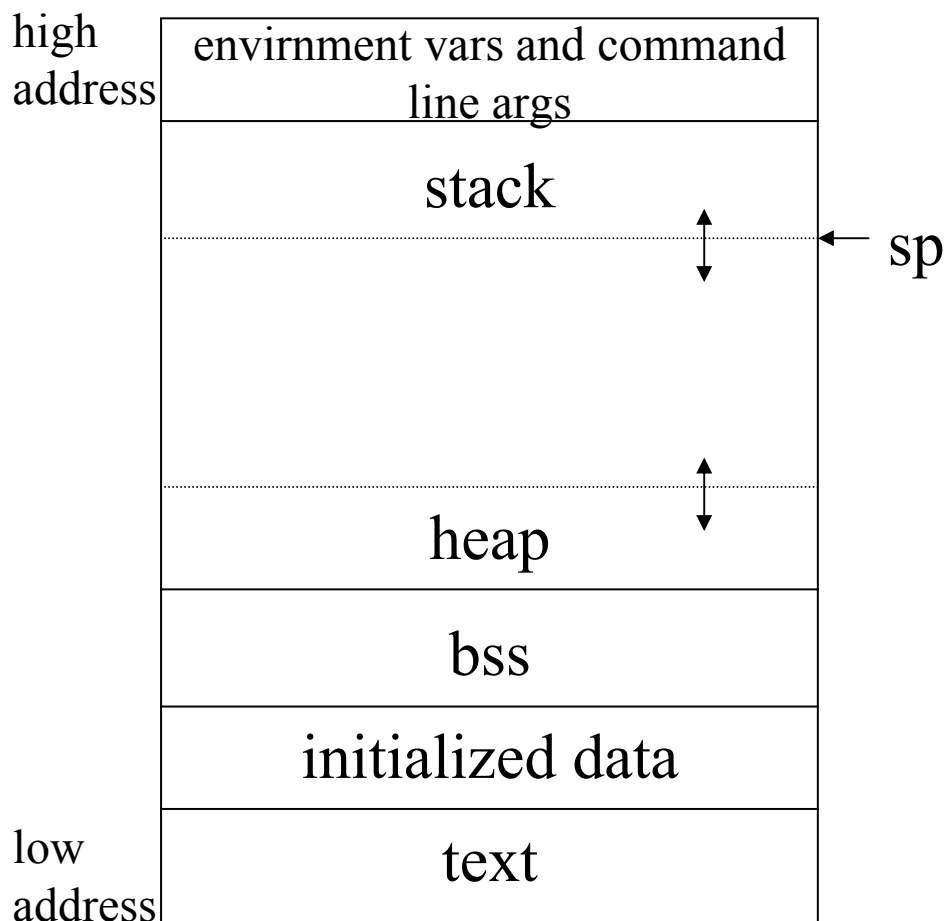




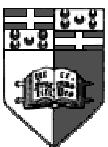
Process Structure

- A process is a ‘program’ with allocated memory space, privileges, rights and identification structure

Memory Layout of a Process



- Stack holds automatic variables, return address of function, function arguments and others





Pointers

- Automatic variables are placed on the stack and have a limited scope
- Declaring a pointer variable does not allocate space for the pointed space.
- Declaring a pointer variable allocates space for the pointer but not the pointed object.
- malloc() reserves space in byte units on the heap which can be used until freed.
- free() releases the allocated space.

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

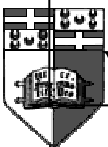
```
void *malloc(size_t size);
```

```
void *calloc(size_t nobj, size_t size);
```

```
void *realloc(void *ptr, size_t newsz);
```

return NULL on error

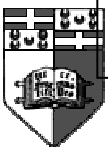
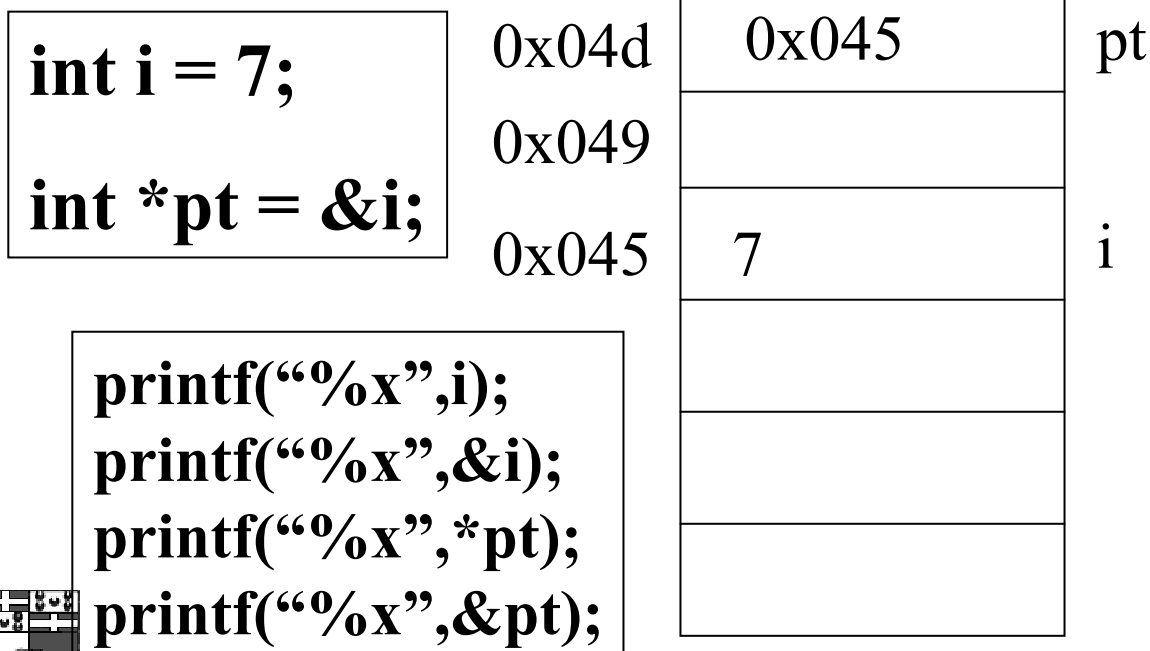
```
void free(void *ptr)
```





Pointers (cont)

- All memory allocation function call the *sbrk* system call
- *void* is a catch all datatype which must always be typecast for data manipulation
- Pointers store the memory location of the object while the dereference operator (*) accessed the pointed object. The address operator (&) gives the memory address of variables.





Dangling Pointers

```
int *pt;
```

```
// segmentation fault warning !!!
```

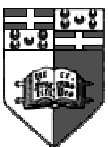
```
*pt = 5;
```

- *pt* points to a random place which is probably outside the process memory space, core dumped !!

```
int *pt=(int *) malloc(sizeof(int));
```

```
*pt = 5; /* OK !!! */
```

- Space is now properly reserved
- Reserving space at infinitum leads to a memory leakage.
- Writing beyond the allocated space is usually catastrophic





Automatic Variables

- When entering a function, space is allocated on the stack for all the variables in the function.
- On exit this space is no longer accessible.

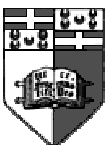
```
char *foo(void) {  
    char c;  
    return &c;  
}
```

```
void foo1(void) {  
    char *pt = foo();  
}
```

- Accessing *pt will crash the program
- malloc'ed data space is scope free

```
char *foo(void) {  
    return (char *)malloc(1);  
}
```

- foo1() is now ok

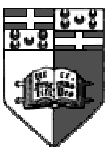




Automatic variables (cont)

- To reserve data on the stack instead of the heap (just like automatic variables) use *alloca()*
- On exit from a function the stack pointer is reversed and stack area is freed automatically

```
#include <stdlib.h>  
void *alloca(size_t size);  
        returns pointer or NULL on error
```





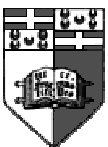
Pointer Arithmetic

- Pointers can be added and subtracted to integers. The result depends on the pointer type size.

sizeof(char) = 1

sizeof(int) = 4

- `int *pti = 0x050;`
`char *ptc = 0x050;`
`ptc++; pti++;`
- `ptc` is now `0x051` and `pti` is now `0x054`





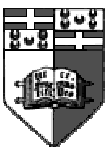
Pointers and Arrays

- There is no such things as arrays in C. They are simply pointers.
- Arrays are a compiler construct.
In fact

$$a[n] = *(a+n)$$

- Name of an array returns the address of the first element in the array.

```
int a[10];
a[0] = 5; a[1] = 9;
printf("%x",a); -> 0x67
printf("%x",*a); -> 5
printf("%x",a+1); ->0x6a
printf("%x",*(a+1)); ->9
```

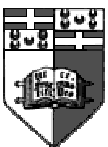




Pointers and Arrays (cont)

- Array names are not Lvalues
unlike pointers

```
char ca[5];
char *cp;
char c = 't';
ca = &c; WRONG
cp = &c; OK
```
- Array names do not have a
memory location themselves,
they exist only until compilation
- Use pointers instead





String, Arrays and Pointers

- Strings are arrays of characters ending with the character ‘/0’
- Thus “hello” is an array of char 0..5 of size 6.
- Pointers can also point to strings

char *c = “Hello”

- Space is reserved in the data section which is part of the executable file

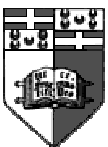
***c** equals ‘H’

c[0] equals ‘H’

***(c+4)** equals ‘o’

***(c+5)** equals ‘/0’

***(c+6)** does not exist and is undefined





String, Arrays and Pointers (cont)

```
char *c = "Hello";
```

```
char *d = c;
```

- This does not copy the string but merely makes two pointers to the same string

- To copy a string

```
char *c = "Hello";
```

```
char *d = malloc(6);
```

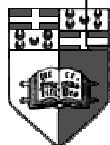
```
strcpy(d,c);
```

- This copies "Hello" on to it with *d* pointing to it

```
#include <string.h>
```

```
char *strcpy(char *s, char* ct);
```

```
return s
```





Type Casting

- While type casting could get quite messy, C gives you the power to treat any variable as any other data type.

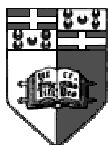
```
char c[4];  
int *i = (int *)c;  
*i = 25;
```

- Allows us to access data as needed.
- Typecasting provides the necessary abstraction, for examples from struct to char and back.
- typedef can hide these messy typecasts

```
typedef int (*PFI)(char *,char *);
```

- Pointer to function that takes two char* and returns an integer !!!

- Another alternative is #define





Time

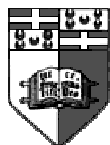
- UNIX provides a time accountability system at different resolutions
- The epoch is 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). (calendar Time: **time_t**)
- Time can be split up in system CPU time and user CPU time

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buf);
           returns clock time in ticks, -1 on error
```

```
struct tms {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
}
```

- To convert ticks to sec divide by `_SC_CLK_TCK` returned from the *sysconf* function. (process time: **clock_t**)





Time (cont)

```
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
```

```
time_t time(time_t *calptr);
    returns time in seconds since the epoch
```

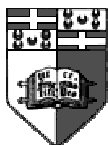
```
int gettimeofday(struct timeval *tv, struct
    timezone *tz);
    returns -1 on error
```

```
struct tm *gmtime(const time_t *calptr);
    return pointer to time structure
```

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_day;
    int tm_isdst;
};
```

```
struct timeval {
    long tv_sec;
    long tv_usec;
}

struct timezone {
    int tz_minuteswest;
    int tz_dsttime;
}
```





Exercises

- Experiment with the examples given previously.
- *malloc* data and display the pointer address given.
- Benchmark one of your lengthy C programs.
- Now run several other processes in the background which run an infinite loop. Does your original program take longer. What is the difference between the time and times values returned ?
- Display the current date and time for every second that passes. You have created a clock!!

