Week 4

- The Stack
- Memory
- Structures
- Pointers
- Lab 2

The Stack

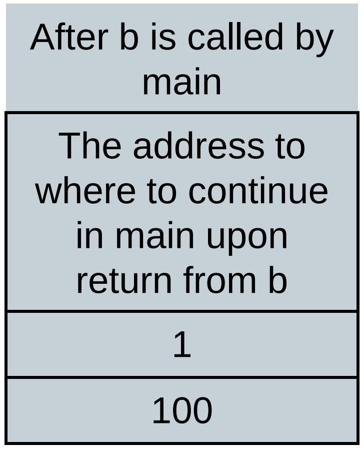
- What is the stack?
- What is its limits?
- What happens when I ignore it?
- Not all hardware really supports a stack or it does with limitations also this varies depending on the compiler as well. Sometimes the compiler does this for the hardware.

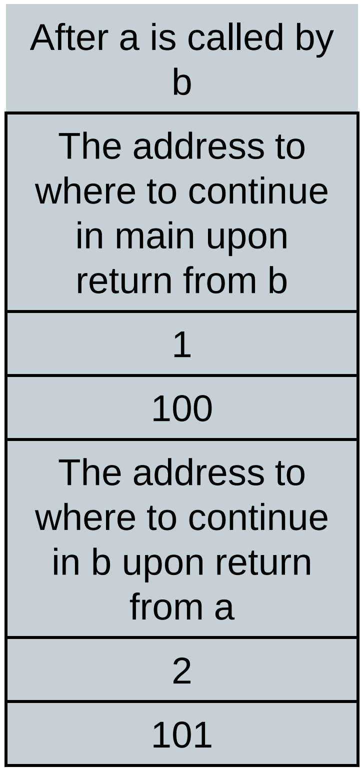
- The examples are typical but vary a bit.

What is the stack?

- The stack is the memory that functions use to pass parameters and store local variables.
- For Example…

```
int a(int parm1,int parm2)
{
  return parm1+parm2;
}
int b(int parm1,int parm2)
{
  return a(parm1+1, parm2+1);
}
void main(void)
{
  int temp;
  temp b(1, 100);
}
```


After b is called by main
The address to where to continue in main upon return from b
1
100


After a is called by b
The address to where to continue in main upon return from b
1
100
The address to where to continue in b upon return from a
2
101

The Stack - What is its limits?

- Stack size is limited. The limits in most embedded systems are created by you but are not endless.
- Stack size in most embedded systems is configured on a task by task (or process by process) basis.
- The stack may be used by code which is not yours, such as interrupts or systems events. Be cautious.

The Stack - What happens when I ignore it?

- Ignore it and your code may blowup.
- You may get weird results.
- You may get inconsistent results.

Avoid problems

- Don't ignore the compiler warnings.
- Prototype all your functions fully.
- Set the compiler to picky mode.
- Never pass structures, always pass pointers to them.
- Assume the stack is small. But, do not make the world global.

Where else does memory get used?

- Heap – mallocs and frees
- Program memory – may be in ROM or may be loaded. This is your code and constants. Sometimes this are separated.

- Memory also comes in two forms - initialized and un-initialized. Always assume un-initialized unless you are sure.

Those crazy structures

- Bit fields
- Byte alignment
- Packing
- Linked lists

Structures - Bit fields

```
struct
{
  int a:1; /* this field is 1 bit wide */
  int b:6; /* this field is 6 bits wide */
  int c:1; /* this field is 1 bit wide */

} myStruct;
```

- Bit fields inside structures are strictly for software (They have no guaranteed representation in memory).
- Bit fields may not be consistent across compilers.

Unions – They are there but be warned

```
union
{
  struct
  {
      int a:1; /* this field is 1 bit wide */
      int b:6; /* this field is 6 bits wide */
      int c:1; /* this field is 1 bit wide */
  } b;
  unsigned char reg;
} myunionstruct;
```

Unions Part 2

```
union
{
   struct
   {
          int a;
          int b;
   } part1;
   struct
   {
          char a;
          char b;
          char c;
   } part2;
} myunion;
```

- What size will this be in memory?
- Unions like bitfields are strictly a software convenience the they may not save memory and their order and representation in memory is not guaranteed.
- If you wish to use them for convenience that is safe if you expect a representation in memory “Don’t”.
- Guess what you can’t tell it will differ by processor, compiler or even setting within the compiler.
- Don’t count on any consistency in the way structures are represented in memory.
- Stay away from unions if you count on any representation in memory.

Structures - Byte alignment

```
struct
{
  char a; /* this field
  is 1 byte wide */
  int b; /* this field is
  4 bytes wide */
  short c; /* this field
  is 2 bytes wide */

} myStruct2;
```

- The fields may be in memory like any of the following.

a			
b	b	b	b
c	c		

a	b	b	b
b	c	c	

a		b	b
b	b	c	c

Structures - packing

- When packing is supported packing forces. Usually via a keyword packed.
- However packing is NOT guaranteed to exist in all compilers and when is does exist it not always consistent.
- Some compilers do packing down to the bit others do not.
- Some processors don't handle unaligned data very well other do fine.
- Good practice when defining structures is to group variable types. (at least group by variable sizes if you know them) ALL EMBEDDED PROGRAMMERS SHOULD HAVE KNOWLEDGE OR THE VARIABLE SIZES FOR THEIR PROCESSOR/COMPILER

Do not use packing if you expect you code to work across multiple platforms. Pack it yourself by using arrays and other tools.

a	b	b	b
b	c	c	

Pointers

- &data – is the pointer (address of data)
- *pData – is the value that is pointed to by the address stored in pData;

```
void examplePointer(void)
{
   int data=0;     /* data is an integer */
   int *pData;     /* pData is a pointer to an integer */

   pData = &data; /* pData is now set to the address of data */

   data++;         /* data is now 1 */
    (*pData)++;      /* data is now 2 since we incremented what
                     pData pointed to not pData */
    pData++;         /* data is still 2 however pData no longer points to
                        data */
}
```

Pointers (continued)

```
void examplePointer2(void)
{
   int data=0;          /* data is an integer */
   int adata[20];       /* array of 20 integers */
   int *pData;          /* pData is a pointer to an integer and in not
                           initialized */
   int *pData2;         /* pData2 is a pointer to an integer and in not
                           initialized */
    int *pData3;        /* un-initialized integer pointer */

   pData = &data;       /* pData is now set to the address of data and is
                           initialized */
   pData3 = adata;      /* What happened to the address operator? */

   pData2 = (int *)malloc(sizeof(int)*10); /* pData2 is now set to the address of
   data
                    and is initialized if the value is not NULL (0).
                   This can be used to create arrays of variable size */
   if (pData2 == NULL)
          return;        /* Error can't initialize pData2 */

}
```

Never write to an un-initialized pointer.

Functions – Pass by Value/Pass by Reference

C passes every by value so to pass by reference you need to pass an address

- func1(a,b) – pass by value
- func2(&a,&b) – pass by reference

In general pass large structure by passing their address.

```
void func1(int a,int b) /* example of the second call */
{
    int a_internal = a;
    int b_internal = b;

    b = 12; /* changes b only in this code NOT the calling code*/
}

void func2(int *a,int *b) /* example of the second call */
{
    int a_internal = *a;
    int b_internal = *b;

    *b = 12; /* changes b in the calling code */
}
```

Pointers to Pointers

```
void modifycptr(char **ptr2cptr)
{
      **ptr2cptr = ‘c’;  /* modifies the c in function main to ‘c’ */

      *ptr2cptr = NULL; /* modifies the cptr to NULL */

      ptr2cptr = NULL; /* changes the pointer locally but has not impact on cptr or c  */
}

void modifyc(char *ptr2c)
{
      *ptr2c = ‘b’;  /* modifies the c in function main to ‘b’ */

      ptr2c = NULL; /* changes the pointer locally but has not impact on cptr or c*/
}

void main()
{
      char c = ‘a’;
      char *cptr;

      cptr = &c; /* set the address of cptr to point to c */

      modifyc(&c);  /* or modifyc(cptr); - either with set c to ‘b’ */
      modifycptr(&cptr);  /* will set c to ‘c’  and then modify cptr to a NULL */
}
```

Incrementing Pointers

- Pointers are always manipulated in units of values they are pointing to.

```
void examplePointer3(void)
{
   int *pData;           /* pData is a pointer to an integer and in not
                                initialized*/
   char *pByte;

   pData = (int *)malloc(sizeof(int)*10); /* pData is now set to the address of data
                                            and is initialized if the value is not NULL (0) */
   if (pData = NULL)
          return;     /* Error can't initialize pData */

   *pData = 0x1234; /* sets the value pointed to by pData */

   pData++;          /* increments the address of pData by the sizeof an int */
   pData = pData + 5;/* increments the address of pData by the sizeof an int times 5 */

   /* want to do something odd I want to decrement the pointer by 5 (not by 5 time size
       of an int */

   pByte = (char *)pData;

   pByte += 5;
}
```

Warning: Not all pointers are the same

- How large is a pointer?

  On some systems it is 32 bits (4 bytes)

  On other systems it is 64 bits (8 bytes)

  And on some microcontroller it is 16 bits (2 bytes)

- Are the pointers physical addresses or virtual address?

  Remember software applications can use virtual address but hardware want physical addresses. So conversion may be required.

Pointers and Arrays

- An array is a pointer which is initialized and also has space allocated for it.

A simplistic way of representing it.

```
int array[10];

is similar to…

int *array;
array = malloc(sizeof(int)*10);
```

Except where memory is created is different.

Pointers and Arrays (Con’t)

```
void examplePointer4(void)
{
   int data[10];      /* data is an integer */
   int *pData;        /* pData is a pointer to an integer and in not
                         initialized */
   pData = &data[0]; /* pData is now set to the address of data[0] and is
                         initialized and is now the same as data */

   pData[1] = 0;
   /* is the same as */
   data[1] = 0;
   /* another way of representing the same thing is */
   *(data+1) = 0;
   /* or */
   *(pData+1) = 0;
}
```

Linked lists (Quick Example)

- A linked list is a structure which includes a pointer to the next structure in the list and possibly the previous structure.

```
typedef struct mystruct
{
    struct mystruct *next;
    int stuff;
} MYLISTSTRUCT;

MYLISTSTRUCT *topoflist, *lastoflist;

void buildlist(void)
{
    int addthismany = 4;

    topoflist = (MYLISTSTRUCT *)NULL;
    lastoflist = (MYLISTSTRUCT *)NULL;

    for (/* addthismany is already initialized to 4 */ ;addthismany > 0; addthismany--)
    {
              if (topoflist == (MYLISTSTRUCT *)NULL)
              {
                            topoflist = (MYLISTSTRUCT *)malloc(sizeof(MYLISTSTRUCT));
                            lastoflist = topoflist;
                            if (lastoflist == (MYLISTSTRUCT *)NULL)
                                          return; /*ERROR */
              } else {
                            lastoflist -> next = (MYLISTSTRUCT *)malloc(sizeof(MYLISTSTRUCT));
                            if (lastoflist -> next == (MYLISTSTRUCT *)NULL)
                                          return; /*ERROR */

              }
              lastoflist = lastoflist -> next;
              lastoflist -> stuff = addthismany;
              lastoflist -> next = (MYLISTSTRUCT *)NULL;
    }


}
```

Lab2

- Create a 64 byte array inside a structure.
- Create a function to allocate that structure in memory such that it is aligned to a 128 byte boundary. Make it so you can free it without storing the initial pointer returned to you by malloc in the structure or in a global structure.
- Create a function to free that structure with just the address of the structure passed to it.
- Remember to test your logic. Write a test utility. Print the address that you first allocate and print the base address of the structure.

Hint: Store it before the data structure. Also don't forget to check your return codes and NO warnings.

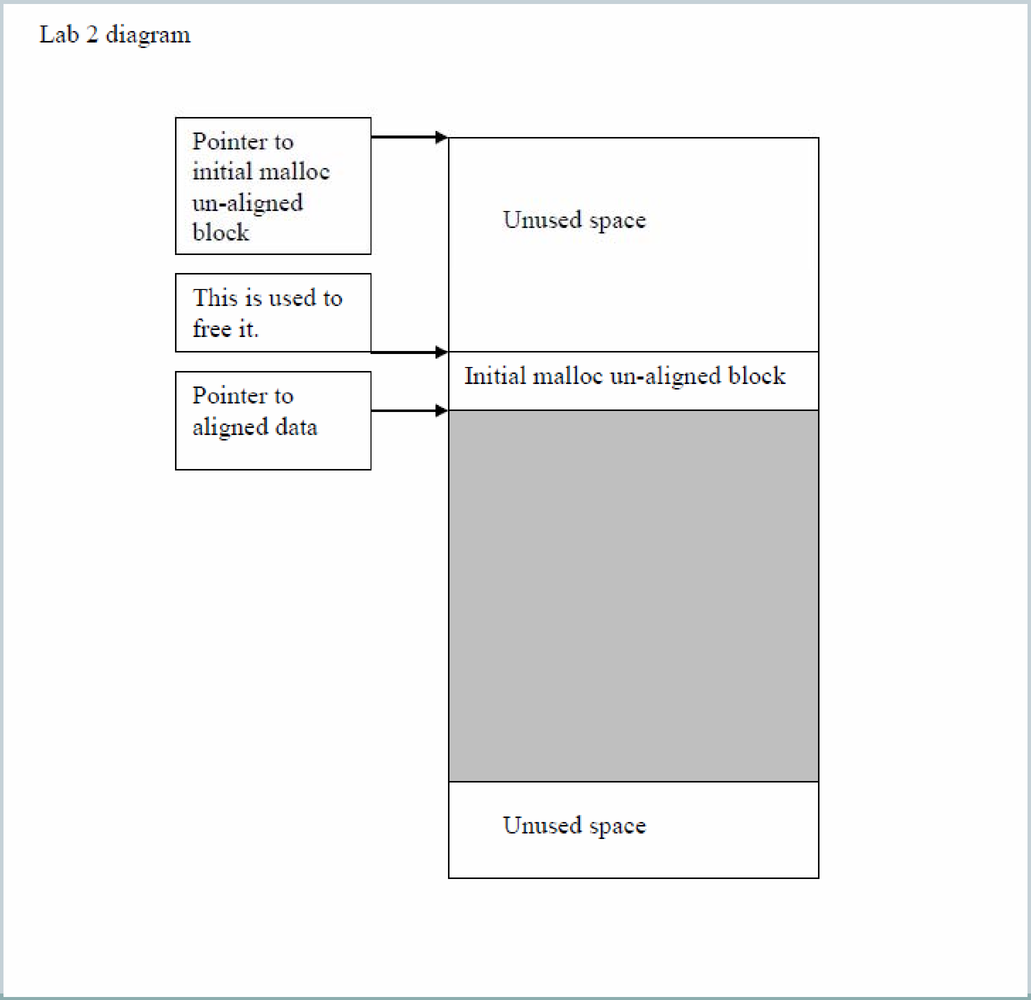
How to test lab 2

```
int main(int argc, char* argv[])
{
    char *mypointer;
    mystruct *mystuff;

    mystuff = allocfunc(&mypointer);
    printf("addr allocated = %x\n", mypointer);
    printf("addr aligned = %x\n", mystuff);

    freefunc(mystuff);

    printf("Hello World!\n");
    return 0;
}
```

Lab 2 the structure

```
typedef struct
{
   char a[64];
} mystruct;
```

Lab 2 Diagram


Lab 2 diagram
Pointer to initial malloc un-aligned block
This is used to free it.
Pointer to aligned data
Unused space
Initial malloc un-aligned block
Unused space