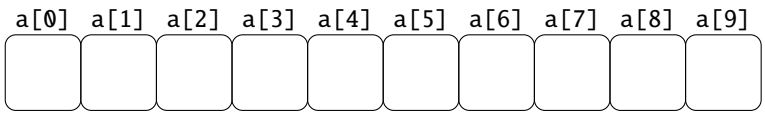
Dynamic memory management

1 Arrays in C

```
int a[10];
```

- declares a fixed size array holding ten `int` values



- `a[i]` is the `i`th element of the array
- `sizeof(a) = 10 * sizeof(int) = 40` bytes
- The array is stored in memory as a single contiguous block that is 40 bytes (10 `int`s) in size
- Note that `sizeof(a) / sizeof(a[0])=10`
  - This is a common way of checking the number of elements in an array.
  - We can't pass an array to a function, but we can pass a pointer to it. The line above will not work correctly on a pointer, so we will need to pass the length of the array too.

2 Strings

- Are represented as an array of characters

```
char a[] = "Hello worlds";
char b[13];
b = a; // Not allowed
char *c;
c = a;
```

- will set pointer `c` to same address as `a`
- assignment of an array to array is not supported in C
- unlike `struct` as we saw last lecture
- `strcpy(b,a);` first argument is the destination, ordered like assignment above
  - need to `#include<string.h>`

```
char a[] = "Hello";
strlen(a) = 5;
sizeof(a) = 6;
```

- Strings are null terminated – important when allocating space to store them

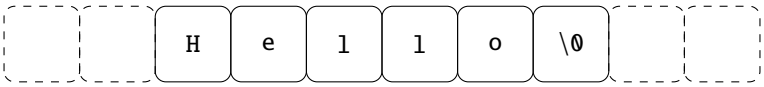
```
printf("%s %c\n", a, a[0]);
```

- Output is:

```
Hello H
```

- If you use the variable `a` on its own, it represents the memory address of the start of the string

3 Pointers, strings and arrays




```
char a[] = "Hello";
char *a = "Hello";
```

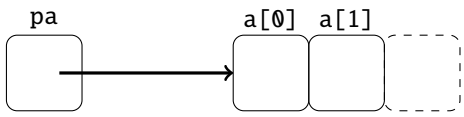
- These are equivalent declarations, and create the identical bytes in memory, as shown above.
- *Warning:* using `sizeof(a)` will give `6` in the first case (the size of the array) and `8` in the second case (the size of the pointer).
- In the second case, the string `"Hello"` is constant and cannot be modified.
- Pointers and arrays are often used interchangeably

4 Pointer arithmetic

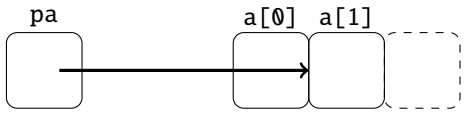
- Pointer arithmetic accounts for the base type of the items:

```
int a[10];
int *pa;

pa = &a[0];
pa = a;
```




```
pa = &a[1];
pa = (a+1);
```



- The two pairs of statements above are equivalent using array or pointer notation: `+1` translates to `+4` bytes (1 `int`)

5 Pointer oddities

- In C if I write `a[x]` this works by adding `x` to `a` to find the pointer
- Hence `a[x]` is the same as `*(a+x)`
- This seems fine if I write `a[2]`
- But what if I write `2[a]`?
- It compiles and works!
- See `array.c`

```
#include <stdio.h>

int main() {
    char a[] = "Hello Worlds";
    char *b;
```

```
    printf("The fourth character of %s is %c\n", a, a[4]);
    printf("The fourth character of %s is %c\n", a, *(a+4));
    printf("The fourth character of %s is %c\n", a, 4[a]);
    b=a;
    b=b+4;
    printf("The fourth character of %s is %c\n", a, *b);
    return 0;
}
```

All of these are valid

5.1 More oddities

- `a[-4]` ?
- Interpreted as `*(a + -4)`
- Is the following valid?

```
int *p;
int i = 5;
int j = 20;
p = &i;
printf("%d %d\n", p[0], p[1]);
```

- `p[0]` will be 5
- `p[1]` could be anything, it's just the next memory location

6 Peeking at memory

- Can look at bits of memory
- See `peek.c`
- Can find adjacent local variables and parameters
- Easy to make mistakes
- Cannot tell what data is by looking at it

7 Breaking things

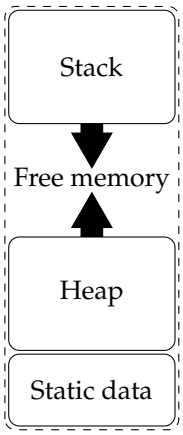
- We can use random numbers to write random values in random places
- See `break.c`
- This can upset the system
- Segmentation fault occurs: hardware tells OS a memory access is not allowed
- Sometimes it goes on for a shockingly long time
- Sometimes the last number is very strange: why?

8 Pointer safety

- Pointers can cause hard to diagnose errors in programs
- For example functions returning pointers to local variables can cause difficult to catch errors when the memory is released back to the runtime system and reused at some time later in the program
- Always set pointers to `NULL` when they are no longer required
- Always use a simple guard before using pointers e.g.: `assert(ptr != NULL);` from `<assert.h>`
- There are various tools e.g. Valgrind that can detect many such errors at runtime (at the cost of massive slowdown and greatly increased memory-usage)

9 Dynamic memory allocation

- Variables and arrays provide fixed size allocation
- What if the memory needed cannot be pre-determined?
- There is a need to be able to dynamically request variable sized blocks of memory from the runtime system
- Close integration between C and Unix (and other OS's)
- Requesting memory at run-time



10 Memory Layout

- Stack
  - Stores "temporary" data
  - Variables in a function
  - Function header
  - Small data
- Heap
  - Used for storing more long-term data
  - YOU control what is in the Heap and when it is released
  - Much more space than the Stack
- Static Data
  - Data that stays in memory for the duration of the program

11 Memory Allocation: `malloc() <stdlib.h>`

- Function prototype for `malloc()`

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

- Allocates a contiguous block of memory `size` bytes long
- The return type is `void*`, which is a generic pointer type that can be used with all types
- `malloc()` returns a `NULL` pointer if it fails to allocate the requested memory
- **Always test for `NULL` return!**
  - An aside: By default, Linux follows an optimistic memory allocation strategy: if `malloc()` returns a non-`NULL` value, this is no guarantee that the memory is really available. The operating system actually allocates the memory when you try to use it for the first time; if the system runs out of memory, a process will be killed by the Out Of Memory (OOM) killer.

11.1 malloc() example of use

```
#define SIZE 41 * sizeof(char)

char *line;
line = malloc(SIZE);
if ( line == NULL ) {
  printf( "Error in malloc() \n" );
  exit(1);
}
```

- N.B. The return value of `malloc()` is automatically cast from `void*` to `char*`
  - Pointers pointing to any object are automatically converted to `void*` pointers and vice versa as required.
  - Conversions between other sorts of pointers with different types may cause problems due to alignment issues.

12 Memory de-allocation: free() <stdlib.h>

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

- Takes a generic pointer to a block of memory and returns the memory for reuse by `malloc()`
- If you "forget" about memory you have `malloc()`ed and don't `free()` it then you have a "memory leak"
- "Memory leaks" can be very dangerous and difficult to trace, see also garbage collection in Java/Python
- Can eventually use up all memory
- `free()` has no return value, so even if you pass it a pointer not allocated by `malloc()`, it will process it!

12.1 free() example of use

```
// allocate some memory
line = malloc(SIZE);

// use line in the program

free( line ); // return memory to the O/S
line = NULL;  // set pointer to NULL
```

- Errors from continuing to use a pointer after the memory has been released can be very hard to detect
- N.B. line is implicitly cast to a `void*` pointer

13 Memory allocation: calloc() <stdlib.h>

- Function prototype for `calloc()`

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

- Allocates a contiguous block of memory of `n` elements each of `size` bytes long, initialised to all bits `0`
- Useful to ensure old data is not reused inappropriately
- The return type is `void*`, which is a generic pointer type that can be used for all types
- `calloc()` returns a `NULL` pointer if it fails to allocate the requested memory
- Always test for `NULL` return!

14 Memory allocation: realloc() <stdlib.h>

- Function prototype for `realloc()`

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

- Allows a dynamic change in size of an allocated block of memory pointed to by `ptr`
  - `ptr` must point to memory previously allocated by `malloc()`, `calloc()` or `realloc()`
- Will move and copy contents if it needs to, freeing original block
- `realloc()` returns a `NULL` pointer if it fails. **Check for this**!
- Cf. ArrayList in Java

14.1 realloc() example

- Simple program that takes integers typed in by the user and stores them in an array
- Each time the array becomes full, it is dynamically increased in size to hold more numbers
- Contains a key function `getline2()`, which reads the integers from the command line

```
int getline2(char line[], int max) {
    int nch = 0;
    int c;
    max = max - 1;/* leave room for '\0' */
    while((c = getchar()) != 'q') {
        if(c == '\n')
            break;
        if(nch < max) {
            line[nch] = c;
            nch = nch + 1;
        }
    }
    if(c == 'q' && nch == 0)
        return 'q';

    line[nch] = '\0';
    return nch;
}
```

- `getline2()`
- Uses `getchar()` to read in characters as they are typed
- Runs in a loop until a `'q'` or a newline is encountered
- Reads in the characters typed by the user one by one and stores them in the array line
- When the character `'\n'` is pressed, the function returns, via use of the `break` statement to exit a loop
- No checking performed to see if the input is an integer

```
ip = malloc(array_size * sizeof(int));
while( getline2(line, MAXLINE) != 'q' ) {
    if(nitems >= array_size ) {/* increase allocation */
        int *newp;
        array_size += INCREASE ;
        newp = realloc(ip, array_size * sizeof(int));
        printf("<< Expanding by %d to size %d >>\n",
                            INCREASE, array_size );
        if(newp == NULL) {
```

```
            printf("out of memory\n");
            exit(1);
        }
        ip = newp;
    }
    ip[nitems++] = atoi(line);
}
```

- `main()`
- Uses `getline2()` to read in a line of text
- Creates an array to store current line of text, `line`
- Creates a second array to store the integers entered: `ip`
- As soon as `ip` is full, `realloc()` is called to resize the array

15 atoi() <stdlib.h>

```
int atoi(const char *s);
```

- Converts a string pointed to by s to an integer
- Also see `atof()`, `atol()` and `atoll()` (since C99) equivalents
- To convert from an integer to a string use :

  ```
  int sprintf( char *s, char *format, <value list> );
  ```

- Where the value list is the variables used in the format string