EECS 2031

Software Tools

Module 8 – Dynamic Memory Allocation

YORK UNIVERSITÉ UNIVERSITY

Dynamic Memory Allocation

- It is often not known at compile time how much memory will be needed to store the program's data

```
int x;
scanf("%d", &x);
int a[x]; /* not allowed in C */
```

- How to allocate memory during run time?

`malloc()`

```
void *malloc( int n );
```

- Allocates memory at run time
- Returns a **`void`** pointer to at least **`n`** bytes available
- Returns **`NULL`** if the memory was not allocated
- The allocated memory is not initialized

`void` Pointer

- A **`void`** pointer is a variable whose value is a memory address but does not point to a specific data type
- To be useful, it must be converted to a typed pointer, typically done by assignment

```
int *a;
a = malloc ( 5 * sizeof(int) );
```

- **`a`** points to a chunk of memory that can hold 5 integers

`calloc()`

```
void *calloc( int n, int s );
```

- Allocates an array of **`n`** elements where each element has size **`s`**
- **`calloc( )`** initializes all the allocated memory to 0

`realloc()`

- What if we want our array to grow?

```
void *realloc(void *ptr, int n);
```

- Resizes a previously allocated block of memory.
- **`ptr`** must have been returned from a previous **`calloc`**, **`malloc`**, or **`realloc`**
- The new array may be moved if it cannot be extended in its current location

`free()`

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

- Releases the memory we previously allocated
- **`ptr`** must have been returned from a previous **`calloc`**, **`malloc`**, or **`realloc`**
- C does not do automatic **garbage collection**
- See **`alloc.c, readname1.c, readname2.c`**

Be extra careful with pointers!

Common errors:

- Overruns and underruns
  - Occurs when you reference memory beyond what you allocated
- Uninitialized pointers
- De-referencing null pointers
- Memory leaks
- Inappropriate use of freed memory

Pointer problems

- See `pow1.c`, `pow2.c`
- Two very small examples of misusing memory
- See `null.c`
- In a real system, one should always test that `malloc` has returned successfully

Memory Leaks

```
int *x;
x = malloc( 20 );
x = malloc( 30 );
```

- The first memory block is lost for ever.
- May cause problems if repeated (available memory will be exhausted)

Using Freed Memory

```
char *x;
x = malloc( 50 );
free( x );
x[0] = ‘A’;
```

May work on some systems

Arrays of Pointers

```
char *s[]={“one”,”two”,”three”};
```

- `s` is an array of pointers to `char`
- Each element of `s` (`s[0], s[1], s[2]`) is a pointer to `char`
- What is the difference between `s` and `t`?

```
char t[][6]={“one”,”two”,”three”};
```

Arrays of Pointers

- In `t`, all characters are stored in the same memory location
- In `s`, all that is stored together is the pointers. These pointers could be pointing to different parts of the memory
- `s` is an array of strings that can be easily rearranged (sorted) by changing the pointer values

Pointers and Structures

- Dynamic memory allocation works in the same way

```
Point *points;
points =
    malloc (20 * sizeof *points);
```

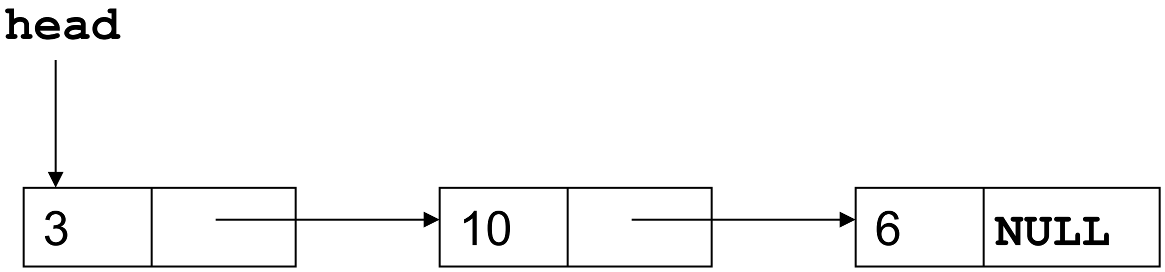
Self-referential structures

```
struct list  {
    int data;
    struct list *next;
};
```

Each **`struct list`** contains a piece of data and a link to another **`struct list`**

Linked List

- Pointer **`head`** points to the first element
- Last element pointer is **`NULL`**
- See **`linkedlist.c`**



Pointers to Pointers

- Pointers can point to any valid type including other pointers

```
int **j;

int *i;

int k = 10;

i = &k;

j = &i;
```

Pointers to Pointers

- What are double pointers useful for?
- Returning a pointer from a function
- Declaring fully dynamic two dimensional arrays
- See **`doublepointer.c`**

Command-Line Arguments

- Up to now, the signature of the main function has been `int main()`
- Usually it is defined as

```
int main(int argc, char *argv[])
```

- `argc` is the number of arguments
- `argv` is an array of pointers to `char` containing the arguments

Command-Line Arguments

- `argv[0]` is a pointer to a string with the program name. So, `argc` is at least 1.
- `argv[argc]` is a NULL pointer.
- See `argv.c`
- See `echo.c` for a possible implementation of the Unix `echo` command