

CSCI-UA.0201

Computer Systems Organization

C Programming – Dynamic Memory Allocation

Thomas Wies

wies@cs.nyu.edu

<https://cs.nyu.edu/wies>

String Conversion Functions

- Conversion functions
 - In `<stdlib.h>` (general utilities library)
 - Convert strings of digits to integer and floating-point values

Prototype	Description
<code>double atof(const char *nPtr)</code>	Converts the string <code>nPtr</code> to double .
<code>int atoi(const char *nPtr)</code>	Converts the string <code>nPtr</code> to int .
<code>long atol(const char *nPtr)</code>	Converts the string <code>nPtr</code> to long int .
<code>double strtod(const char *nPtr, char **endPtr)</code>	Converts the string <code>nPtr</code> to double .
<code>long strtol(const char *nPtr, char **endPtr, int base)</code>	Converts the string <code>nPtr</code> to long .
<code>unsigned long strtoul(const char *nPtr, char **endPtr, int base)</code>	Converts the string <code>nPtr</code> to unsigned long .

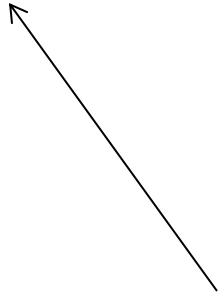
String Manipulation Functions

- In **<string.h>**
- String handling library has functions to
 - Manipulate string data
 - Search strings
 - Determine string length

Function prototype	Function description
<code>char *strcpy(char *s1, const char *s2)</code>	Copies string s2 into array s1 . The value of s1 is returned.
<code>char *strncpy(char *s1, const char *s2, size_t n)</code>	Copies at most n characters of string s2 into array s1 . The value of s1 is returned.
<code>char *strcat(char *s1, const char *s2)</code>	Appends string s2 to array s1 . The first character of s2 overwrites the terminating null character of s1 . The value of s1 is returned.
<code>char *strncat(char *s1, const char *s2, size_t n)</code>	Appends at most n characters of string s2 to array s1 . The first character of s2 overwrites the terminating null character of s1 . The value of s1 is returned.

String Manipulation Functions

```
int strcmp ( const char * str1,  
            const char * str2 )
```



return value

<0

0

>0

indicates

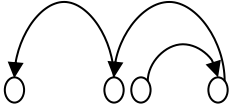
the first character that does not match has a lower value in *ptr1* than in *ptr2*

the contents of both strings are equal

the first character that does not match has a greater value in *ptr1* than in *ptr2*

How to Parse C Types

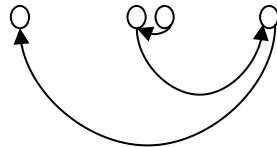
C type names are parsed by **starting at the name** and working outwards according to the rules of precedence:



`int *x[10];`

x is
an array of
pointers to
int

`int (*x)[10];`



x is
a pointer to
an array of
int

Using typedef

At this point we have seen a few basic types, arrays, pointer types, and structures. So far we've glossed over how types are named.

```
int x;           /* int;           */ typedef int T;
int *y;         /* pointer to int; */ typedef int *U;
int z[10];      /* array of ints;  */ typedef int v[10];
int *k[10];     /* array of pointers to int; */ typedef int *w[10];
int (*m)[10];  /* pointer to array of ints; */ typedef int (*N)[10];
```

typedef defines a new type



Now:

T x; is the same as **int x;**
U y; is the same as **int * y;**
and so on ...

What if you want to allocate an array of N elements, and you don't know N beforehand?

What if you want to allocate an array that should persist after the current function returns?

Dynamic Memory Allocation

So far all of our examples have allocated variables **statically** by defining them in our program. This allocates them in the stack.

But, what if we want to allocate variables based on user input or other dynamic inputs, at run-time? This requires **dynamic** allocation.

sizeof() reports the size of a type in bytes

```
int * alloc_ints(size_t requested_count)
{
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }

    /* now big_array[0] .. big_array[requested_count-1] are
     * valid and zeroed. */
    return big_array;
}
```

calloc(N, K) allocates memory for N elements of size k

Returns NULL if can't alloc

It's OK to return this pointer. It will remain valid until it is freed with free()

Dynamic Memory Allocation

- `void *malloc (size_t size);`
- `void* calloc (size_t num, size_t size);`
- `void free (void* ptr);`
- Unary operator **sizeof** is used to determine the size in bytes of any data type. Examples:
 - `sizeof(double)`
 - `sizeof(int)`

Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and freed when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory is called a “memory leak”.

It is easy to accidentally keep a pointer to dynamic memory that has been freed. Whenever you free memory you must be certain that you will not try to use it again. It is safest to erase any pointers to freed dynamic memory.

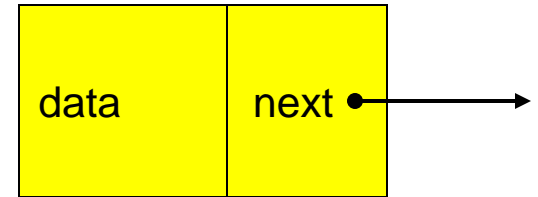
Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic allocation.

```
int a[3] = {1, 2, 4};
```

Back to struct

- Assume we have a structure called node:

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



- Inserting a node in a linked list of nodes (with head pointer called pHead):

```
struct node *pNew;  
pNew = (struct node *) malloc(sizeof(struct node));  
pNew -> data = item;  
if (pHead == NULL){  
    //add before first node or to an empty list  
    pNew -> next = pHead;  
    pHead = pNew;  
}  
else {  
    //add in the middle or at the end  
    pNew -> next = pHead -> next; // pHead points to previous node  
    pPre -> next = pHead;  
}
```

Back to struct

