Dynamic Memory Allocation (malloc and calloc).

King Chapter 17: describes the mechanism for obtaining a pointer to a block of new memory. This is accomplished with the routines, **malloc** and **calloc**, found in <stdlib.h>

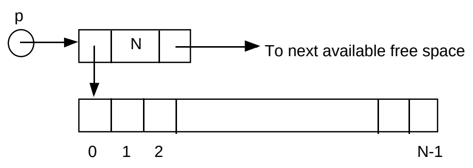
malloc() returns N bytes of space for the user, and
calloc() gets space for an array of objects each of size N.

Thus malloc is best viewed as returning a pointer to an array of chars, while calloc will return a pointer to an array of the same type as the objects in each element.

```
int N = 100;
char* p = (char*) malloc (N);
```

will return a pointer to a space in the heap were a contiguous block of 100 bytes of memory exists. NONE of those bytes will be initialized. If space cannot be found, then p will be NULL.

However, keep in mind that to manage this memory the system actually needs some overhead space.



Remember, if you want to use malloc to obtain memory for a string, you must leave space for the **'\0'** terminator.

To give the space back to the system simply write

```
   free(p);
```

To obtain space for N integers write

```
int* q = (int*) malloc ( N*(sizeof (int)) );
```

Note the necessary and different coercion (casting). Because malloc is a void* function it can be coerced to point to any object. In C++ no default coercion occurs.

calloc is similar in many ways, but it **initializes the memory** obtained **to zero**. More general and better.

To obtain N integers, each initialized to zero, write:

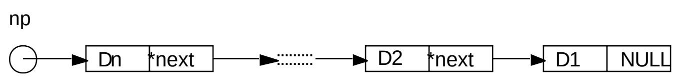
```
int* q = (int*) calloc (N, sizeof(int));
```

Similarly

```
   free (q);
```

gives the space back to the system.

calloc is especially useful for acquiring space for a data structure element.

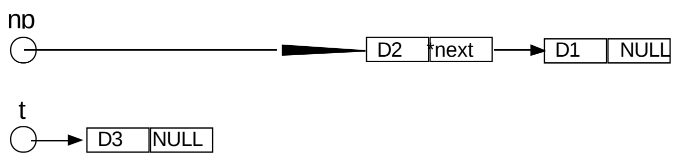


// prototypes and declarations in header file

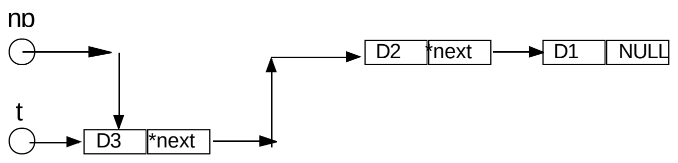
```
struct node {
   int data;
   struct node* next;
};                              // prototype node

typedef struct node* NodePtr;   // defining NodePtr
#define NodePtr struct node*    // same as above

NodePtr makenode ( int );       // prototype funct
```



t->next = np;
np = t;



```
   // functions and main program
NodePtr makenode ( int item )
{
   NodePtr ptr;             // definition of ptr
   ptr = (NodePtr) malloc ( sizeof (struct node) );

   if (ptr != NULL) {       // check valid pointer
      ptr->data = item;
      ptr->next = NULL;
   } else exit (1);
   return ptr;
}

int main (void) {
   NodePtr np = NULL;          // np start of list
   NodePtr t;                  // t is a temp

   for (i = 1; i < 5; i++) {
      t = makenode (i);        // assumes t != NULL
      t->next = np;
      np = t;
   }
   for ( t = np; t != NULL; t = t->next)
      printf ("%d\n", t->data);
}
```

What we have actually built here is a stack.
A Last In First Out (LIFO) Queue

NOTE: We have simplified the programming by using a typedef in the header file to define `NodePtr`.

Operations on strings

```
char* s = "My sample string";
```

we can also use the typedef statement to form a user type

```
typedef char* String;      // #define String char*
String s = "My sample string";
```

We can now find the length of this string with:

```
int len = strlen (s);           // will yield 16
```

but if we want to copy this string into an array then we would need a declaration like:

```
char letters [1+strlen(s)];      // Why 1+?
```

and then use the copy procedure **strcpy()**

```
strcpy (&letters[0], s);
```

This however is not common usage.
Note the prototype for the strcpy function

```
char* strcpy ( char* StrNew, const char* StrOrig);
```

Questions:
why is this a char* function and not a void function?

why is the second parameter of type const char* ?

const attached to a parameter means that the parameter will not change. In this case it means that the parameter StrOrig cannot be an L-value within the strcpy function.

Thus it can provide valuable protection and assurance that the copy will not be in the opposite direction!

So what is happening here?
Let us consider an example

```
char* s = "My sample string";
String tmp;            // char* tmp;
strcpy (tmp, s);       // tmp has a copy of s
```

First this is not the simple case in which tmp simply points to the original string s, as in:

```
tmp = s;
```

why on earth would we need strcpy to do this?

strcpy() actually does something like:

```
tmp = (String) malloc (1+strlen(s));   // space

for ( i = 0; *(s+i) != '\0'; i++ )
   *(tmp+i) = *(s+i);
                              // String terminator?
return (tmp);
```

Of course we don't really need this **return(tmp),** but it does help us handle the error condition where malloc fails.

In the above we have assumed that malloc always succeeds. strcpy() must handle failure by returning the NULL pointer.

```
tmp = (char*) malloc (1+strlen(s));

if ( tmp == NULL) return NULL;

for ( i = 0; *(s+i) != '\0'; i++ ) {
   *(tmp+i) = *(s+i);
}

*(tmp+i) = '\0';          // String terminator!

return (tmp);
```

Outside in the calling program one might well actually use strcpy as follows:

```
if (strcpy (tmp, s) != NULL) {.........}
```

Another useful string function is strcmp() which is used to determine if two strings are identical.

The prototype is:

```
int strcmp (const char* s1, const char* s2);
```

It returns values of <0, 0 or >0 depending whether s1 is less than, equal or greater than s2

What does "less than" mean in this context?

A comparison is made on a character by character basis in each string until the two strings are different. Then the ordering of the characters in the **ASCII character** set (see Appendix E in King) is used to determine the relative order of the two strings. Typical usage might be:

```
if ( strcmp (s1, s2) == 0 ) {..//strings identical..
```

The three functions **strlen()**, **strcpy()** and **strcmp()** are enough for our purpose. When you need more you will be skilled enough to read the prototypes in **<string.h>**

Read King Chapter 13 and K&R Chapter 7 for details

Review of fgets() and command line args.

To aid in input/output of strings we have the **fgets()**

char* fgets (char* line, int maximum, FILE* fp);

Let us assume that **fp** correctly points to an open file. However if we are now at the end of that file, then fgets will return NULL, indicating that an EOF was read. Otherwise **fgets** reads the next line of input (**including the newline**) from the file pointed to by **fp** into the character array **line**, but no more than **maximum-1** characters will be read! The resulting string (in array line) is terminated with **'\0'**.

Avoid the functions gets() and puts() on stdin and stdout

Command line arguments

use of **int main (int argc, char* argv[ ]);**

Imagine we want to write a program **reverseline** that reverses every line in a file, but the file name is specified in the command line. Thus instead of invoking as

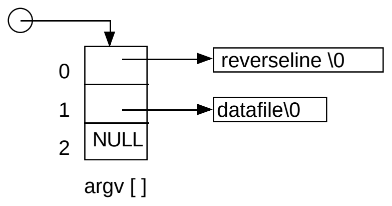
reverseline <datafile

and taking input from stdin, we want to type:

reverseline datafile

specifying the name of the input file as a command line parameter

```
FILE* fopen (const char* file, const char* mode);
void reverse (char* buffer);
```



```
int main (int argc, char* argv[ ]) {
   char line[80];

   if (argc != 2) {
      printf ("Wrong usage for %s\n", argv[0]);
      return (1);
   }

   fp = fopen (argv[1], "r");    // datafile

   if (fp != NULL) {
      while ( fgets ( &line[0], 80, fp) != NULL)
         reverse (&line[0]);
         printf ("%s", line);
      }     // found EOF
   } else return (2);  // file open failure

   return 0;
}
```

Of course we still have to write the “reverse” function, but is left as a (non-trivial) exercise.