Dynamic Memory Allocation

- Dynamic memory allocation
  - How to allocate memory for variables (esp. arrays/strings) during run time
  - malloc(), calloc(), realloc(), and free()

Why dynamic memory allocation?

Usually, so far, the arrays and strings we're using have fixed length (i.e., length is known at compile time)

- Example:

```
char myStr[11];        // allocates memory for 10 chars
printf("Enter a string: ");
fgets(myStr, 11, stdin);
```

What if the user wants to enter a string more than 10 chars long or if the length is known only at run time?

malloc()

- malloc() is used to request additional memory from the operating system during program execution

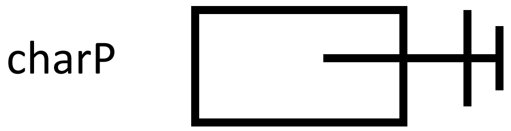
  Syntax: **`malloc(numBytes)`**

- Input is the number of consecutive bytes to be allocated
- Return value is a **pointer** to the beginning of the block of memory allocated or NULL if malloc fails

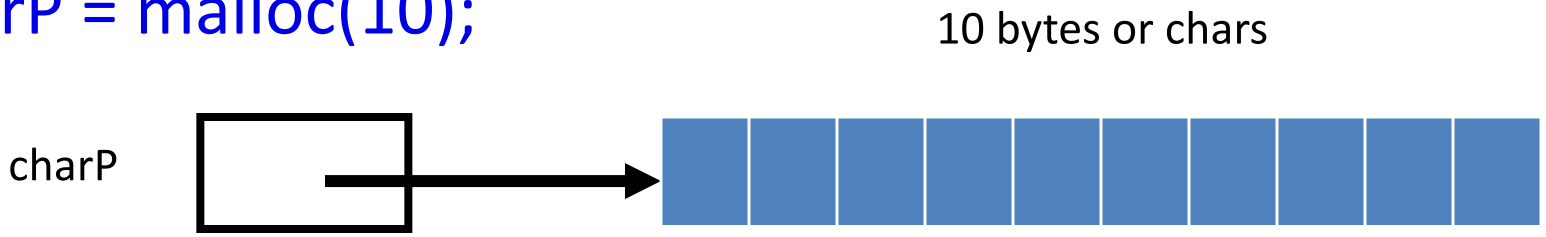
- To use malloc(), you must **#include <stdlib.h>**

malloc()

```
char *charP;        /* declare a pointer to char */
```



```
charP = malloc(10);
```



charP contains the address of the beginning of that block.

free()

- The function free() returns memory to the memory pool. It “frees” up memory

  Syntax: free(ptr)

  - where ptr “points to” memory previously allocated by malloc() function

- To use free(), you must #include <stdlib.h>

Example

This program allows the user to specify the length of a string, allocates memory for the string, and then applies string operations

```
#include <stdlib.h>          /* you need this library for malloc(), free(), exit() */
#include <stdio.h>
#include <string.h>          /* you need this library for strchr(), strlen() */

int main ()
{
  char *charP, *q;
  int maxlen;

  printf("Enter maximum string length: ");
  scanf("%d", &maxlen);
  getchar();                              /* reads the newline character */

  printf("Enter the string: ");
  fgets(charP, maxlen, stdin);

  if ((q = strchr(charP, '\n')) != NULL)
       *q = '\0';
  printf("You've entered a string %s of length %d\n", charP, strlen(charP));
  free(charP);
}
```

What it does:

```
if ((q = strchr(charP, '\n')) != NULL)
      *q = '\0';
```

The function fgets returns the entire line input by the user including the newline at the end (when the user hit return). The function strchr(charP, ‘\n’) will return a pointer to that character (newline) if it exists or NULL otherwise. If the result in the variable q is not NULL, the if statement executes the line of code to replace the newline with a null termination for the string.

strchr

```
if ((q = strchr(charP, '\n')) != NULL)
        *q = '\0';
```

This code finds the first occurance of the character ‘\n’ which is the newline character that fgets will obtain. If found (value in q is not NULL), it sets that character to the string null termination.

Memory leak

If malloc'ed memory is not free'ed, then the OS will "leak memory"

- This means that memory is allocated to the program but not returned to the OS when it is finished using it
- The program therefore grows larger over time.

Memory leak example

```
int main ()
{
  char *charP, r[80];
  int length;

  while (1)
  {
        printf("Enter the maximum string length: ");
        fgets(r, 80, stdin);
        sscanf(r, "%d", &length);

        if ((charP = malloc(length)) == NULL) {
                printf("Out of memory. Quitting\n");
                exit(1);
        }
        printf("Enter the string: ");
        fgets(charP, length, stdin);
        /* free(charP); */
   }
}
```

Memory leak example

Each iteration of the loop allocates memory for a string. But, that memory is never freed. Hence, we have a memory leak.

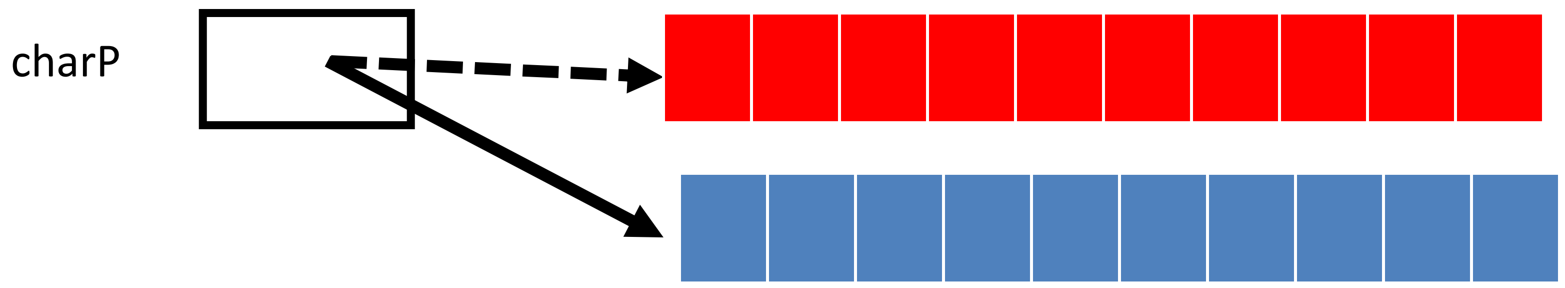
```
int main ()
{
  char *charP, r[80];
  int length;

  while (1)
  {
      printf("Enter the maximum string length: ");
      fgets(r, 80, stdin);
      sscanf(r, "%d", &length);

      if ((charP = malloc(length)) == NULL) {
          printf("Out of memory. Quitting\n");
          exit(1);
      }
      printf("Enter the string: ");
      fgets(charP, length, stdin);
      /* free(charP); */
  }
}
```

malloc without freeing

```
while (1)
{
      charP = malloc(length+1);
  …
}
```



If you don't free the allocated memory, previous block is still "ours" according to the OS, but we can no longer find it (no pointer to it). That block is an orphan!

It's like you bought a house, but then lost the address. You still own it and pay taxes on it, but you can't use it because you can't find it.

Always free what you malloc

- You are responsible for your memory, you must allocate it and free it.

- Unlike other languages, it is all up to you!

- If you don’t free it, your program grows larger and eventually runs out of memory!

malloc allocates bytes

- If you want a character array that stores 10 characters (including ‘\0’):

```
char *p = malloc(10);
```

- If you want to allocate storage for 10 ints (or doubles or floats), you can’t do this:

```
int *p = malloc(10);      /* WRONG! Why? */
```

allocate int and double array

```
int *intP;
double *doubleP;

// Allocate space for 10 integers
intP = malloc(10 * sizeof(int));

// Allocate space for 10 doubles
doubleP = malloc(10 * sizeof(double));
```

allocate int and double array

```
int *intP;
double *doubleP;

// Allocate space for 10 integers
intP = malloc(10 * sizeof(int));

// Allocate space for 10 doubles
doubleP = malloc(10 * sizeof(double));
```

Allocates 40 bytes
sizeof(int) = 4

Allocates 80 bytes
sizeof(double) = 8

realloc

realloc takes a pointer to allocated memory and reallocates the memory to a larger size

- if it can make the old block bigger, great
- if not, it will get another, larger block, copy the old contents to the new contents, free the old contents and return a pointer to the new

```
intP = malloc(sizeof(int));
intP = realloc(intP, 2*sizeof(intP));
```

intP may be different after a realloc!

An example program

Step 1: Prompt the user to enter the number of random numbers to generate

```
int main ()
{
 double *dblPtr;
 int howMany, randNum;

 printf("How many random numbers to generate:");
 howMany=ProcessInput(stdin);

 dblPtr = malloc(howMany * sizeof(double));
 if (dblPtr == NULL)
 {
  printf("memory allocation error, exiting\n");
  exit(1);
 }

 for (int i=0;i<howMany;i++)
 {
  randNum = random();
  dblPtr[i]=(randNum%10000)/1000.0;
 }

PrintArray(dblPtr,howMany);
```

```
int main ()
{
  double *dblPtr;
  int howMany, randNum;

  printf("How many random numbers to generate:");
  howMany=ProcessInput(stdin);

  dblPtr = malloc(howMany * sizeof(double));
  if (dblPtr == NULL)
  {
    printf("memory allocation error, exiting\n");
    exit(1);
  }

  for (int i=0;i<howMany;i++)
  {
    randNum = random();
    dblPtr[i]=(randNum%10000)/1000.0;
  }

PrintArray(dblPtr,howMany);
```

Step 2: create a dynamic array to store the random numbers

In this example we have tested to be sure malloc succeeded. If not, we are out of memory.

Step 3: generate the random numbers and print them

```
int main ()
{
  double *dblPtr;
  int howMany, randNum;

  printf("How many random numbers to generate:");
  howMany=ProcessInput(stdin);

  dblPtr = malloc(howMany * sizeof(double));
  if (dblPtr == NULL)
  {
    printf("memory allocation error, exiting\n");
    exit(1);
  }

  for (int i=0;i<howMany;i++)
  {
    randNum = random();
    dblPtr[i]=(randNum%10000)/1000.0;
  }

PrintArray(dblPtr,howMany);
```

```
    dblPtr = realloc(dblPtr, 2*howMany);
```

Step 4: double the size of the array using realloc

```
    for (int i=howMany; i<howMany*2; i++)
    {
        randNum = random();
        dblPtr[i]=(randNum%10000)/1000.0;
    }

    howMany *= 2;

    PrintArray(dblPtr, howMany);
    free(dblPtr);
}
```

Step 5: generate more random numbers and print them

```
int ProcessInput(FILE *f)
{
    int val;
    char in[100];
    fgets(in,100,f);
    sscan(in, “%d”, &val);
    return val;
}
```

Safe data entry, just like last week

Print Array

```
void PrintArray(double *ptr, int cnt)
{
  printf("Printing Array Values\n");
  for(double *p=ptr;  p<ptr+cnt;  p++)
    printf("Val is:%lf\n",*p);
}
```