Dynamic Memory Allocation

IC-100

December, 2022

Simplified View of Memory

- In programming, “type” helps us decide whether 1004001 is an integer or a pointer to block containing ‘E’ (or something else)

```
#include<stdio.h>
int main() {
      char x[5] = {‘A', ‘E', ‘I', ‘O', ‘U'};
      int y = 1024;
      char *p = x+1;
      ...
}
```

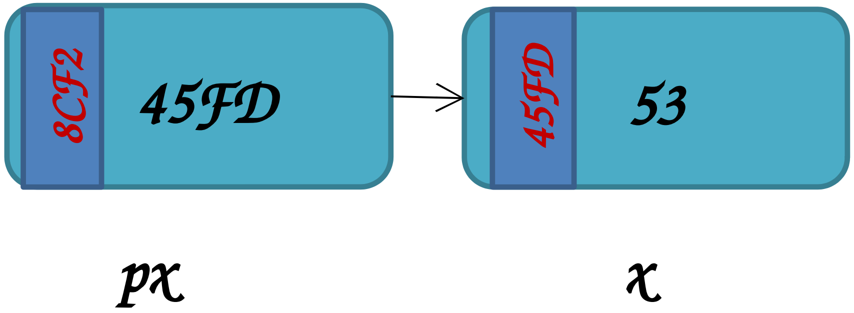
Declaration of a pointer to char box

```
#include<stdio.h>
int main() {
      char x[5] = {‘A', ‘E', ‘I', ‘O', ‘U'};
      int y = 1024;
      int p = 1004001;
      ...
}
```

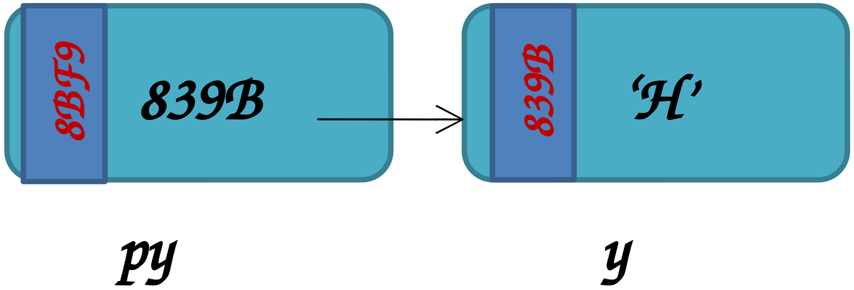
1004000	‘A’	x
1004001	‘E’	
1004002	‘I’	
1004003	‘O’	
1004004	‘U’	
1004005		
1004006		
1004007		
1004008	1024	y
1004009		
1004010		
1004011		
1004012	1004001	p
1004013		
1004014		
1004015		

Pointers: Visual Representation

- Typically represented by box and arrow diagram



- x is an int variable that contains the value 53.
- Address of x is 45FD.
- px is a pointer to int that contains address of x.

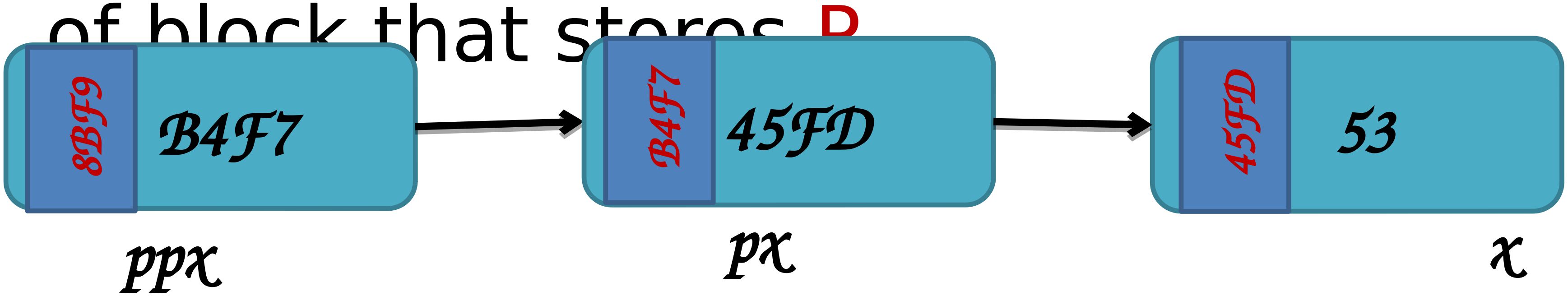


- y is an char variable that contains the character 'H'.
- Address of y is 839B.
- py is a pointer to char that contains address of y.

We are showing addresses for explanation only.
Ideally, the program should not depend on actual addresses.

Pointer to a Pointer

- If we have a pointer P to some memory cell, P is also stored somewhere in the memory.
- So, we can also talk about address of block that stores P.



```
int x = 53;
int *px = &x;
int **ppx = &px;
```

Size of Datatypes

- The smallest unit of data in your computer's memory is one bit. A bit is either 0 or 1.
- 8 bits make up a byte.
- 2^{10} bytes is 1 kilobyte (KB). 2^{10} KB is 1 megabyte (MB). 2^{10} MB is 1 gigabyte (GB).

  2^{10} GB is 1 terabyte (TB)
- Every data type occupies a fixed amount of space in your computer's

Size of Datatypes

- There is an operator in C that takes as argument the name of a data type and returns the number of bytes the data type takes
    - the sizeof operator.
- For example, sizeof(int) return the number of bytes a variable of type int uses up in your computer’s memory.

sizeof Examples

```c
printf("int: %d\n", sizeof(int));
printf("float: %d\n", sizeof(float));
printf("long int: %d\n", sizeof(long int));
printf("double: %d\n", sizeof(double));
printf("char: %d\n", sizeof(char));
printf("int ptr: %d\n", sizeof(int *));
printf("double ptr: %d\n", sizeof(double*));
printf("char ptr: %d\n", sizeof(char *));
```

```
int: 4
float: 4
long int: 8
double: 8
char: 1
int ptr: 8
double ptr: 8
char ptr: 8
```

- The values can vary from computer to computer.
- Note that all pointer types occupy the same number of bytes (8 bytes in this case).
  - Depends only on total # of memory blocks (RAM/Virtual Memory) and not on data type

Static Memory Allocation

- When we declare an array, size has to be specified before hand.
- During compilation, the C compiler knows how much space to allocate to the program
  - Space for each variable.
  - Space for an array depending on the size.
- This memory is allocated in a part of the memory known as the stack.
- Need to assume worst case scenario
  - May result in wastage of Memory

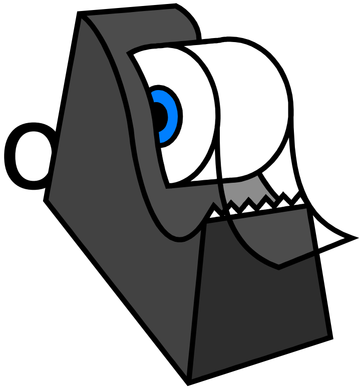
Automatic Memory Allocation

- The compiler automatically creates boxes to hold return values and actual parameters of function calls
  - Memory only allocated for the duration of the function call
  - Destroyed after function call
- Also allocated on stack
- Can’t use for persistent operations
  - Except with ‘static’ type, which is limited to single variables

Middle Path: Dynamic Memory Allocation

- There is a way of allocating memory to a program during runtime.
- This is known as dynamic memory allocation.
- Dynamic allocation is done in a part of the memory called the heap.
- You can control the memory allocated depending on the actual input(s)
  - Less wastage

Memory Allocation: mallo

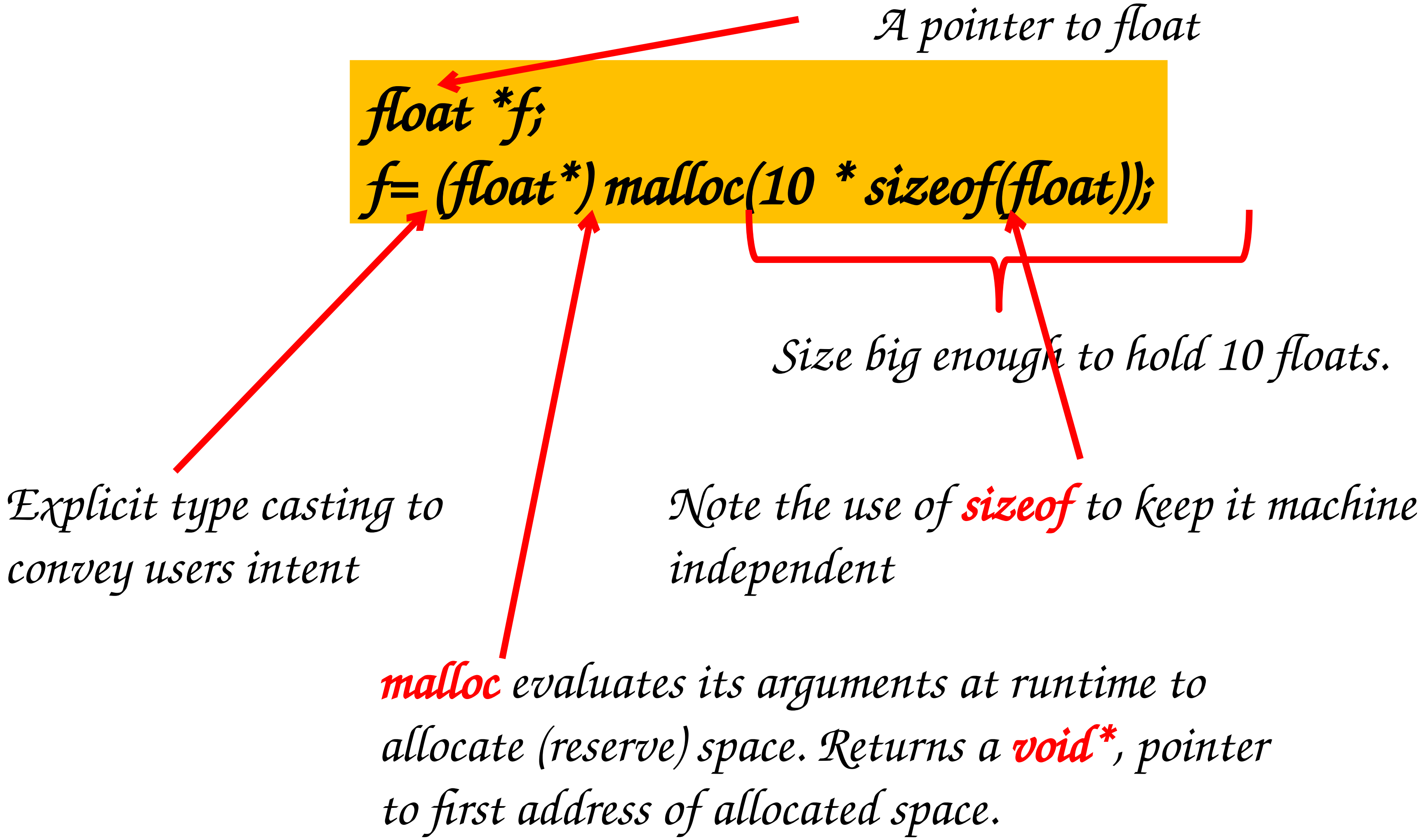


- The malloc function is declared in stdlib.h
- Takes as argument an integer (say **n,** typically > 0),
- Allocates **n** consecutive bytes of memory space
- returns the address of the first cell of this memory space
- The return type is void*

void* is NOT pointer to nothing!

- malloc knows *nothing* about the use of the memory blocks it has allocated
- void* is used to convey this message
  - Does not mean pointer to nothing, but means pointer to *something* about which *nothing is known*
- The blocks allocated by malloc can be used to store "anything" provided we allocate enough of them

malloc: Example



A pointer to float

```
float *f;
f= (float*) malloc(10 * sizeof(float));
```

Size big enough to hold 10 floats.

Explicit type casting to convey users intent

Note the use of **sizeof** to keep it machine independent

malloc evaluates its arguments at runtime to allocate (reserve) space. Returns a **void***, pointer to first address of allocated space.

malloc: Example

Key Point: The size argument can be a variable or non-constant expression!

After memory is allocated, pointer variable behaves as if it is an **array**!

```
float *f; int n;
scanf(“%d”, &n);
f= (float*) malloc(n * sizeof(float));

f[0] = 0.52;
scanf(“%f”, &f[3]); //Overflow if n<=3
printf(“%f”, *f + f[0]);
```

This is because, in C, f[i] simply means *(f+i).

Exercise

- Write a program that reads two integers, n and m, and stores powers of n from 0 up to m (n^0,

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *pow, i, n, m;
    scanf("%d %d", &n, &m); // m>= 0
    pow = (int *) malloc ((m+1) * sizeof(int));
    pow[0] = 1;
    for (i=1; i<=m; i++)
        pow[i] = pow[i-1]*n;
    for (i=0; i<=m; i++)
        printf("%d\n",pow[i]);
    return 0;
}
```

Note that instead of writing **pow[i]**, we can also write

***(pow + i)**

NULL

- A special pointer value to denote "points-to-nothing"
- A malloc call can return NULL if it is not possible to satisfy memory request
  - negative or ZERO size argument
  - TOO BIG size argument

Pointers and Initialization

- Uninitialized pointer has GARBAGE value, NOT NULL
- Memory returned by malloc is not initialized.
- Brothers of malloc
  - calloc(n, size): allocates memory for n-element array of size bytes each. Memory is initialized to 0.
  - realloc(ptr, size): changes the size of the memory block pointed to by ptr to size bytes.

Both malloc, calloc return a **logically contiguous** block of memory.

Calloc also **clears-memory** with zeros.

malloc()

To allocate memory use

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

- Takes number of bytes to allocate as argument.
- Use sizeof to determine the size of a type.
- Returns pointer of type void *. A void pointer may be assigned to any pointer.
- If no memory available, returns NULL.

e.g.

```
char *line;
int linelength = 100;
line = (char*)malloc(linelength);
```

malloc() example

To allocate space for 100 integers:

```
int *ip;

if ((ip = (int*)malloc(100 * sizeof(int))) == NULL){
  printf("out of memory\n");
  exit();
  }
```

- Note we cast the return value to int*.
- Note we also check if the function returns NULL.

free()

To release allocated memory use

```
free()
```

- Deallocates memory allocated by malloc().
- Takes a pointer as an argument.

```
e.g.
free(newPtr);
```

Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed will be automatically released.

calloc()

Similar to malloc(), the main difference is that the values stored in the allocated memory space are zero by default. With malloc(), the allocated memory could have any value.

calloc() requires two arguments - the number of variables you'd like to allocate memory for and the size of each variable.

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

Like malloc(), calloc() will return a void pointer if the memory allocation was successful, else it'll return a NULL pointer.

calloc() example

```
/* Using calloc() to initialize 100 floats to 0.0 */
#include <stdlib.h>
#include <stdio.h>
#define BUFFER_SIZE 100

int main(){
    float * buffer;
    int i;

    if ((buffer = (float*)calloc(BUFFER_SIZE,sizeof(float))) ==
     NULL){
         printf("out of memory\n");
         exit(1);
     }

    for (i=0; i < BUFFER_SIZE; i++)
         printf(“buffer[%d] = %f\n”, i, buffer[i]);

    return 0;
}
```

realloc()

If you find you did not allocate enough space use realloc().

You give realloc() a pointer (such as you received from an initial call to malloc()) and a new size, and realloc does what it can to give you a block of memory big enough to hold the new size.

```
int *ip;

ip = (int*)malloc(100 * sizeof(int));
...
/* need twice as much space */
ip = (int*)realloc(ip, 200 * sizeof(int));
```

Next Class

- Pointers and Arrays