ESC101: Introduction to Computing

Pointers

Pointers

- Declaration of pointers
  - int *p; // pointer to an integer
  - int *p1, *p2; // two integer pointers
- Assigning pointers
  - int a;
  - int *p;
  - p = &a; //p points to address of a
- Obtaining value
  - printf("%d",*p); //dereferencing
- Functions
- int fun(int *pa, int *pb); //function with two pointers
- int main(){ int a, int b; fun(&a, &b); return 0;}

The program to swap integers

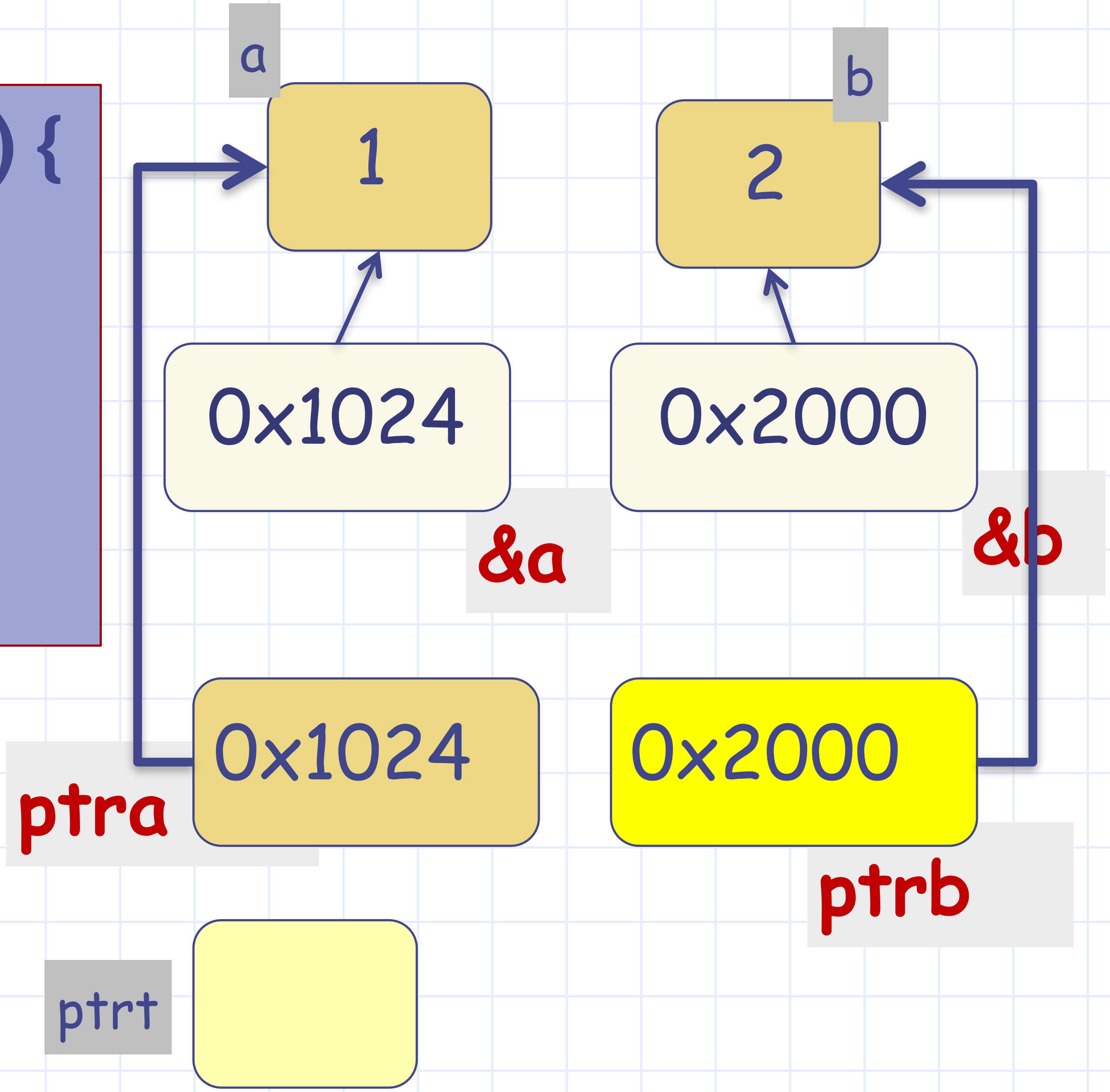
```
void
swap(int *ptra, int *ptrb)
{
    int t;
    t = *ptra;
    *ptra= *ptrb;
    *ptrb =t;
}
```

```
int main() {
   int a = 1, b = 2;
   swap(&a, &b);
   printf("a=%d, b=%d", a,
b);
   return 0;
}
```

1. The function swap() uses pointer to integer arguments, int *ptra and int *ptrb.
2. The main() function calls swap(&a,&b), i.e., passes the addresses of the ints it wishes to swap.

Question: Will the following code perform swap correctly?

```
void swap(int *ptra, int *ptrb) {
    int *ptrt;
    ptrt = ptra;
    ptra= ptrb;
    ptrb =ptrt;
}
```



What is the output of the following code?

```
#include <stdio.h>

int foo(char *parr)
{
        int cnt=0;
        while(*parr!='\0')
        {
                printf("%s\n",parr);
                parr++;
                cnt++;
        }
        return cnt;
}
int main()
{
        char arr[]="text";
        char *parr = arr;
        printf("%d\n",foo(parr) );
        return 0;
}
```

What is the output of the following code?

```
#include <stdio.h>

int foo(char *parr)
{
        int cnt=0;
        while(*parr!='\0')
        {
                printf("%s\n",parr);
                parr++;
                cnt++;
        }
        return cnt;
}
int main()
{
        char arr[]="text";
        char *parr = arr;
        printf("%d\n",foo(parr) );
        return 0;
}
```

Output is:

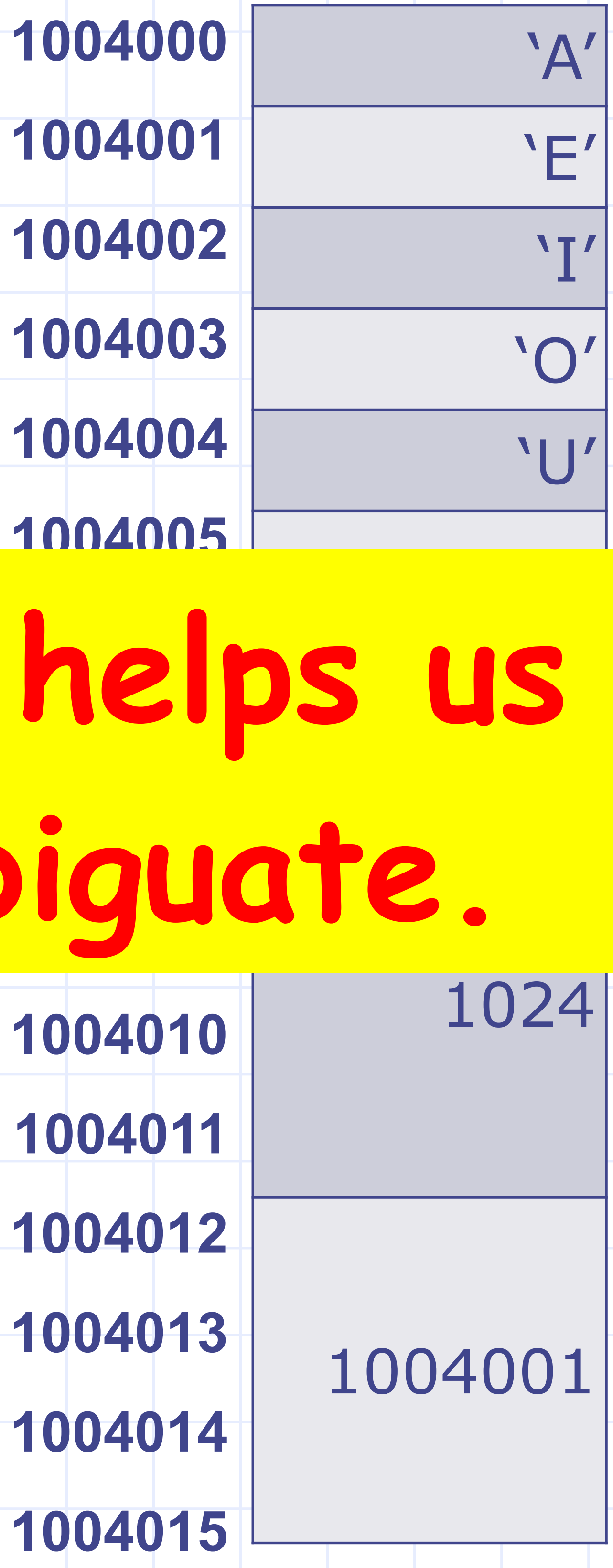
```
text
ext
xt
t
4
```

Simplified View of Memory: Recap

- Content of the 4-blocks starting at address 1004012
  - ✓ 1004001
- Without knowing the context it is not possible to determ significance of number
  - ✓ It could be an intege 1004001
  - ✓ It could be the "location" of the block that stores 'E'

"Type" helps us disambiguate.

How do we decide what it is?

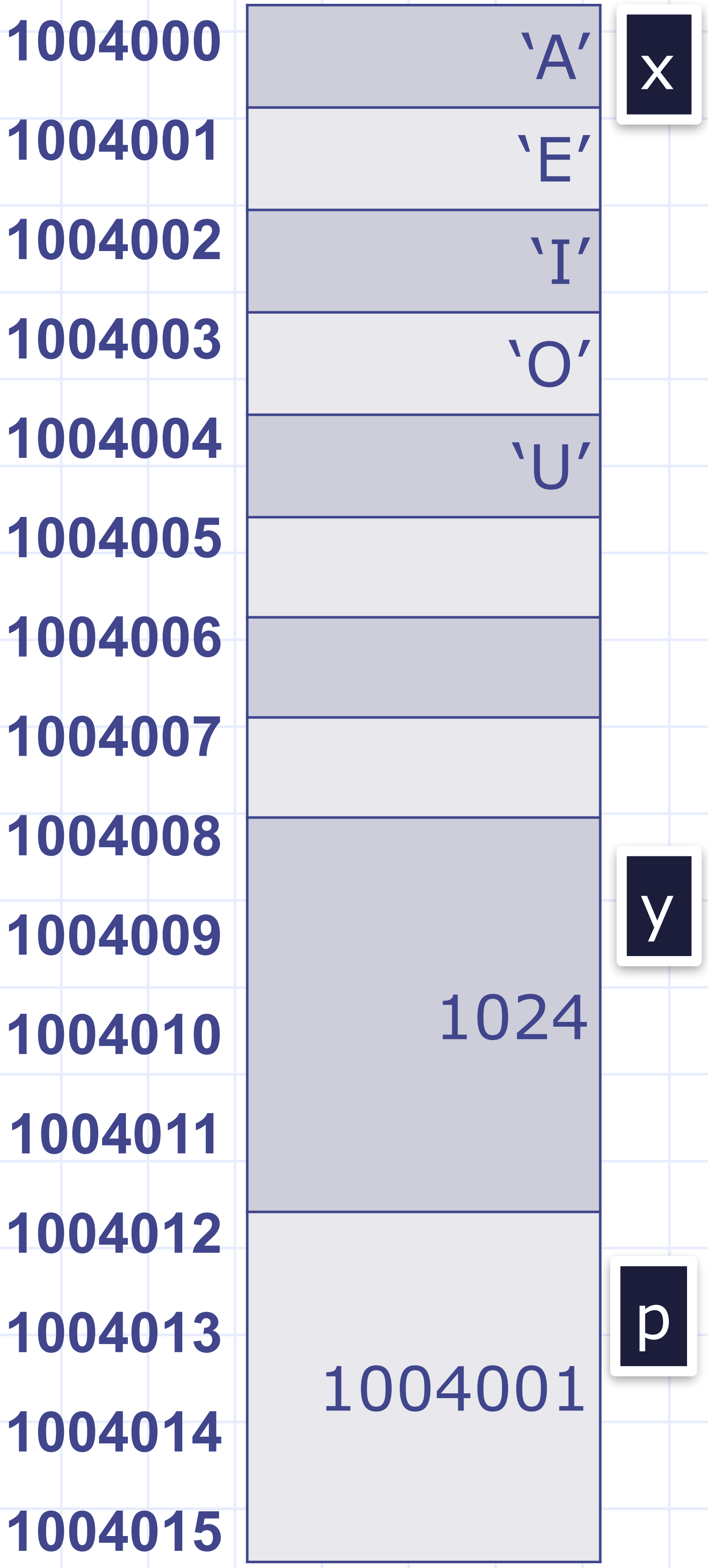

1004000 'A'
1004001 'E'
1004002 'I'
1004003 'O'
1004004 'U'
1004005
1024
1004010
1004011
1004012
1004013
1004001
1004014
1004015

Simplified View of Memory

- In programming also, “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



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		

What is the output of the following code?

```
#include <stdio.h>

int main()
{
        int *px;
       *px = 100;
       printf(“%d”,*px);
       return 0;
}
```

Should result in run time error as px points to an integer, but the integer variable is not defined (but assigned a value)

What is the output of the following code?

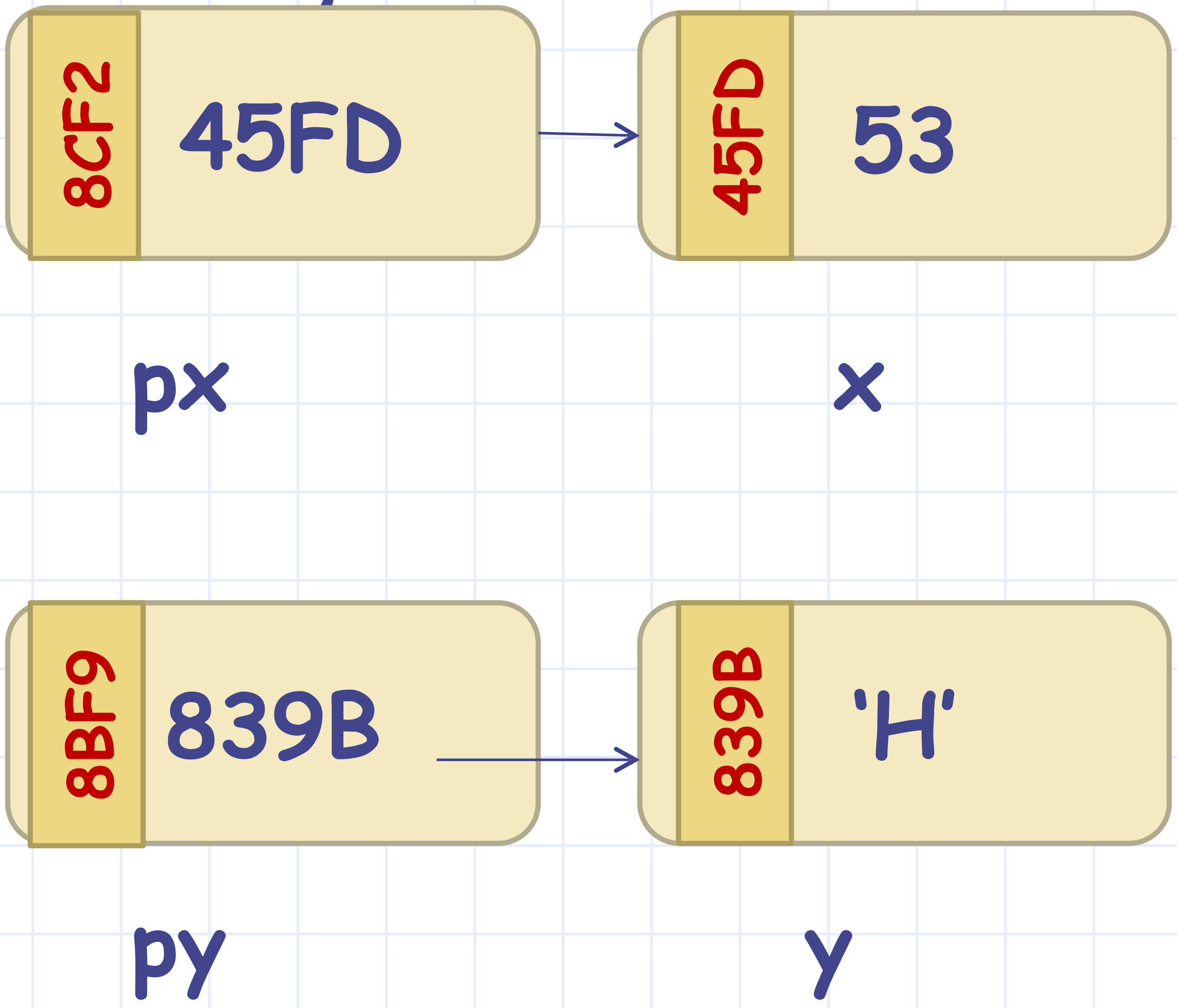
```
#include <stdio.h>

int main()
{
        int *p, x, y;
        x=10; y = 20;
        p = &x;
        *p = *p +y;
        p = &y;
        *p = x- *p;
        x = x-*p;
        printf("%d %d",x, y);
        return 0;
}
```

Output: 20 10

Pointers: Visual Representation

- Typically represented by box and arrow diagram

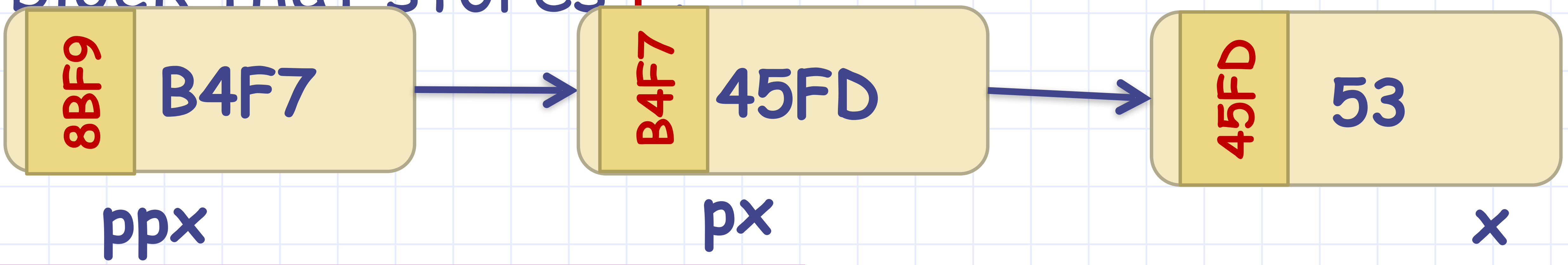


- x is an int variable that contains the value 53.
- Address of x is 45FD.
- px is a pointer to int that contains address of x.
- y is a char variable that contains the character 'H'.
- Address of y is 839B.
- py is a pointer to char that contains address of y.

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.



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

```
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 memory.

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) returns the number of bytes a variable of type int uses up in your computer's memory.

sizeof Examples

```
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

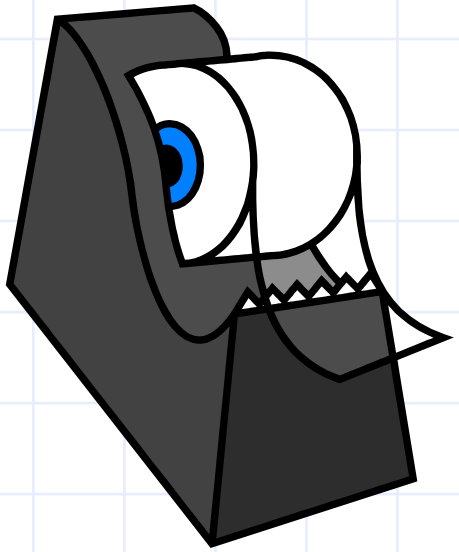
Static Memory Allocation

- 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

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: malloc



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

WAIT!! Doesn't void means "nothing" in C? What is the meaning of void*? Pointer to nothing!

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).

free: Example

malloc: allows us to allocate memory. It is our job to release the memory once we are done using the memory

```
float *f;
f= (float*) malloc(10 * sizeof(float));
//use f
free(f);
```

memory in f is released, future references to f will result in error if memory in f not initialised

Exercise

- Write a program to read two integers, n, m and store powers of n from 0 up to m (n^0, n^1, …, n^m)

Exercise

- Write a program to read two integers, n, m and store powers of n from 0 up to m (n^0, n^1, …, n^m)

```
#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]);
    free(pow);
    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”
- C uses the value 0 or name NULL
- In Boolean context, NULL is equivalent to false, any other pointer value is equivalent to true
- A malloc call can return NULL if it is not possible to satisfy memory request
  - negative or ZERO size argument
  - TOO BIG size argument