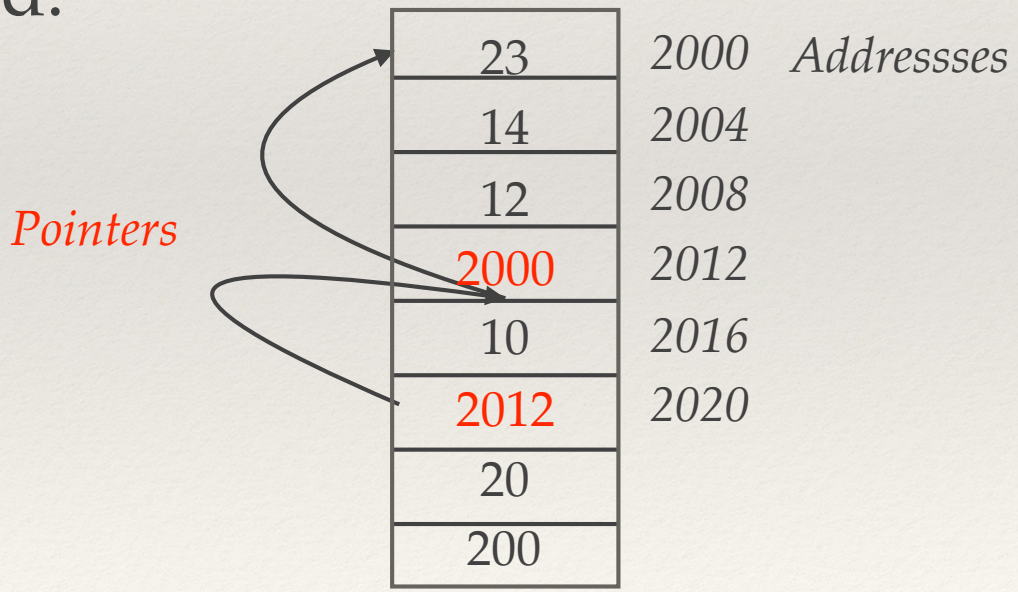
Allocate…use…free…repeat…

Memory

Let's start working with memory and pointers…endless hours of fun!

What is Memory?

- Memory is an **internal** storage area in the computer.
- A **pointer** is a programming language object whose value "*points to*" the location (memory **address**) where another object is stored.



Memory Allocation

- When you declare a variable you are **statically** allocating memory.

```
int count;
```

  - Allocates 4 bytes of memory for the integer count

```
float bigNum[4];
```

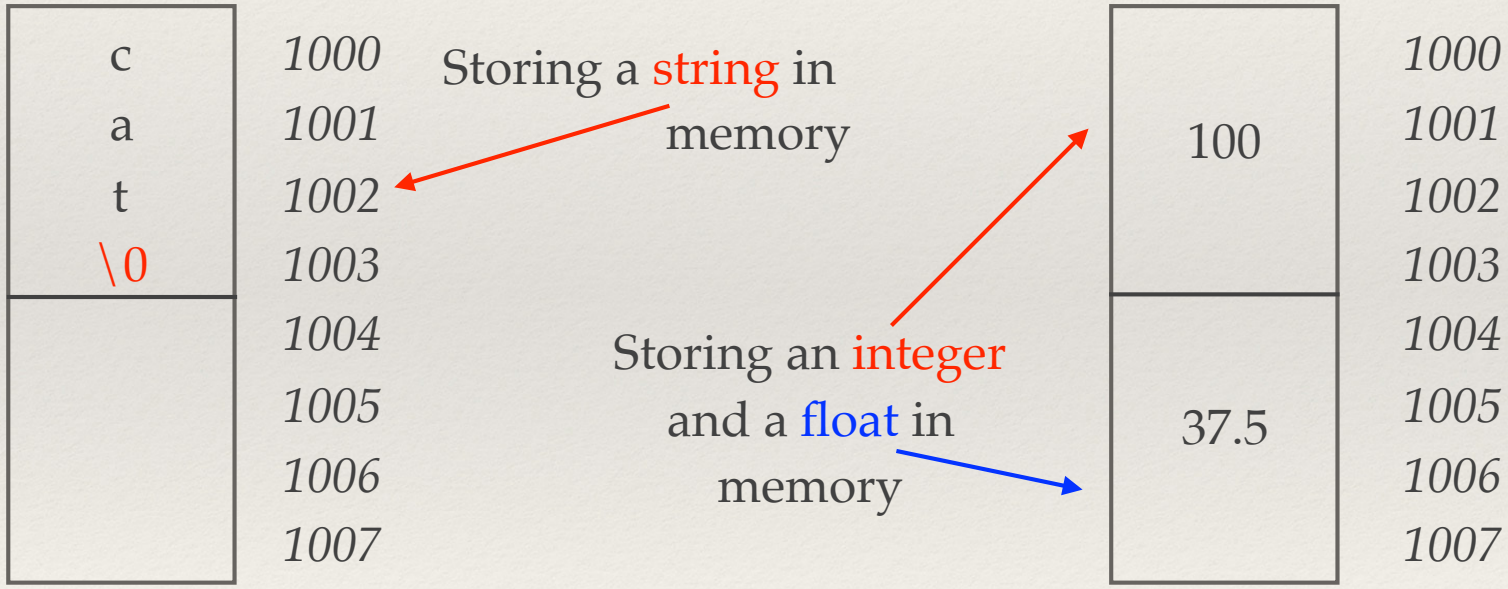
  - Allocates space for an array of 4 floating point numbers (4 bytes X 4 = 16 bytes)

Dynamic Memory Allocation

- **Dynamic** memory allocation is used to create a variable of a size determined while the program is **running** (not when it is **compiled**).
- Terms:
  - Compile time
  - Run time

Memory Addresses and Pointers

- Memory is organized as a series of bytes each of which has its own address.



Memory Addresses and Pointers

- We can **reference** memory in different ways.
- **Variable name**

```
int count;

float sum;
```

- **Address**
  - We can find the address of a variable by using the variable name and the & operator

The & Operator

```
printf ( “%d\n”, count );
```

- prints out the value of count

```
printf ( “%p\n”, &count );
```

- prints out the memory address where count is stored
- `%p` means = format as an address

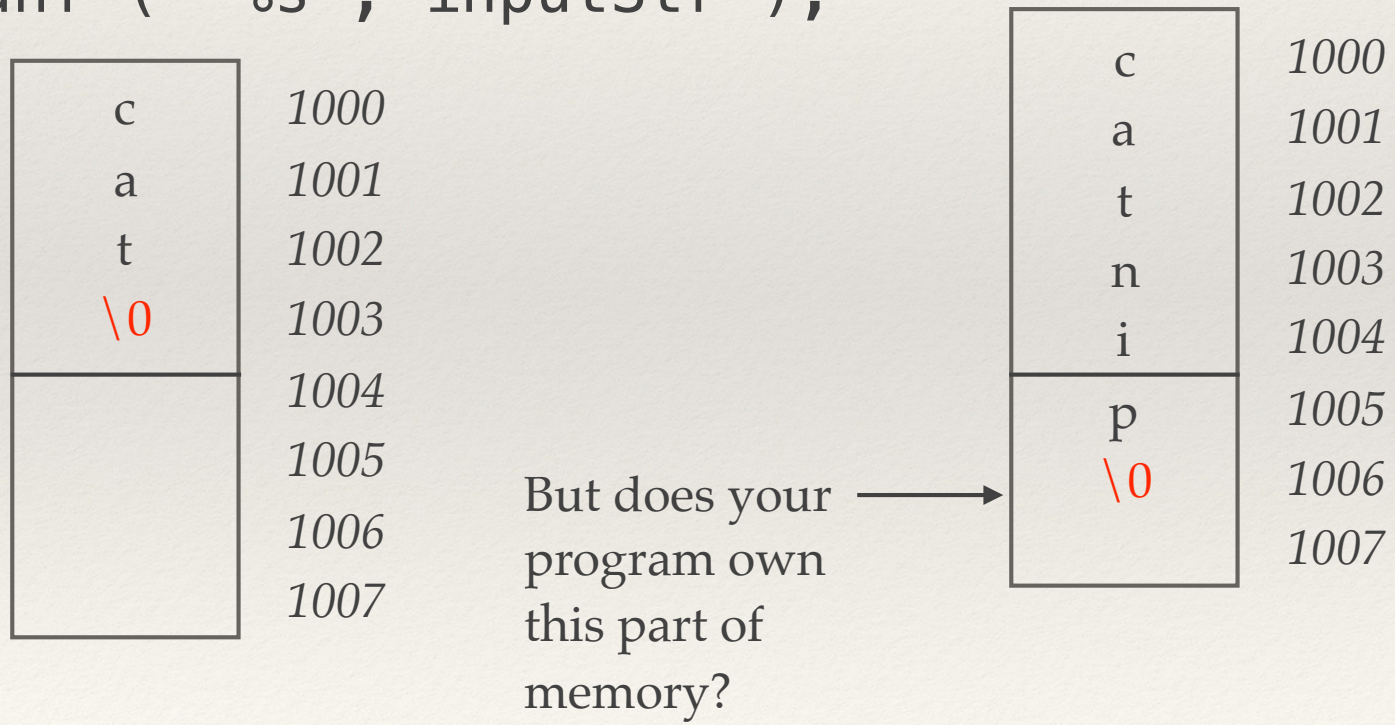
The & Operator and `scanf`

```
scanf ( “%d %d”, &num1, &num2);
```

- `&num1` points to the location for `scanf` to use to store the integers that you type in.
- `%d` tells `scanf` what type of variable is being read in so that the size of the variable can be determined.

Sidebar: Why you should not use `scanf`

```
char inputStr[5];
printf ( “Input a string: “ );
scanf ( “%s”, inputStr );
```



The `scanf` Problem

```
#include <stdio.h>
int main()
{
   char inputStr[5];
   printf ( "Enter a string: " );
   scanf ( "%s", inputStr );
   printf ( "The string entered was %s\n", inputStr );

   return(0);
}
```

```
$ gcc -Wall -ansi -o scanfProblem scanfProblem.c

$ ./scanfProblem
Enter a string: cat
The string entered was cat

$ ./scanfProblem
Enter a string: abcdefghijklmnopqrstuvwx
The string entered was abcdefghijklmnopqrstuvwx
Segmentation fault
$
```

Memory and Pointers

- The & operator allows us to discover the address of a variable.
- To store the address we need to define a variable of type **pointer**.

```
int count;
int *ptr;
```

- The `*` means that the variable is a pointer (in this case to a variable of type integer). But it is **not** an integer!

Memory and Pointers

```
int count;
int *ptr;

count = 7;
ptr = &count;

printf ( “%d\n”, count );
printf ( “%p\n”, &count );
printf ( “%p\n”, ptr );
```

- the * means that the variable is a pointer to an integer

- store the address of `count` in `ptr`

- outputs 7
- outputs the address of `count`
- outputs the address of `count`

```
7
0x7fff585ff66c
0x7fff585ff66c
```

Dereferencing

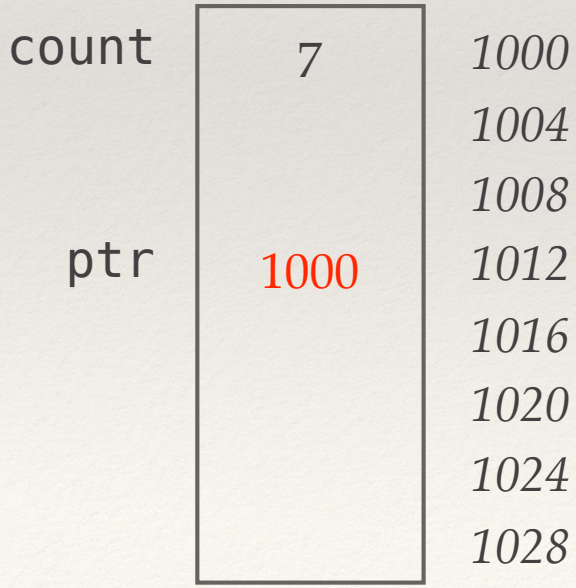
- Once you have the address of a variable you can use it to access the value by **dereferencing** the pointer using the * operator.

```
int count;
int *ptr;
count = 7;
ptr = &count;
printf ( “%d\n”, *ptr );
```

*The * means that you take the value stored in ptr and reference the memory that it points at.*

Dereferencing

- If `count` is stored at address *1000* then `ptr` is *1000* and `*ptr` is the value stored at address *1000* and that is 7.
- & gives the address or location of a variable
- `*` looks at a location (address) and extracts the value



`*ptr` - go to the memory address in `ptr` which is *1000* and get the value there - which is 7.

`&count` produces the memory address of `count` which is *1000* and this can be stored in a pointer variable like `ptr`.

How Arrays are Stored

- Arrays are simply a sequence of memory locations with some convenient access structures built in.

```
int count[5];
a[3] = 9;
a[1] = 7;
```

Each int is 4 bytes long.

	value	*address*
a[0]		1000
a[1]	7	1004
a[2]		1008
a[3]	9	1012
a[4]		1016

You can print the address of element 3 using `&(a[3])`

Arrays

- The address of element 3 is `&(a[3])`.

  `&(a[3])`

  = **base address** + (3 * `sizeof int`)

  = 1000 + (3 * 4)

  = **1012**

	value	*address*
a[0]		**1000**
a[1]	7	1004
a[2]		1008
a[3]	**9**	**1012**
a[4]		1016

Pass by Value

- In C, you can only pass **values** to *function* call
- Get information out through a return.

```
int function ( int f )
{
    …
    …
    f = 7;
    …
    return ( f );
}
```

Changes are not passed back by operations such as this.

If you wish to change the value of `f` in the calling routine then you could `return` it.

Pass by Value

a bad swap function

```
void bad_swap(int x, int y)
{
   int original_x = x;

   x = y;
   y = original_x;
}
```

using bad_swap

```
int a = 10, int b = 23;

printf("a = %d, b = %d \n", a, b);

bad_swap(a, b)

printf("a = %d, b = %d \n", a, b);
```

output

```
a = 10, b = 23
a = 10, b = 23
```

Changes are not passed back
s such as this.

```
...
f = 7;
...
return (f);
}
```

If you wish to change the value of f in the calling routine then you could `return` it.

Pass by Value

- In C, yo
- Get info

```
int fu
{
    …
    …
    f =
    …
}
```

```
int a = 20, b = 30, f = 40

printf("a = %d, b = %d, f = %d\n", a, b, f)

b = function(2 * a)

printf("a = %d, b = %d, f = %d\n", a, b, f)

a = function(f)

printf("a = %d, b = %d, f = %d\n", a, b, f)
```

output

```
a = 20, b = 30, f= 40
a = 20, b = 7, f = 40
a = 7, b = 7, f = 40
```

Passing Variables "by Reference"

- Passing variables by reference requires the use of the address (&) and pointer (∗) operators.
- In C, you can only pass **values** to *function* calls but with a pointer we can pass the **address** of a variable instead of its value.
- Called "pass by reference"

Passing by Reference

- To access a variable that is passed by reference we need to use the value at the given address - the address passed to the subroutine.
- This is called **dereferencing** and is done using the * operator.

Passing by Reference

Declare `p` as a **pointer** to `int`

```
void Function ( int *p ) {
    *p = 10;
}
```

To access the value stored at the address in `p` we need to use the `*` operator to access what `p` *"points to"*

Passing by Value

- **Pass by Value**

```
int function ( int f )
{
    …
    …
    f = 7;
    …
    return ( f );
}
```

Changes are not passed back by operations such as this.

If you wish to change the value of `f` in the calling routine then you could `return` it.

Pass by Reference

- **Pass by Reference**

```
int function ( int *f )
{
    …
    …
    *f = 7;
    …
    return ( );
}
```

The changes to `f` are available to the calling routine since the information in memory has been changed.

This can still be used to return other information. You do not need to return `f` since its updated value is already available to the calling routine.

Pass by Reference

a working swap function

```
void swap(int *x, int *y)
{
   int original_x = *x;

   *x = *y;
   *y = original_x;
}
```

using swap

```
int a = 10, b = 23;

printf("a = %d, b = %d \n", a, b);

swap(&a, &b)

printf("a = %d, b = %d \n", a, b);
```

output

```
a = 10, b = 23
a = 23, b = 10
```

Pass by Reference

a working swap function

```
void swap(int *x, int *y)
{
   int original_x = *x;

   *x = *y;
   *y = original_x;
}
```

another way of using swap

```
int *a = 10, int *b = 23;

printf("a = %d, b = %d \n", *a, *b);

swap(a, b)

printf("a = %d, b = %d \n", *a, *b);
```

output

```
a = 10, b = 23
a = 23, b = 10
```

Passing Arrays

- Arrays are automatically passed by reference since they are **pointers**.

```
int function ( int a[] )
{
    …
    a[3] = 10;
    …
    return (0);
}
int main ( )
{
    int a[10];
    int function();
    …
    function ( a );
    …
}
```

Summary

- **Pass by Value with a Variable**

```
int x;                  int function ( int x )
function ( x );         {
                            … x = 3; /* Only visible in function */
                        }
```

- **Pass by Reference with a Variable**

```
int x;                  int function ( int *x )
function ( &x );        {
                            … *x = 3; /* Visible in calling routine */
                        }
```

Summary

- **Pass by Reference with a Pointer**

```
int *a;                 int function ( int *a )
function ( a );         {
                            … *a = 3; /* Visible in calling routine */
                        }
```

- **Pass by Reference with an Array**

```
int b[5];               int function ( int b[] )
function ( b );         {
                            … b[5] = 3; /* Visible in calling routine */
                        }
```

Dynamically Allocated Memory

- Statically allocated memory has a fixed size that is determined at compile time.
- Dynamic memory is allocated at run time, *i.e.* size is determined once the program is running.
- This is very powerful as it allows programs to allocate the amount of memory they require instead of creating a huge amount of memory space for a variable that may use very little of it.
- *Example*: allocating a string that is 1000 characters long but only using 10 characters.

malloc()

- The command to allocate memory is `malloc()`.
- You use a pointer to store the base location of the allocated memory.
- Let's allocate a 10 character long string:

```
char *name;
name = malloc ( sizeof(char) * 10 );
```

number of characters

pointer to the string

calculate the size of a char

number of bytes to allocate

Allocating 100 Integers

```
int *numbers;
numbers = malloc ( sizeof(int) * 100 );
```

- `malloc()` returns the address of the memory that has been allocated. This is stored in the pointer variable `numbers`.
- Now you can access the allocated memory using the pointer and pointer arithmetic or you can treat it as an array.

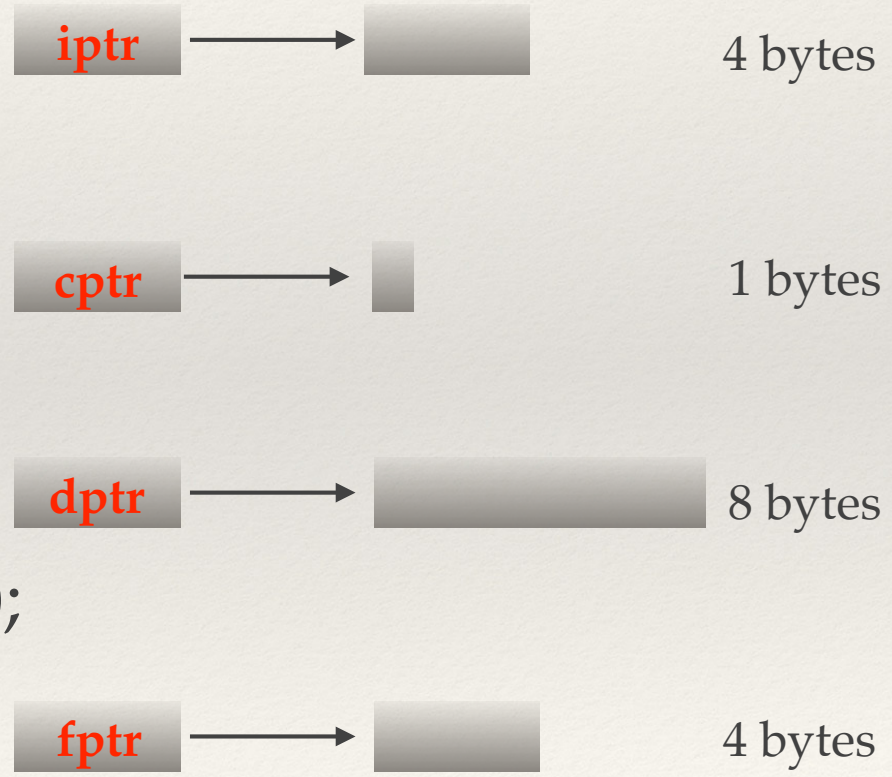
Accessing as an Array

```
int *ptr;
ptr = malloc ( sizeof(int) * 20 );
ptr[0] = 3;
ptr[1] = 7;
printf ( “%d\n”, ptr[1] );
```

Pointer Type

- The pointer **type** indicates what the pointer is pointed at, **not** the pointer variable itself.

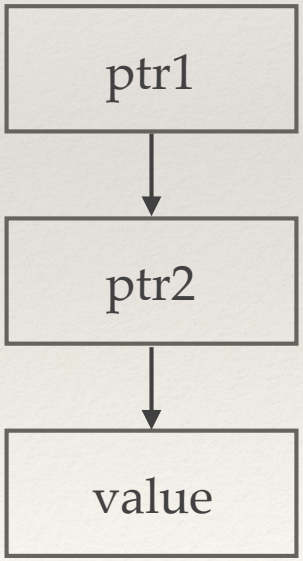
```
int *iptr;
iptr = malloc ( sizeof(int) );
char *cptr;
cptr = malloc ( sizeof(char)  );
double *dptr;
dptr = malloc ( sizeof(double) );
float *fptr;
fptr = malloc ( sizeof(float) );
```



What can a Pointer Reference?

1. Static variables
2. Dynamically allocated memory
3. Functions
4. Structures
5. Other pointers

Q. *Why is this important?*
A. *Because an array of pointers is how multi-dimensional arrays are created.*

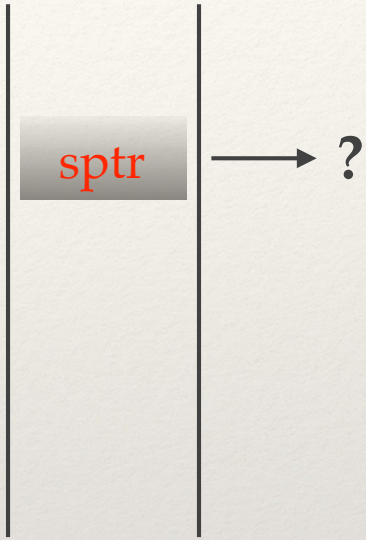


Creating an Array of Strings

- Create an array of pointers to characters.
- Put the strings (array of characters) at the end of the pointers.

Code to Allocate an Array of Strings

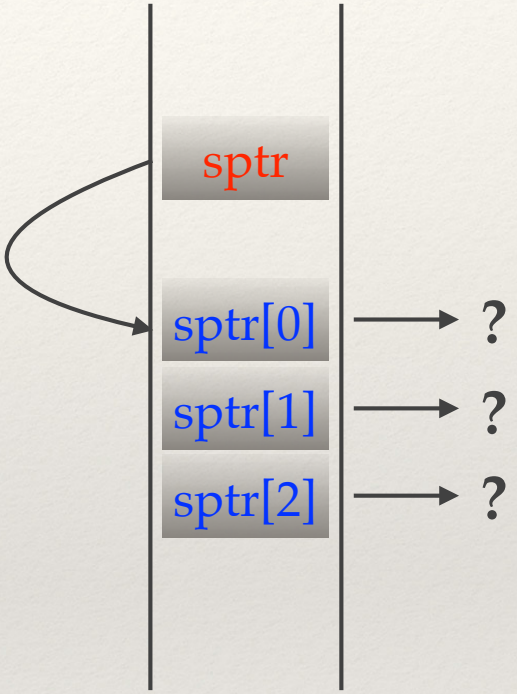
```
char **sptr;

int num = 3;
int i;

sptr = malloc ( sizeof(char *) * num );

for ( i=0; i<num; i++ ) {
   sptr[i] = malloc ( sizeof(char) * 5 );
   strcpy ( sptr[i], “name” );
}
```



1. Declare a pointer-to-pointer variable - call it `sptr`

Code to Allocate an Array of Strings

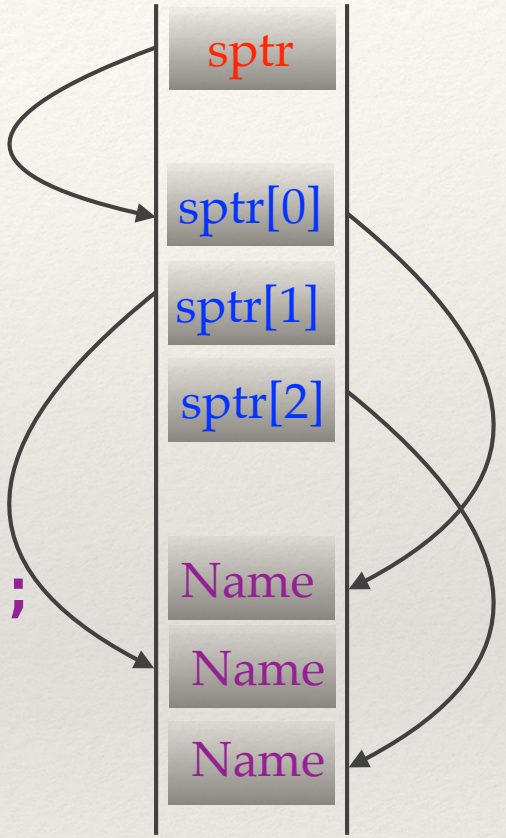
```
char **sptr;

int num = 3;
int i;

sptr = malloc ( sizeof(char *) * num );

for ( i=0; i<num; i++ ) {
   sptr[i] = malloc ( sizeof(char) * 5 );
   strcpy ( sptr[i], “name” );
}
```



1. Declare a pointer-to-pointer variable - call it `sptr`
2. Allocate the memory for the character pointers (array)

Code to Allocate an Array of Strings

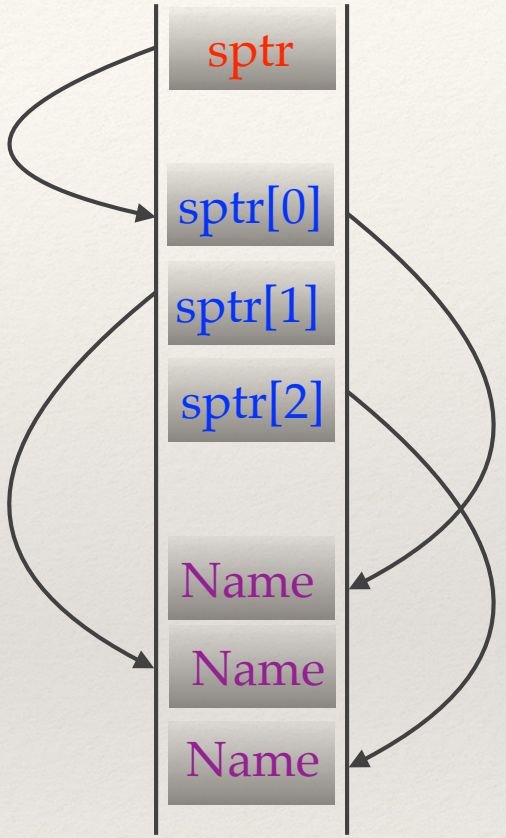
```
char **sptr;

int num = 3;
int i;

sptr = malloc ( sizeof(char *) * num );

for ( i=0; i<num; i++ ) {
   sptr[i] = malloc ( sizeof(char) * 5 );
   strcpy ( sptr[i], "Name" );
}
```



3. For each element in the array, allocate space for the character string and point to them from the array of pointers. Copy the string `Name` into the allocated space (3X).

Print out the Array of Strings

```
for ( i=0; i<num; i++ ) {
  printf ( “%s\n”, sptr[i] );
}
```



4. The strings can be treated as an array and referenced by sptr[i]

Sidebar: `strcpy`

```
strcpy ( char *destination, char *source );
```

- Copies the string pointed to by source (including the terminating '\0') into the memory pointed to by destination.
- **Problem**: what are the sizes of the destination and the source? Destination **MUST** be large enough but what if it is not? Buffer Overflow!!
- **Solution**:

```
strncpy (char *destination, char *source, int max_copy);
```

But there is a problem with `strncpy()` too - what if there is no '\0' character in the first `max_copy` characters?

Notes

- The number and lengths of the items/strings created in the `malloc` can be determined at run time - totally derived from variables.
- When you `free` the memory that you have allocated
  - free both the array of pointers and all of the strings
  - the string storage must be freed first