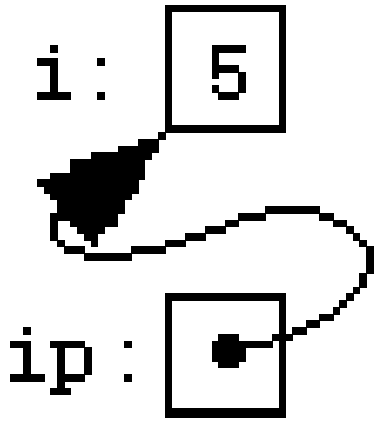
Pointers

What is a pointer?

A pointer is a variable that holds address of another object (i.e., variable)

Basic pointer operations

```
int i = 5;
int *ip = &i;
```

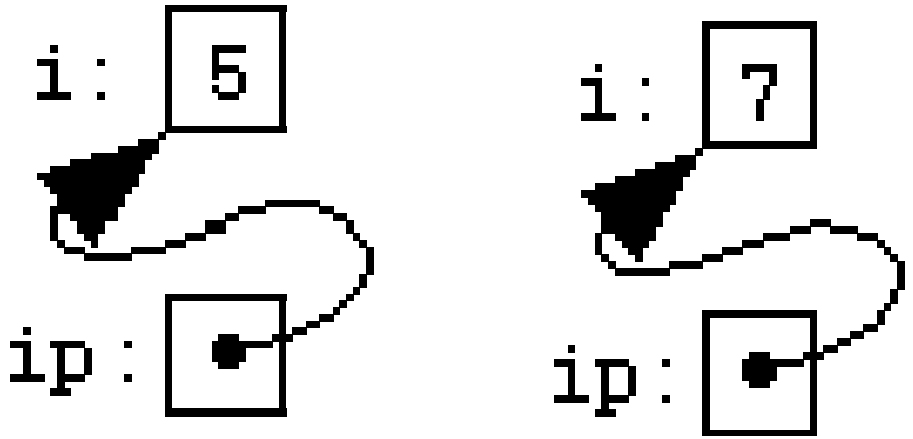



```
printf("%d\n", *ip);
```

Prints 5

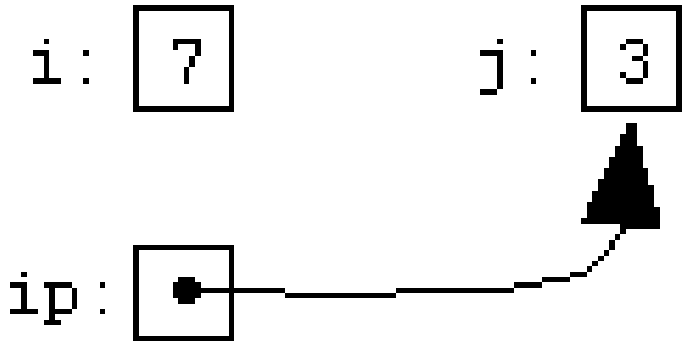
Basic pointer operations

```
*ip = 7;
```



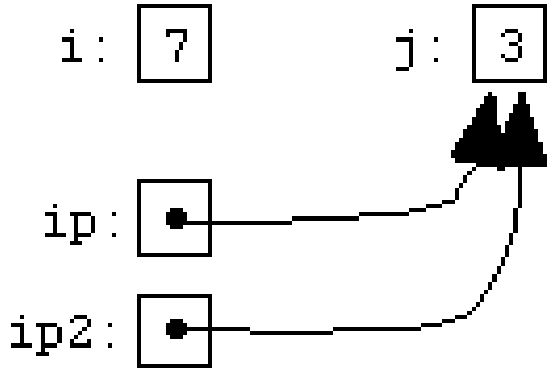
Basic pointer operations

```
int j = 3;
ip = &j;
```



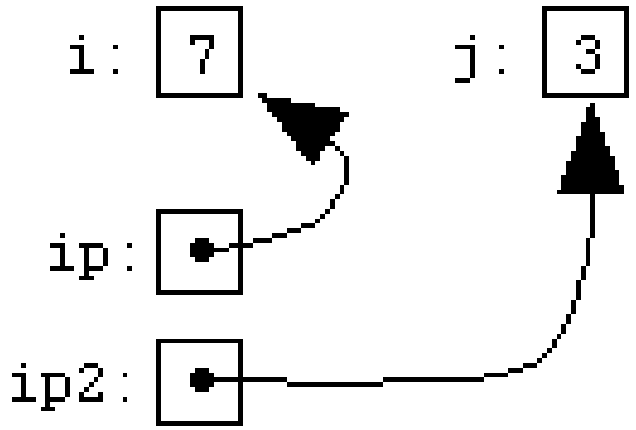
Basic pointer operations

```
int *ip2;
ip2 = ip;
```



Basic pointer operations

```
ip = &i;
```



Pointers and arrays

- **Is there a difference?**

Pointers and arrays

- **Array indexing: a[0], a[1], a[2], …**

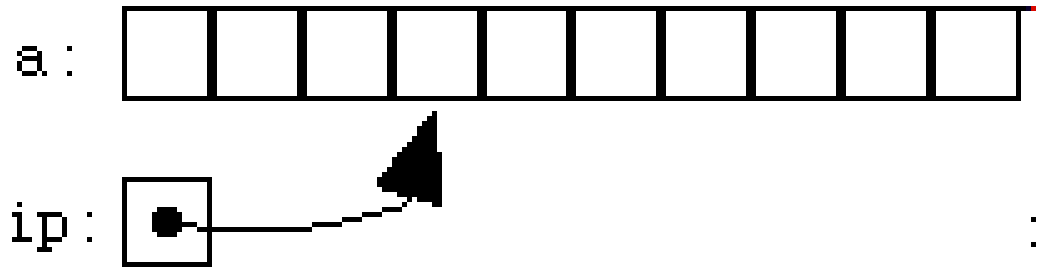
- **int *ip = &a[0];**
- **Pointer indexing: *ip, *(ip+1), *(ip+2), …**

- **In general a[i] is “equivalent” to *(ip+i)**

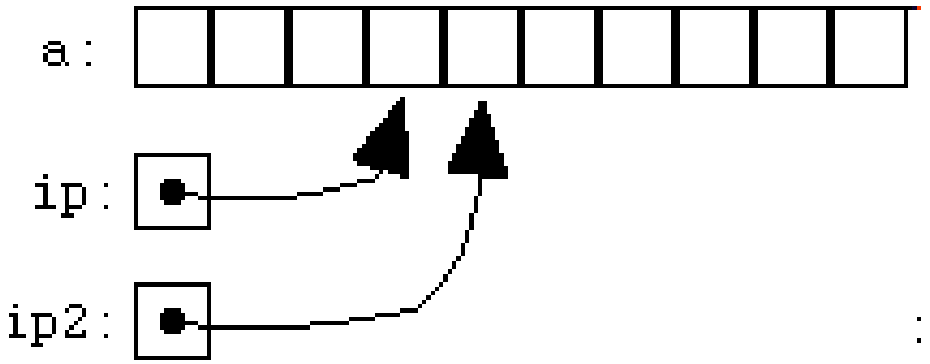
Pointer Arithmetic

- **ip2 = ip1 + 3;**
- **ip2 - ip1 == 3 (true or false?)**

```
int *ip;
int a[10];
ip = &a[3];
```




```
ip2 = ip + 1;
```



Autoincrement operator ++ (and its companion, --)

- Both of these are defined for pointers
- Array version: a[i++]
- Pointer version: *ip++

Prefix form is defined too

- **Array version: a[++i]**
- **preincrement form: *++ip**
- ***ip-- and *--ip.**

Copying an array using pointers

```
int array1[10], array2[10];

int *ip1, *ip2 = &array2[0];
int *ep = &array1[9];
for(ip1 = &array1[0]; ip1 <= ep; ip1++)
     *ip2++ = *ip1;
```

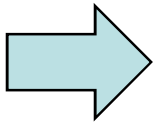
[http://www.eskimo.com/~scs/cclass/notes/sx1

Comparing strings using pointers

```
char *p1 = &str1[0], *p2 = &str2[0];

while(1)
{
    if(*p1 != *p2)
       return *p1 - *p2;
    if(*p1 == '\0' || *p2 == '\0')
        return 0;
    p1++;
    p2++;
}
```

String copy using pointers

```
char *dp = &dest[0], *sp = &src[0];

while(*sp != '\0')
    *dp++ = *sp++;

*dp = '\0';
```



```
while(*sp)
    *dp++ = *sp++;
```

Arrays of pointers

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int* pArray[10];

   int a;
   int b;

   pArray[0] = &a;
   pArray[1] = &b;

   system("pause");
}
```

Pointers to functions

```
// function returning an int
// and having two arguments (an int and a char)
int fun1(int a, char c);

// function returning pointer to an int
// and having one int argument
int *fun2(int a);

// pointer to function returning int
// and having two arguments (an int and a char)
int (*funp)(int a, char c);

// two ways to call the function
(*funp)(1,'b');
funp(1,'c');
```

Example

```
#include <stdio.h>
#include <stdlib.h>

int fun1(int a, int b)
{
    printf("fun1\n");
    return a+b;
}

int fun2(int a, int b)
{
    printf("fun2\n");
    return a-b;
}

int main()
{
    // pointer to function returning int  and having two arguments: an int and a float
    int (*funp)(int a, int b);

    funp = fun1;   // take the address of the function and assign it to the  function pointer
    (*funp)(1,2);  // call the function using the pointer


    funp = fun2;   // reassign the pointer to point to fun2
    funp(1,2);     // an alternative way of calling a function using a pointer

    system("pause");
}
```

Pointers to Structures

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
   int value;
} node_t;

int main()
{
   node_t Node;
   Node.value = 5; // initialize it to 5
   printf("value = %d\n", Node.value);

   // pointer to the statically allocated struct Node
   node_t *p = &Node;
   p->value = 6; // change it to 6
   printf("value = %d\n", p->value);
}
```

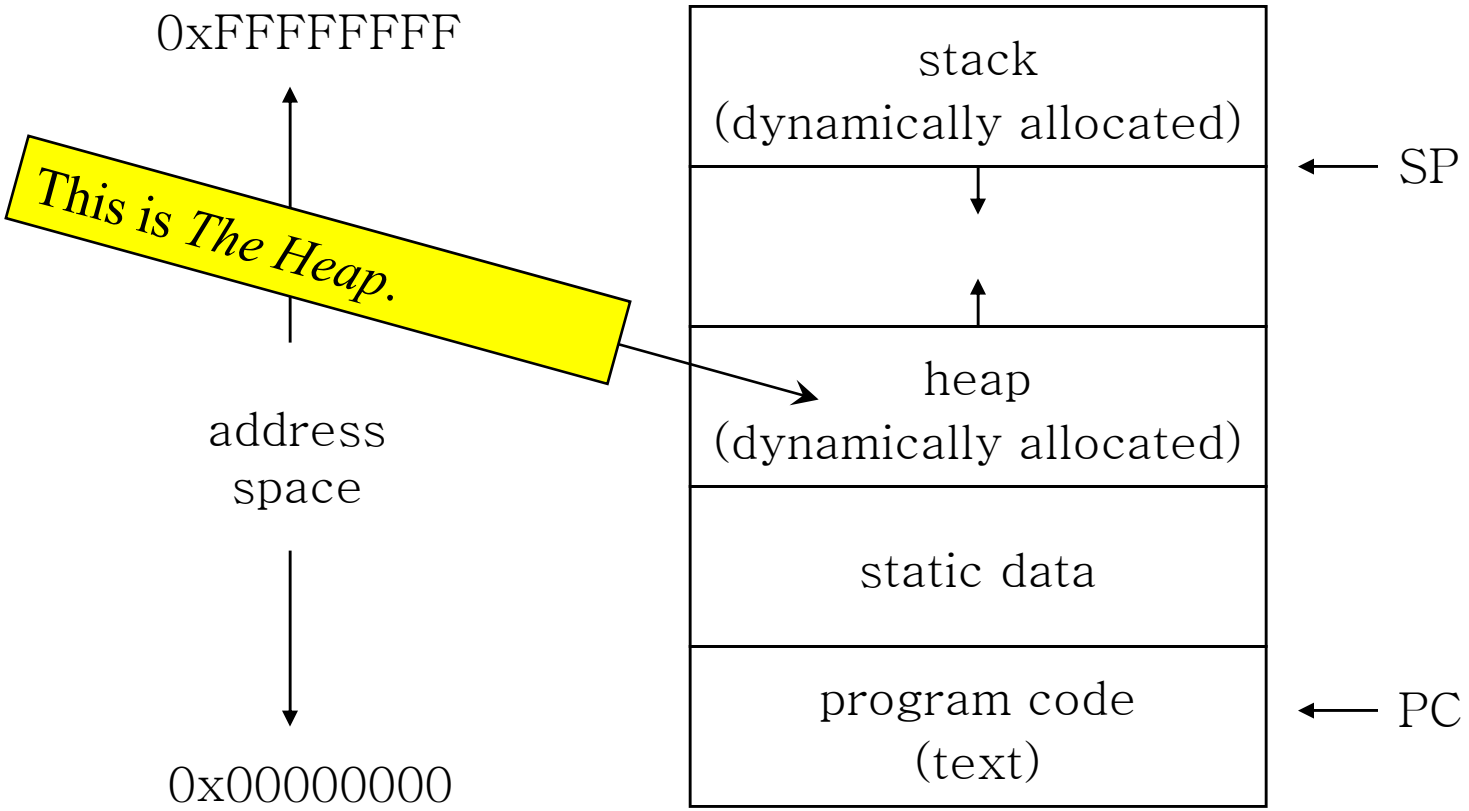
Dilemma

- **Question:–**
  - **If strings are arrays of characters, …**
  - **and if arrays cannot be returned from functions, …**
  - **how can we manipulate variable length strings and pass them around our programs?**
- **Answer:–**
  - **Use storage allocated in *The Heap! (i.e., dynamic memory allocation)***

Definition — *The Heap*

- **A region of memory provided by most operating systems for allocating storage *not* in *Last in, First out* discipline**
  - **I.e., not a stack**
- **Must be explicitly allocated *and* released**
- **May be accessed *only* with pointers**
  - **Remember, an array is equivalent to a pointer**
- **Many hazards to the *C* programmer**

Static Data Allocation



Allocating Memory in The Heap

- See `<stdlib.h>`

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- `malloc()` — allocates `size` bytes of mem… from the heap and returns a point…
  - NULL pointer if allocation fails fo…
- `free()` — returns the chu… …mory pointed to by `ptr`
  - *Must* have been allocated by `malloc` or `calloc`

Segmentation fault and/or bus-time error if bad pointer

Allocating Memory in The Heap

- See `<stdlib.h>`

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

`free()` knows *size* of chunk allocated by `malloc()` or `calloc()`

- `malloc()` — allocates `size` bytes of memory from the heap and returns a pointer to it.
  - NULL pointer if allocation fails for any reason
- `free()` — returns the chunk of memory pointed to by `ptr`
  - ***Must*** have been allocated by `malloc` or `calloc`

Notes

- `calloc()` is just a variant of `malloc()`
- `malloc()` is analogous to `new` in *C++* and Java
  - `new` in *C++* actually calls `malloc()`
- `free()` is analogous to `delete` in *C++*
  - `delete` in *C++* actually calls `free()`
  - Java does not have `delete` — uses *garbage collection* to recover memory no longer in use

Typical usage of `malloc()` and `free()`

```
char *getTextFromSomewhere(…);

int main(){
  char * txt;
  …;
  txt = getTextFromSomewhere(…);
  …;
  printf("The text returned is %s.", txt);
  free(txt);
}
```

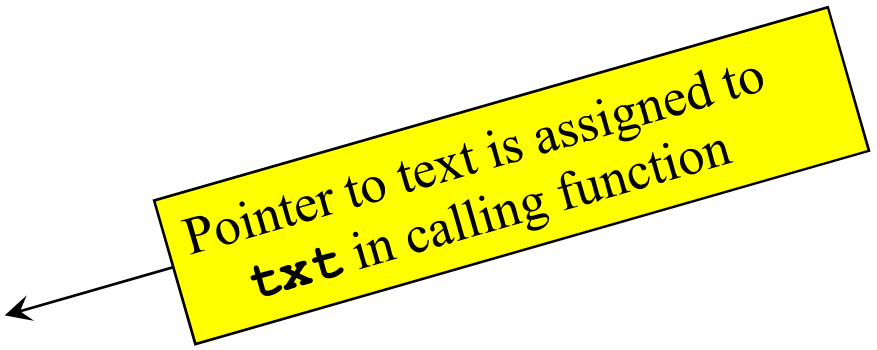
Typical usage of `malloc()` and `free()`

```
char * getTextFromSomewhere(…){
   char *t;
   ...
   t = malloc(stringLength);
   ...
   return t;
}

int main(){
   char * txt;
   …;
   txt = getTextFromSomewhere(…);
   …;
   printf("The text returned is %s.", txt);
   free(txt);
}
```

`getTextFromSomewhere()` creates a new string using **`malloc()`**

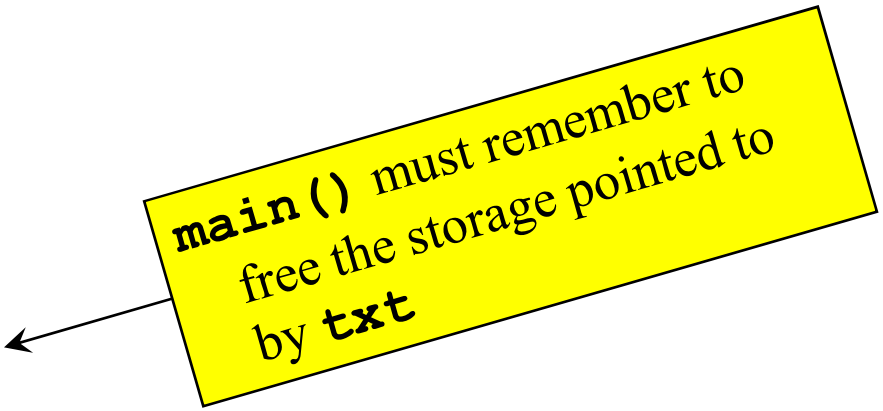
Typical usage of `malloc()` and `free()`

```
char * getTextFromSomewhere(…){
   char *t;
   ...
   t = malloc(stringLength);
   ...
   return t;
}

int main(){
   char * txt;
   …;
   txt = getTextFromSomewhere(…);
   …;
   printf("The text returned is %s.", txt);
   free(txt);
}
```



Usage of `malloc()` and `free()`

```
char *getText(…){
   char *t;
   ...
   t = malloc(stringLength);
   ...
   return t;
}

int main(){
   char * txt;
   …;
   txt = getText(…);
   …;
   printf("The text returned is %s.", txt);
   free(txt);
}
```



Definition – *Memory Leak*

- **The steady loss of available memory due to forgetting to `free()` everything that was `malloc`'ed.**
    - **Bug-a-boo of most large *C* and *C++* programs**

- **If you "forget" the value of a pointer to a piece of `malloc`'ed memory, there is no way to find it again!**
    - **Killing the program frees *all* memory!**

In class examples

- **See the class web page for source code.**