Dynamic Allocation in C

The previous examples involved only targets that were declared as local variables.

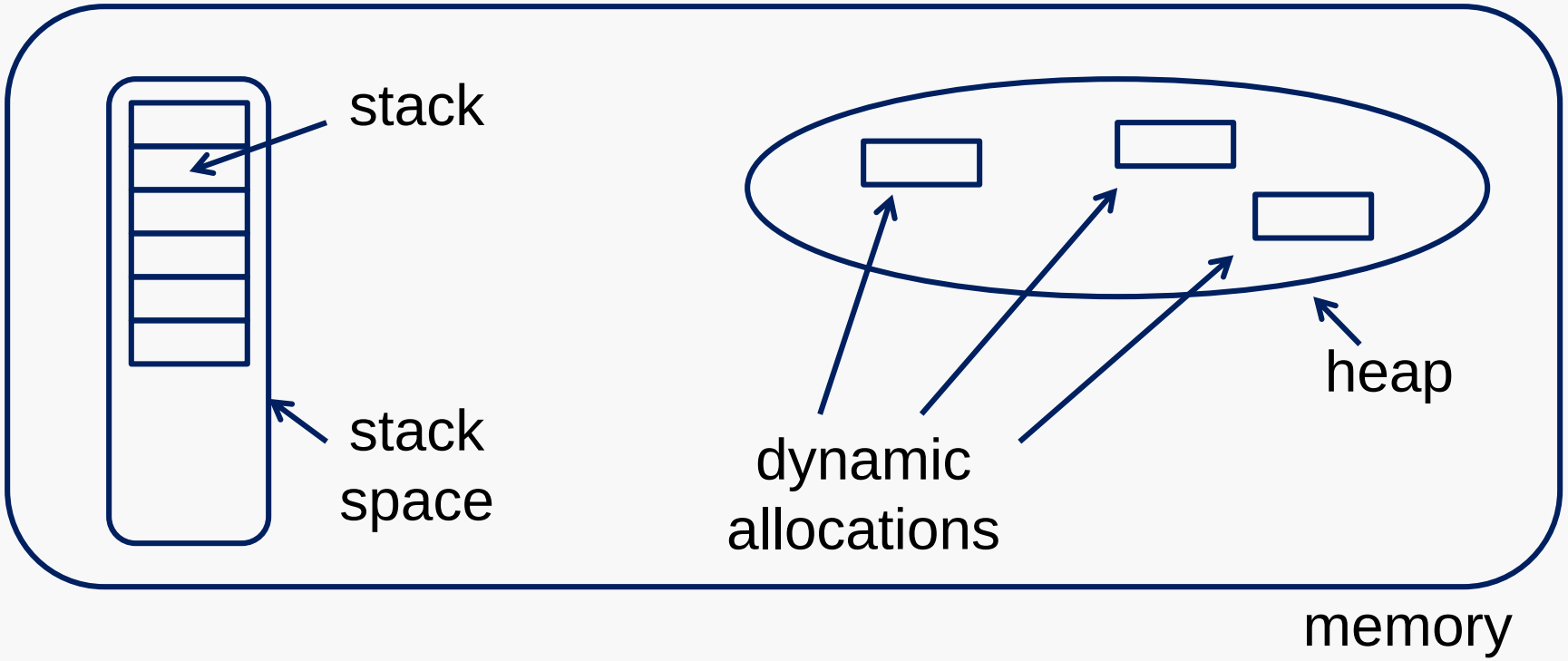
For serious development, we must also be able to create variables dynamically, as the program executes.

In C, this is accomplished via the Std Library function `malloc()` and friends:

`malloc()`	allocates a block of uninitialized memory; returns the address
`calloc()`	allocates a block of memory and clears it; returns the address
`realloc()`	resizes a previously allocated block of memory; returns the address

```
int  *A = malloc( 1000 * sizeof(int) );
char *B = malloc( 5000 );
uint64_t Size = 100;
double *C = malloc( Size * sizeof(double) );
```

The Heap



Dynamic allocations take place in a region of memory called the "heap".

Successful calls to `malloc()` return a pointer to a block of memory that is now available for your program to use.

The block of memory may be larger than your request (but you will never know that).

Allocating Arrays Dynamically

You allocate an array by allocating a suitably-sized block of memory:

```
int N;
. . .   // assume N is assigned a value

int  *A = malloc( N * sizeof(int) );    // allocate array
                                        //   dynamically

for (int pos = 0; pos < N; pos++) {     // access using ptr name

   A[pos] = pos * pos;
}
```

Any pointer name can be used with array syntax (bracket notation)… but you'd better make sure that the pointee really is an array.

Dynamic Allocation Failure

It is always possible that an allocation request will be denied; in that case, `malloc()` and friends will return `NULL`.

A deadly sin is to not check the return value from `malloc()` to be sure it isn't `NULL`:

```
int  *pA = malloc( 1000 * sizeof(int) );
if ( pA == NULL ) {
   fprintf(stderr, "Failed to allocate space for pA!\n");
   exit(1);
}
```

Without the check of `A`, the subsequent code will probably lead to a runtime error (unless there was no need for the array).

Deallocation in C

One of the most glaring differences between Java and C is how memory deallocation is accomplished.

In C, we have static allocations, local or automatic allocations, and dynamic allocations. The first two are of no particular interest here.

Everything that your C program allocates dynamically must eventually be deallocated.

The responsibility is yours.

Failure to deallocate memory in a timely but safe manner is one of the most common programming mistakes in many languages, including C.

Deallocation is accomplished by using the Std Library function `free()`:

```
int  *pA = malloc( 1000 * sizeof(int) );
. . .  // do stuff with the array
free(pA);
```

`free()` does not reset the value of the pointer on which it is invoked!

C `free()`

It's important to understand just what `free()` does (and does not do).

First, `free()` can only be applied to a pointer storing the address of a target that was allocated by calling `malloc()` and friends.

Second, `free()` can only be applied to a pointer whose a target that has not already been deallocated.

Third, when `free()` is invoked on a pointer, the pointer is not automatically reset to `NULL`.

Fourth, `free()` causes the deallocation of the target of the pointer, not the deallocation of the pointer itself. You don't free a pointer, you free its target.

Array Names are const Pointers

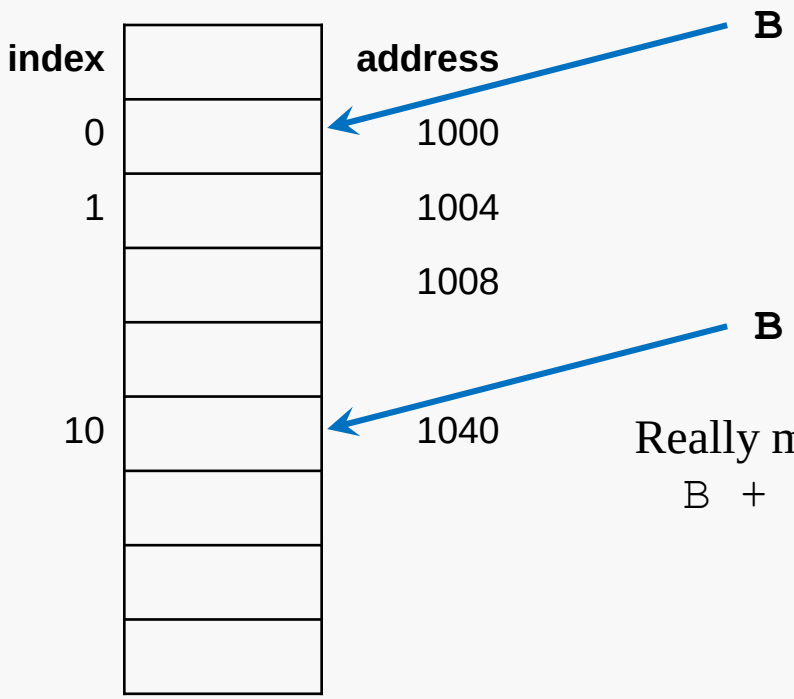
So, what's the difference between an array name (static allocation) and a pointer?

```
int N;
. . .   // assume N is assigned a value
int  *A = malloc( 1000 * sizeof(int) );   // allocate array
int   B[1000];                            // declare array

for (int pos = 0; pos < N; pos++) {
   A[pos] = pos * pos;          // access using ptr name
   B[pos] = pos * pos;          // access using array name
}

free( A );     // OK; deallocates array A
A = NULL;      // OK
free( B );     // NO!  cannot deallocate static memory
B = NULL;      // NO!  array name is const pointer
```

Array Notation vs Pointers

```
int A[1000];
. . .
int x = A[10];
```

```
int *B = malloc(1000*sizeof(int));
. . .
int x = *(B + 10);
```



Really means:

```
B + 10 * sizeof(int)
```

Arithmetic on Pointers

So, what's the effect of:

```
int A[1000];              // allocate array; static doesn't matter

int *p = A;               // p points to A[0]

p = p + 100;              // where does p point now??    A[100]

p = p - 50;               // now?                        A[50]

p++;                      // now?                        A[51]

p--;                      // now?                        A[50]
```

The effect of adding a value to a pointer depends on the pointer type:

Arithmetic on Pointers

The effect of adding a value to a pointer depends on the pointer type:

When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand.

If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression.

In other words, if the expression **P** points to the *i*-th element of an array object, the expressions **(P)+N** (equivalently, **N+(P)**) and **(P)-N** (where **N** has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist.

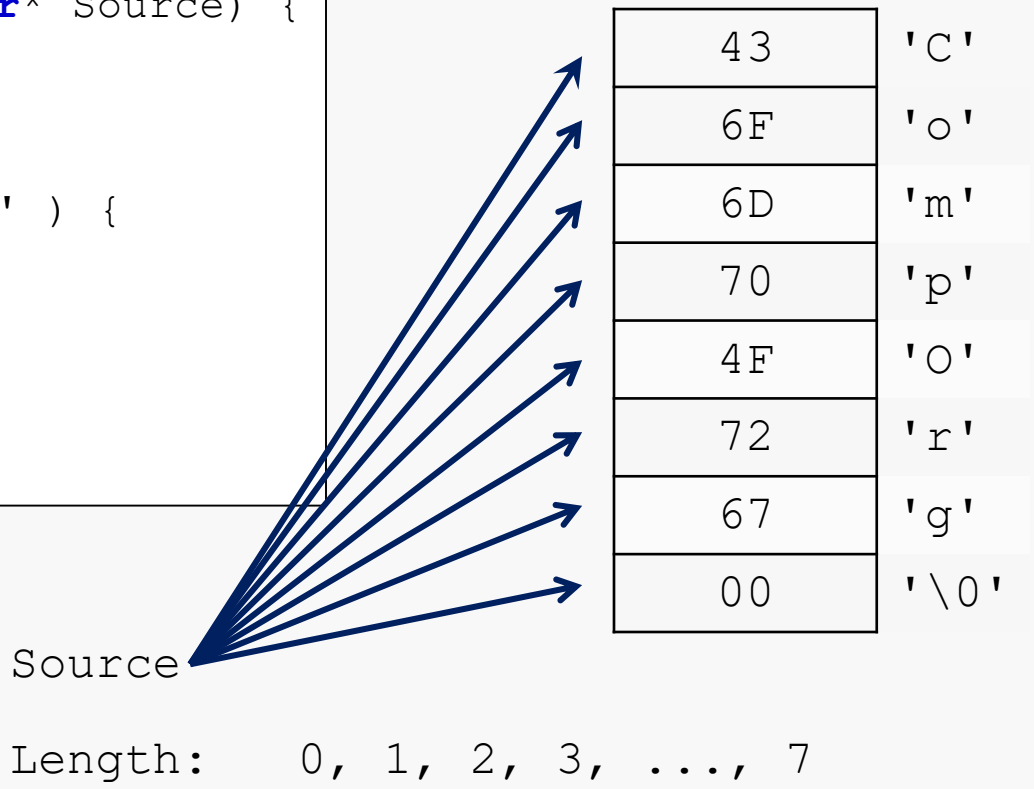
Moreover, if the expression **P** points to the last element of an array object, the expression **(P)+1** points one past the last element of the array object, and if the expression **Q** points one past the last element of an array object, the expression **(Q)-1** points to the last element of the array object.

If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary ***** operator that is evaluated.

Using Arithmetic on Pointers

Here's an implementation of a variation of the C library `strlen()` function; note the use of pointer arithmetic:

```
uint64_t strlen(const char* Source) {

   uint64_t Length = 0;

   while ( *Source != '\0' ) {
      Length++;
      Source++;
   }
   return Length;
}
```



Using Arithmetic on Pointers

And here's a slightly more idiomatic version:

```
uint64_t strlen(const char* Source) {

   uint64_t Length = 0;

   while ( *Source++ != '\0' )
      Length++;
   return Length;
}
```

Precedence note: the dereference operator (`*`) is evaluated before the increment operator (`++`)

So, we access the current target of `Source` first, then we move `Source` to the next array element.

QTP: is `*Source++` equivalent to `(*Source)++`?

… or to `*(Source++)`?

Using Arithmetic on Pointers

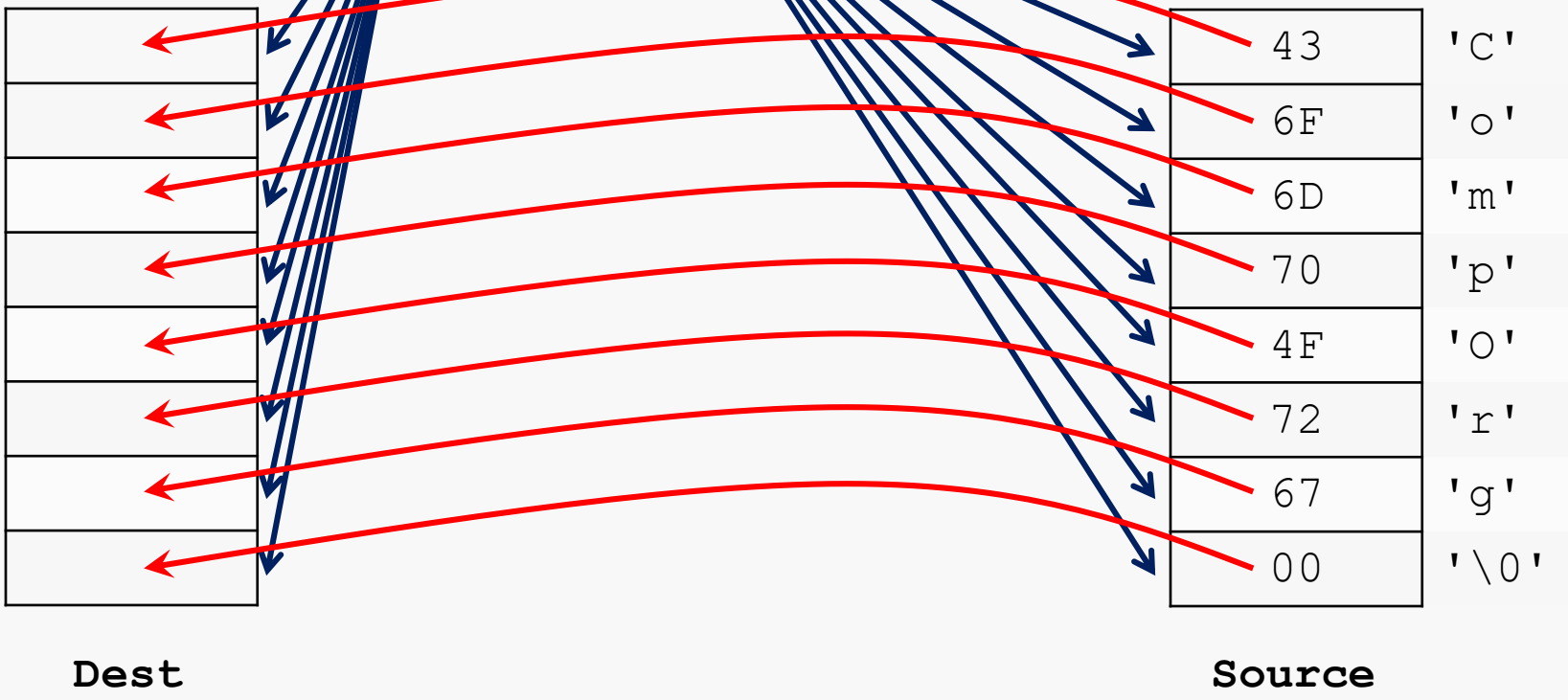
Here's an implementation of the C library `strcpy()` function:

```
char* strcpy(char* Dest, const char* Source) {

   int i = 0;
   while ( true ) {
      Dest[i] = Source[i];
      if (Dest[i] == '\0') break; // we're done
      i++;
   }
   return Dest;
}
```

Using Arithmetic on Pointers

And here's a version that uses pointer arithmetic to achieve the same effect:

```
char* strcpy(char* Dest, const char* Source) {

   while ( (*Dest++ = *Source++) != '\0' ) ;
}
```



Identity vs Equality Revisited

x equals y	x and y, in some precise sense, have the same value In C, this is equivalent to `x == y`.
x is identical to y	x and y are actually the same object In C, this is equivalent to `&x == &y`.

Side notes:

If x and y are pointers, then x equals y if and only if x and y have the same target.

In other words, two pointers are equal if and only if their targets are identical.

Memory Leaks

A *memory leak* occurs when a process allocates memory dynamically and then fails to deallocate that memory before losing access to it:

```
int *p = malloc(1000 * sizeof(int));
. . .                                  // no calls to free() here
p = malloc(1500 * sizeof(int));        // leaked original array
. . .                                  //     or here
p = NULL;                              // leaked second array
```

Memory leaks are common in badly-written C/C++ code.

Taken to extremes, they can consume all available memory on a system.

Garbage-collected memory management automates the process of deallocation, but can never be more efficient than well-written code that deallocates memory manually.

Memory Leaks

One issue you'll encounter in testing is that the Linux implementation of `malloc()` will zero memory when it is allocated to your process.

That hides errors rather than fixing them. For better testing, add the following include:

```
#include <malloc.h>
```

Then in `main()` add this call: `mallopt(M_PERTURB, 205);`

This will guarantee that memory that is allocated with `malloc()` or `realloc()` is NOT automatically written with zeros.

We recommend ALWAYS doing this when you're testing code that uses `malloc()` or `realloc()`.

Or… use the tool Valgrind…

The Key to Good Memory Management

The key to good memory management practice can be stated quite easily:

> In designing your system, keep careful track of exactly who (i.e., module or function) has ownership of each dynamically-allocated object.

Practicing this is not easy.

But, with careful attention to detail, and meticulous recording of design decisions about ownership in comments, it is certainly possible to eliminate memory leaks altogether.

Example

```
#define MAX_LINELENGTH 1024  // maximum length guaranteed?

// reading a line of text from a file

char* line = calloc( MAX_LINELENGTH + 1, sizeof(char) );

fgets(line, MAX_LINELENGTH + 1, fp);

// but the line is actually likely to be much shorter than
// that, so we can offer to "shrink" it

line = realloc( line, sizeof(char)*(strlen(line) + 1) );
```

Example

```
// copying a string; suppose we have line from last slide

// first, make an array of exactly the right size:
char* copy = calloc( strlen(line) + 1, sizeof(char) );

// then use strncpy() to duplicate the characters:
strncpy(copy, line, strlen(line));
```