Dynamic Memory Allocation

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics For Today

- Void pointers
- The free memory heap
- Using malloc/free
- Dynamic allocation hazards
- Copying strings
- Resizable arrays
- Memory performance comparison

Void Pointers

- You can't have a value of type void

```
void v;
```

- But you can make a pointer to void

```
void *vPtr;
```

- And that void pointer can hold any pointer value you want

```
float f = 3.14;
vPtr = &f;

long l = 123456789;
vPtr = &l;
```

You don't even need a cast.

Void Pointers

- But, you can't directly dereference a void pointer.

```
printf( "%ld\n", *vPtr );
```

- Think of a void pointer as a generic way of holding an address
  - they could be used to point to anything
  - but, to use what they point to, you need to know its type.

Using Void Pointers

- You need to convert a void * to a more specific type before you dereference.

```
long *lPtr = vPtr;
printf( "%ld\n", *lPtr );
```

You don't even need a cast.

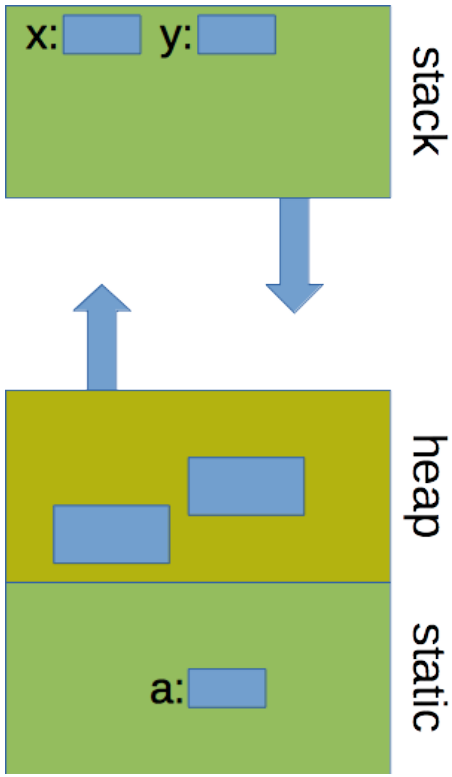
```
lPtr = (long *)vPtr;
printf( "%ld\n", *lPtr );
```

But, it's probably a good idea.

- Void pointers are used:
  - As parameter/return types for functions that can work with memory (no matter what it stores)
  - As temporary holders for pointers of various types.

Another Type of Memory

- We already have two types of memory
  - Stack space : for values with a lifetime tied to a single function execution
  - Static space : for values with a lifetime from the start to the end of the program's execution
- But, there's another type of memory, the heap
  - The program says when it needs heap memory (and how much)
  - The program says when it's done with a particular value in heap memory
  - Called: dynamic memory allocation



Asking for Heap Memory

- First, you must include stdlib.h

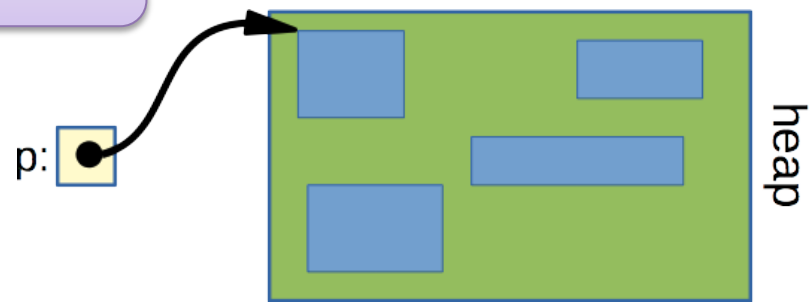
```
#include <stdlib.h>
```

- The C Standard Library has functions to let us request heap memory

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

Here's a pointer to your memory.

How many bytes do you want?



Using malloc()

- To use malloc():
  - Pass in the number of bytes you need, probably using sizeof()
  - Store the result in a pointer for the type you need.
- Allocate a 1000-element array of ints:

```
int *list = (int *) malloc( 1000 * sizeof(int) );
```

This cast is a good idea,
but a C compiler doesn't care.

- Allocate space for a copy of a string:

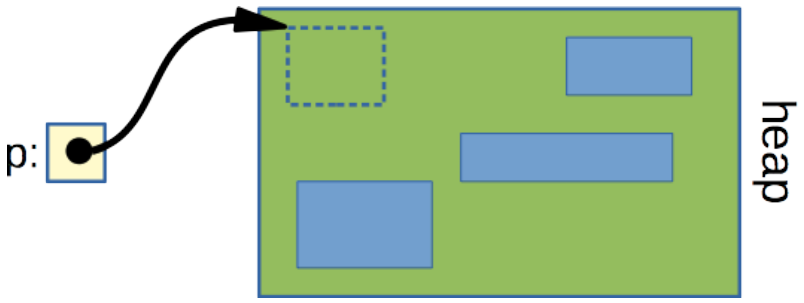
```
char *str2 = (char *) malloc( strlen( str ) + 1 );
```

Freeing Memory

- You get to keep each block of dynamically allocated memory as long as you need.
- You must tell the C standard library when you are done with a block:

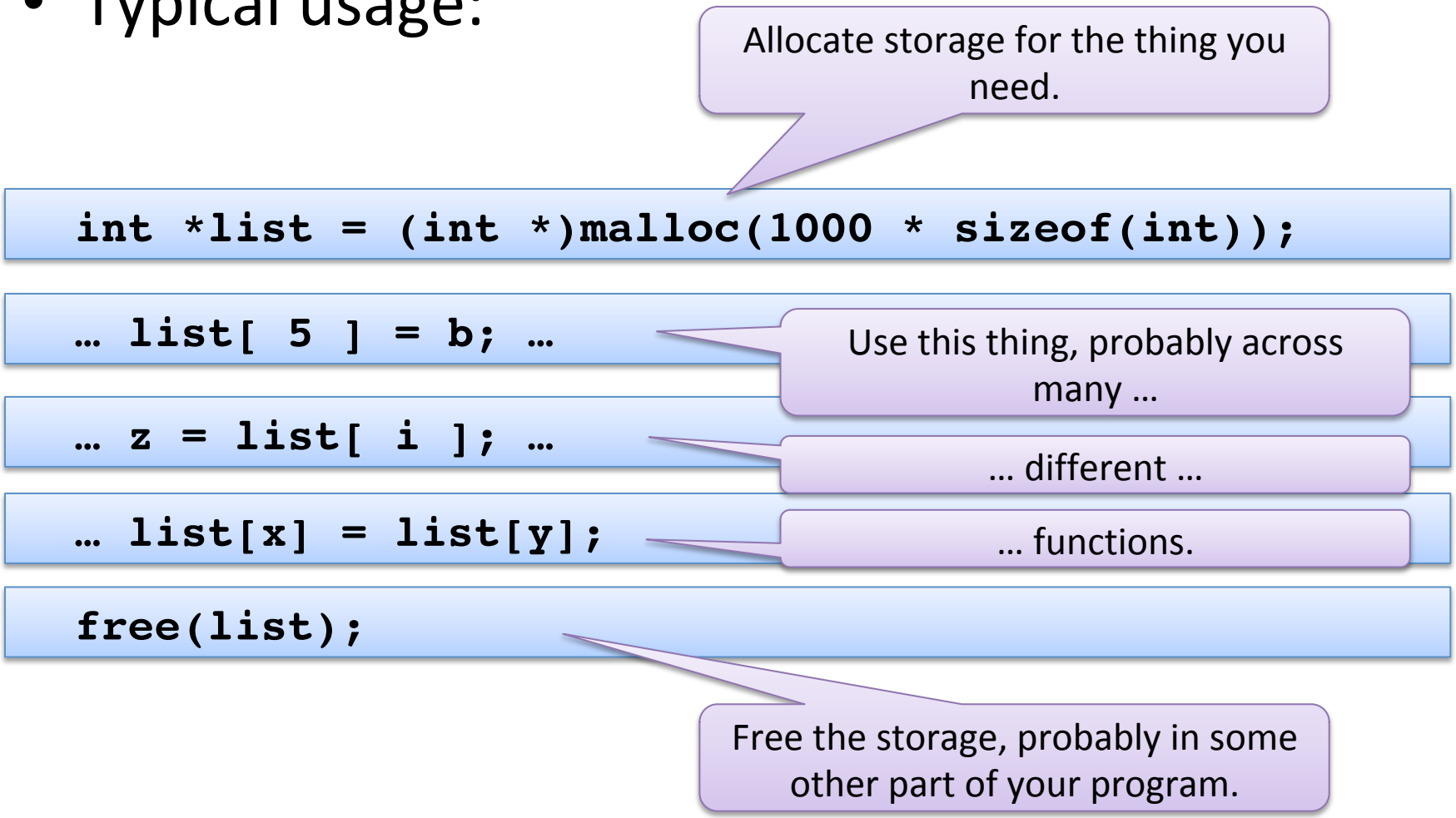
```
void free( void *ptr );
```

Pointer to previously allocated memory.



Using Malloc/Free

- Typical usage:



A Trivial Example

- A 20-element array

```
double *list = (double *)malloc(20 * sizeof(double));

 for ( int i = 0; i < 20; i++ )
   list[ i ] = i * 1.5;

 for ( int i = 0; i < 20; i++ )
   printf( "%d : %.2f\n", i, list[ i ] );

 free( list );
```

Get a 20-element array of doubles.

Use it.

Free it.

A Better Example

- Read and return a list of integers.

Here's how we return the array.

Here's how we return the length.

```
int *readList( int *len )
{
  scanf( "%d", len );

  int *list = (int *)malloc( *len * sizeof( int ) );

  for ( int i = 0; i < *len; i++ )
    scanf( "%d", list + i );

  return list;
}
```

A Better Example

- Code to use readList()

```
int len;
int *list = readList( &len );

for ( int i = 0; i < len; i++ )
  printf( "%d\n", list[ i ] );

free( list );
```

Get the list.

Use it.

Free it when we're done.

Using Dynamically Allocated Memory

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

```
void free( void *ptr );
```

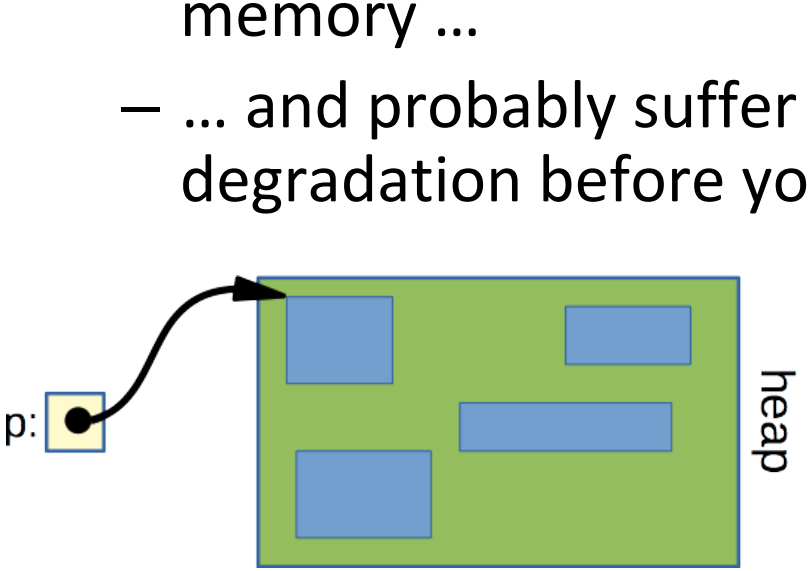
- Memory returned by malloc() is uninitialized, it may contain garbage from a previous allocation
- Returns NULL on failure (e.g., if memory is full)
- Type size_t for representing the size of storage regions
  - It's what sizeof() evaluates to
  - And what printf() prints with format string "%zd"
  - Just another name for an integer type (unsigned long on the common platform)

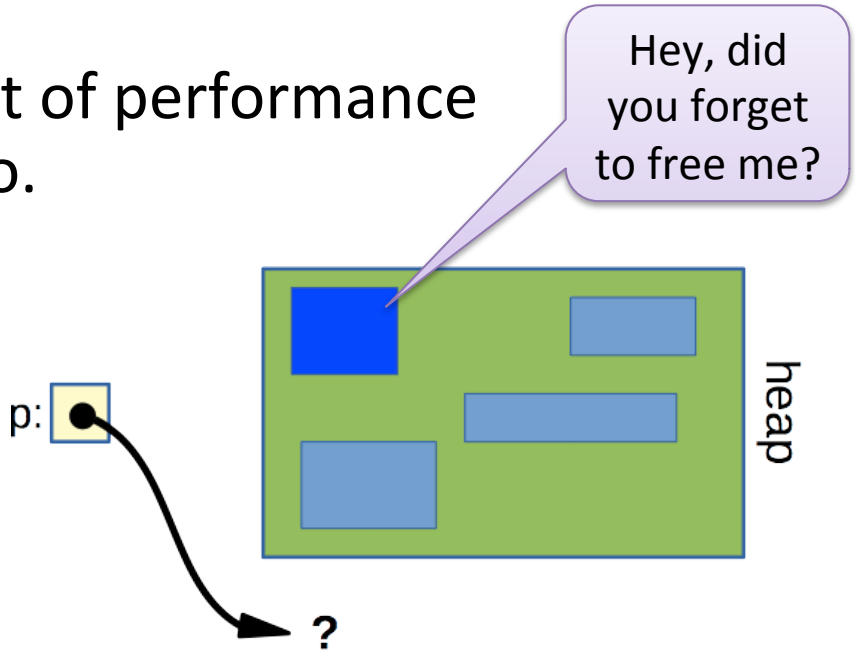
Dynamic Allocation Hazards

- All previous warnings apply
  - Buffer overflow, we can accidentally read/write off the end of a dynamically allocated array.
  - Dereferencing a NULL or uninitialized pointer
- But, now there are some new ways to make mistakes.

Leaking Memory

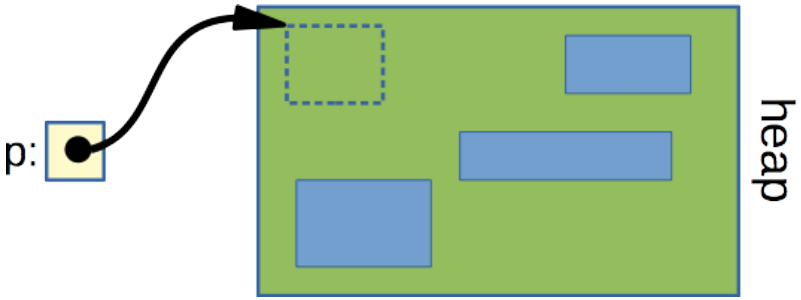
- What if you forget to free memory when you're done with it?
  - That's called a *memory leak*.
  - If you keep doing this, you'll eventually run out of memory …
  - … and probably suffer a lot of performance degradation before you do.

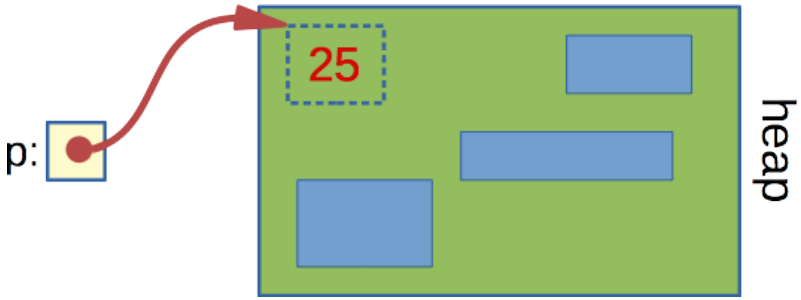




Dangling Pointers

- Freeing memory doesn't make your pointer variable go away (or set it to NULL)
- So, there's a possibility of using a pointer to memory you've already freed.
  - This is called an *dangling pointer*
  - If you're lucky, using a dangling pointer will crash your program right away.
  - If you're unlucky, … well maybe much later.





Using Dynamically Allocated Memory

- Other, less common mistakes:
  - Free memory you didn't malloc.

```
int *p = (int []) { 5, 10, 15, 20 };
…
free( p );
```

  - Free the same memory twice.

```
char *p = (char *)malloc( 1024 );
…
free( p );
…
free( p );
```

Thinking about Dynamic Allocation

- In Java, we have dynamic memory allocation; that's what `new` does.
  - In fact, we usually have to dynamically allocate anything larger than a primitive type (int, double, char, etc.)
  - But, Java performs Garbage Collection; it automatically finds memory we can no longer use and reclaims it as free space.
- Standard C doesn't perform garbage collection
- We need to tell the system when were done with each block

Thinking about Dynamic Allocation

- Why use malloc/free?
  - When you need to store something that must live longer than a single function call.
  - When you need something with size that's not known until execution.
  - When you need to store something that's too large to put on the stack.
- But, only use it where you need it.
  - Stack and static storage can be much less expensive to allocate/free.
  - And, they may have less overhead to use.
  - And, they are easier to work with.

Saving Strings

- Consider this (bad) parsing code, to read a list of strings.

```
int readWords( char *words[],
               int capacity )
{
  int count = 0;


  char str[ 100 ];
  while ( count < capacity &&
          scanf("%99s", str) == 1 ) {
    words[ count ] = str;
    count++;
  }

  return count;
}
```

Where to store the list of strings.

Keep reading strings as long as you can.

Save them in the array.

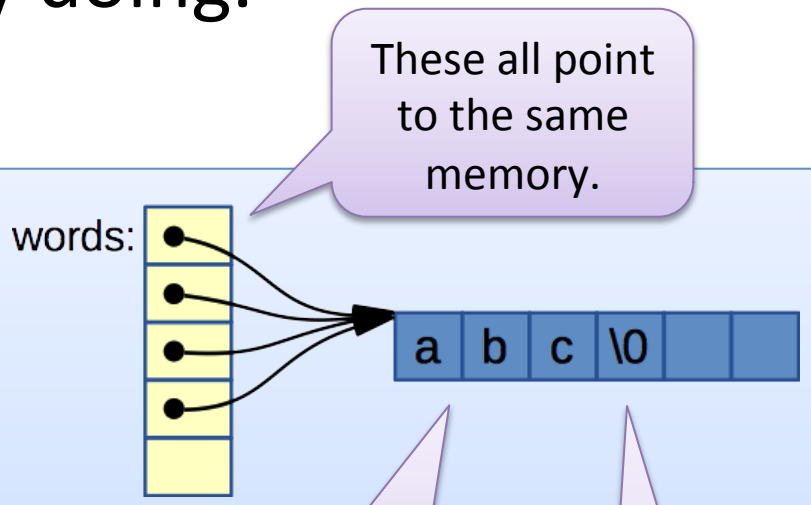
Saving Strings

- What this code is really doing.

```
int readWords( char *words[],
               int capacity )
{
  int count = 0;


  char str[ 100 ];
  while ( count < capacity &&
          scanf("%99s", str) == 1 ) {
    words[ count ] = str;
    count++;
  }

  return count;
}
```

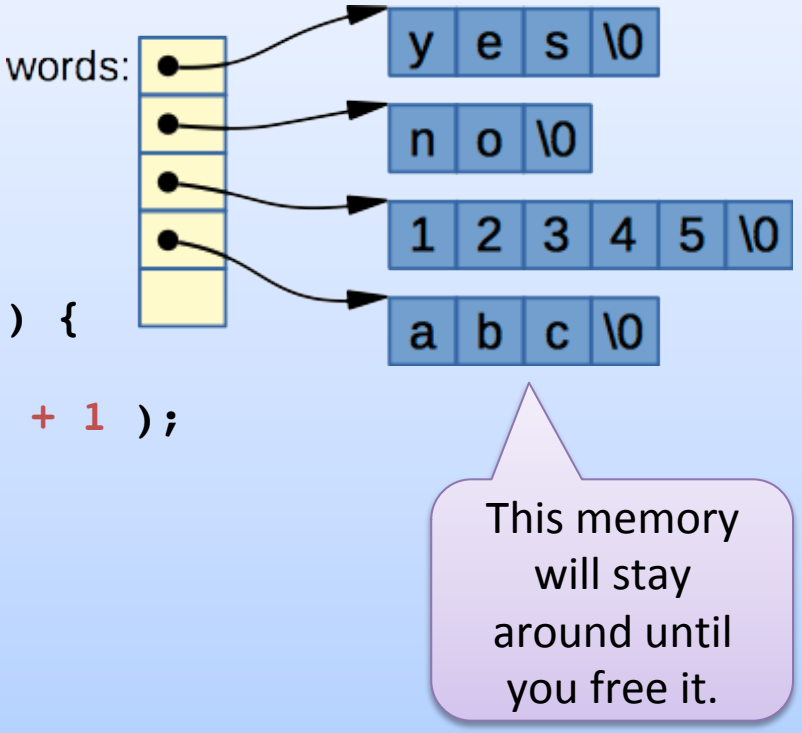


Saving Strings

- This code is better. It copies each string to its own memory.

```
int readWords( char *words[],
               int capacity )
{
  int count = 0;
  char str[ 100 ];
  while ( count < capacity &&
          scanf("%99s", str) == 1 ) {
    words[ count ] =
      (char *) malloc( strlen(str) + 1 );
    strcpy( words[ count ], str );
    count++;
  }

  return count;
}
```

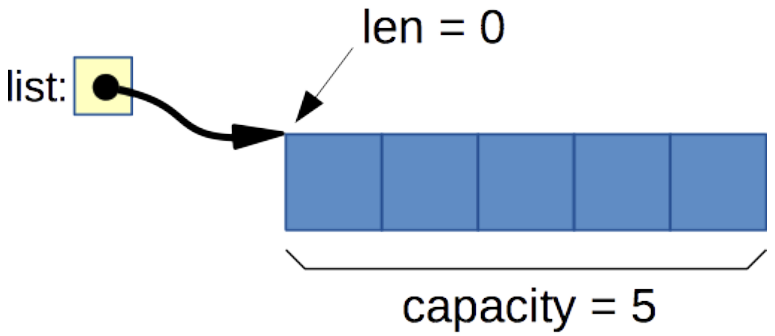


Resizing Arrays

- Our earlier example, readList(), was easy
  - We were told how many items to expect, so we could make our array exactly the right size.
- Imagine we had to read a list of values, without knowing *a priori* how many there were
  - We could use a dynamically allocated array …
  - … but we'd need to be able to enlarge it if we ran out of room.
  - This is what a resizable array does

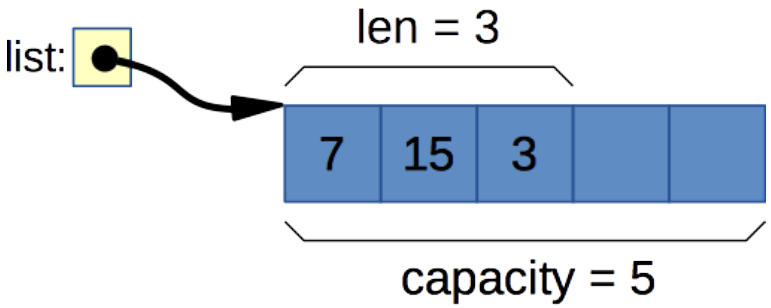
Resizing Arrays

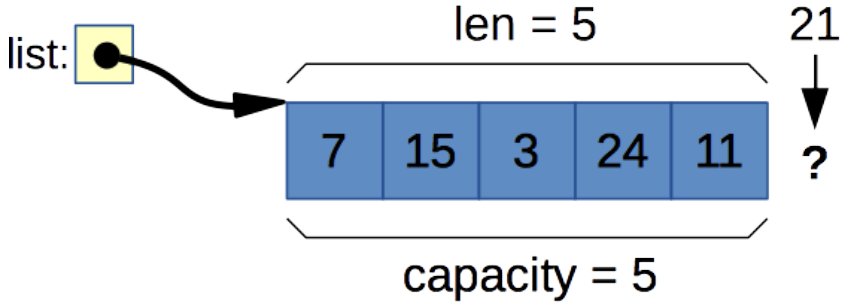
- We need to keep up with three things
  - A pointer to our array (list)
  - The number of elements used (len)
  - The capacity of the array (capacity)
- Start out with an array that's large enough … for now.



Resizing Arrays

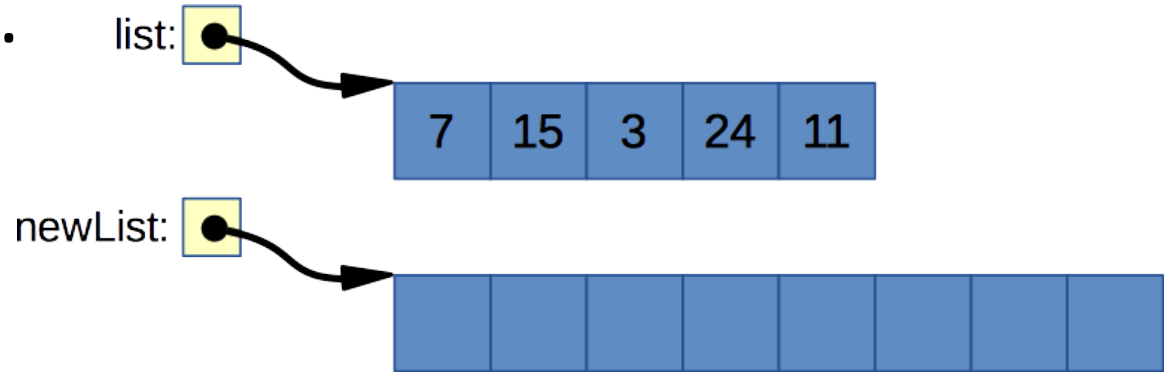
- As we append items:
  - len tells us where to put the next one.
  - increment len on each append
- Everything's easy until until we reach the capacity.



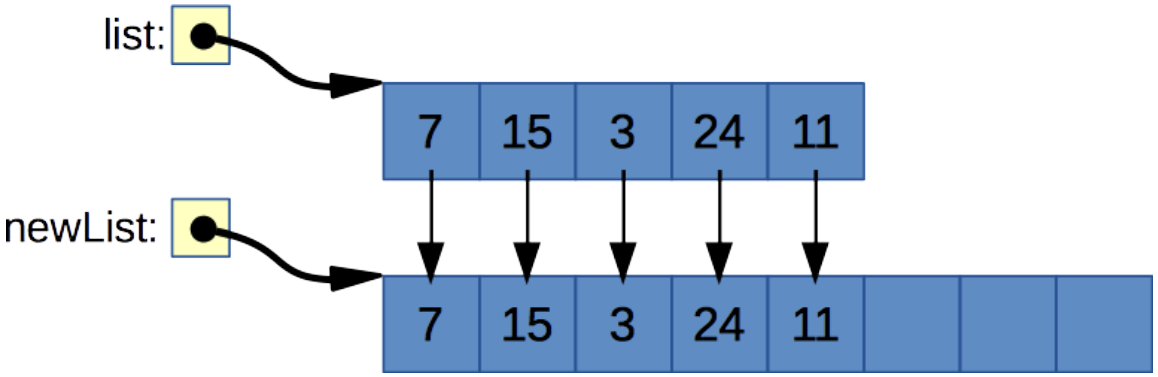


Resizing Arrays

- That's OK. We'll just allocate a new, larger array.

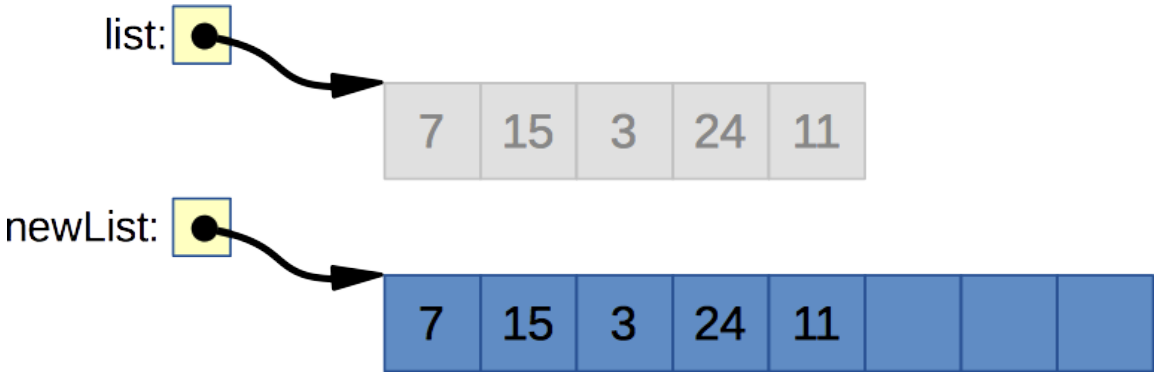


- Then copy everything over to the new array.

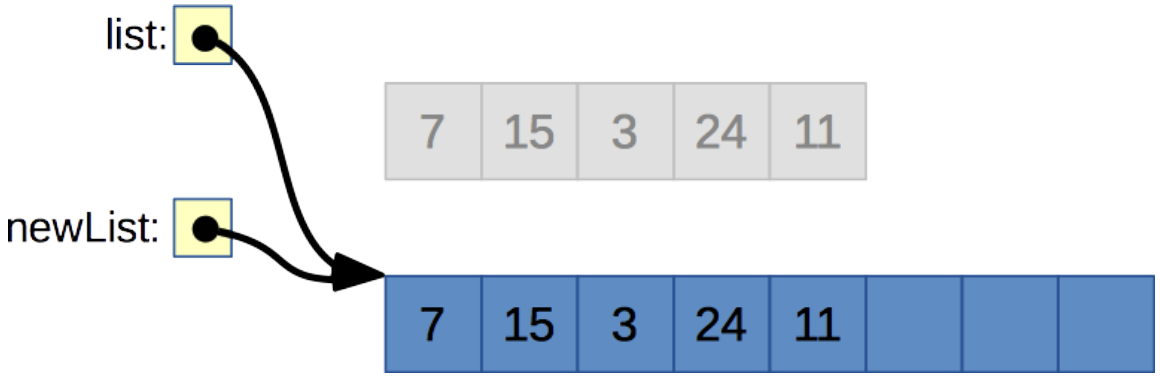


Resizing Arrays

- Then, we can free the old array.

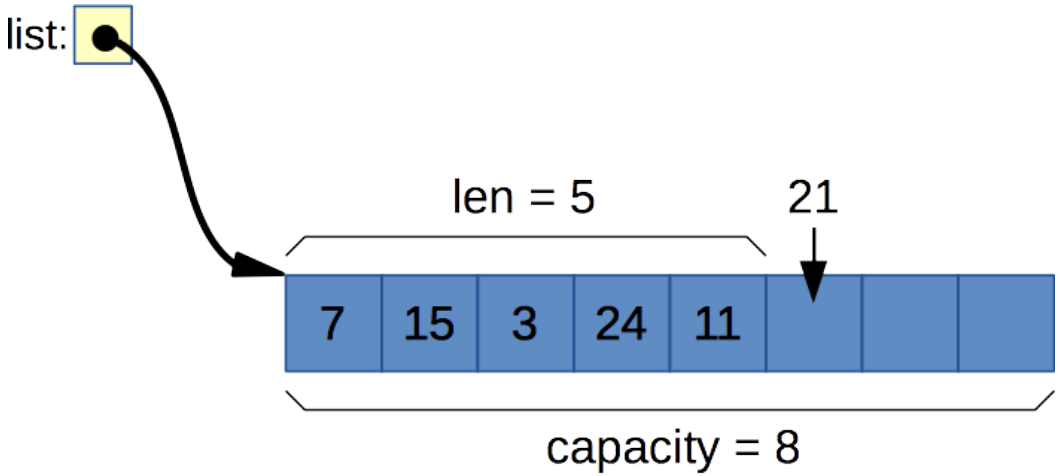


- And start using the new array instead.



Resizing Arrays

- Now, you have more room to insert new items.



- There are a few example programs that show this technique.

Resizing an Array, the Slow Way

```
int capacity = 5;
int len = 0;
int *list = (int *)malloc( capacity * sizeof( int ) );

int val;
while ( scanf( "%d", &val ) == 1 ) {
  if ( len >= capacity ) {
    capacity += 1;
    int *newList = (int *)malloc(capacity * sizeof(int));

    for ( int i = 0; i < len; i++ )
      newList[ i ] = list[ i ];

    free( list );
    list = newList;
  }

  list[ len++ ] = val;
}
```

Bad Idea, just grow a little

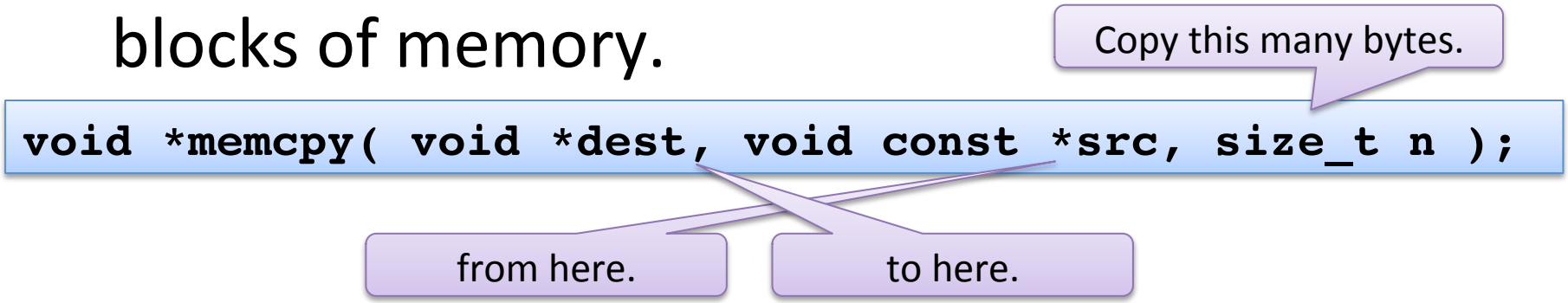
Allocate larger array

Copy everything over

Switch to the new array

Copying Memory

- The C Standard Library has functions to copy blocks of memory.



- Here, src and dest blocks can't overlap.
- But they can with memcpy's slightly more expensive friend:

```
void *memmove( void *dest, void const *src, size_t n );
```

A Better Way

```
int capacity = 5;
int len = 0;
int *list = (int *)malloc( capacity * sizeof( int ) );

int val;
while ( scanf( "%d", &val ) == 1 ) {
  if ( len >= capacity ) {
    capacity *= 2;
    int *newList = (int *)malloc(capacity * sizeof(int));

    memcpy( newList, list, len * sizeof( int ) );

    free( list );
    list = newList;
  }

  list[ len++ ] = val;
}
```

That's better.

Help from a standard function to copy everything over.

More storage overhead, but much less time overhead.

Meet realloc()

- There's a C standard library function to help.

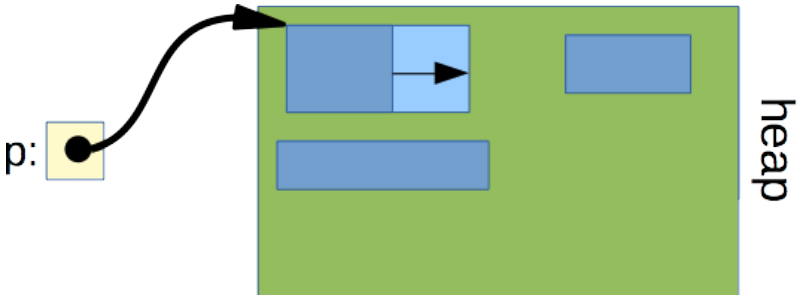
```
void *realloc(void *ptr, size_t size);
```

Pointer to larger block.

Pointer to previously allocated memory.

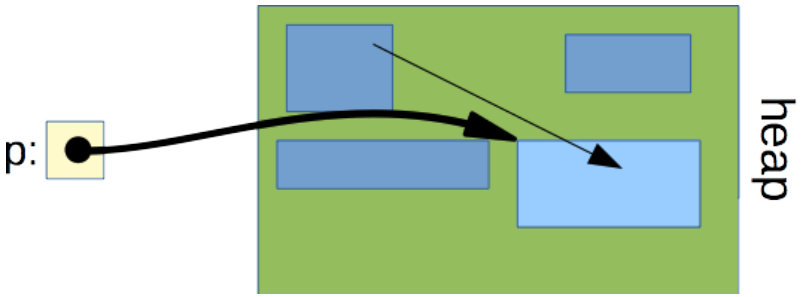
Size you need now.

- realloc() will enlarge your block and give you back the same pointer … if it can.



Meet realloc()

- If it can't enlarge your block, realloc() will:
  - find a larger block
  - copy everything over
  - free the old block
  - return a pointer to the new block.
- This will let us simplify our resizable array code.



Using Realloc

```
int capacity = 5;
int len = 0;
int *list = (int *)malloc( capacity * sizeof( int ) );

int val;
while ( scanf( "%d", &val ) == 1 ) {
  if ( len >= capacity ) {
    capacity *= 2;
    list = (int *)realloc( list, capacity * sizeof( int ) );
  }

  list[ len++ ] = val;
}
```

Choose new capacity.

Enlarge array and start using it.

Beware, this isn't going to work so well if we actually run out of memory.

Meet calloc()

- There's one more dynamic allocation function

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

Pointer to your memory.

Number of items you need.

Size of each item.

- Good for allocating arrays, but you don't really need it for that.
- Returned memory is filled with zeros
  - So, calloc() may be a little more expensive.

Performance Comparison

- We have three storage regions we can use
  - each with different behavior
  - .. and performance characteristics
- Let's try out some different techniques and see what they cost.

Uninitialized Stack Storage

```
void f() {
  int a[ 1000 ];
  a[ 999 ] = 1;
  counter += a[ 999 ];
}

int main( void )
{
  for ( long i = 0; i < 100000000; i++ )
    f();
  return 0;
}
```

Make a big array on the stack, but don't initialize it.

Just use it a little.

Using this to modify a global variable (so the compiler doesn't optimize-out all this code)

Do this 100,000,000 times.

Runtime: 0.42 seconds

Initialized Stack Storage

```
void f() {
  int a[ 1000 ] = {};
  a[ 999 ] = 1;
  counter += a[ 999 ];
}

int main( void )
{
  for ( long i = 0; i < 100000000; i++ )
    f();
  return 0;
}
```

Same thing, but initializing the array.

Runtime: 9.22 seconds

Uninitialized stack space is cheap, but initializing it has a cost.

Uninitialized Heap Storage

```
void f() {
  int *a = malloc( 1000 * sizeof( int ) );
  a[ 999 ] = 1;
  counter += a[ 999 ];
  free( a );
}

int main( void )
{
  for ( long i = 0; i < 100000000; i++ )
    f();
  return 0;
}
```

This time, on the heap, uninitialized.

Runtime: 6.31 seconds

Allocating stack space is much cheaper than heap.

Initialized Heap Storage

```
void f() {
  int *a = calloc( 1000, sizeof( int ) );
  a[ 999 ] = 1;
  counter += a[ 999 ];
  free( a );
}

int main( void )
{
  for ( long i = 0; i < 100000000; i++ )
    f();
  return 0;
}
```

On the heap, initialized this time.

Runtime: 16.17 seconds

Initializing any memory has a cost.

Static Storage

```
int a[ 1000 ];

void f() {
  a[ 999 ] = 1;
  counter += a[ 999 ];
}

int main( void )
{
  for ( long i = 0; i < 100000000; i++ )
    f();
  return 0;
}
```

Created once at the start of execution.

Probably, this is what makes it faster.

Runtime: 0.28 seconds

You may be able to compute static addresses at compile time (depending on the OS and hardware)

Now in Java

```
public class Test {
  static long counter = 0;

  static void f() {
    int[] a = new int [ 1000 ];
    a[ 999 ] = 1;
    counter += a[ 999 ];
  }

  public static void main( String[] args ) {
    for ( long i = 0; i < 100000000; i++ )
      f();
  }
}
```

In Java, all arrays are on the heap

… and they're initialized to zero

… and they're eventually garbage collected.

Runtime: 97.1 seconds

This all has a cost.