Advanced Use of Pointers

CS2023 Winter 2004

Outcomes: Advanced use of Pointers

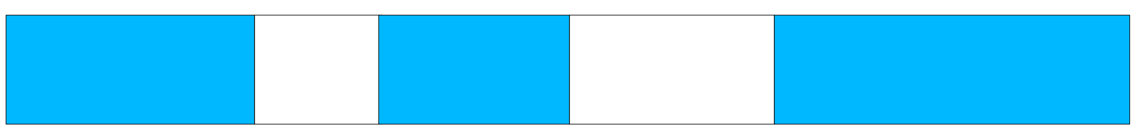
- "C for Java Programmers", Chapter 8, section 8.11, 8.15
- Other textbooks on C on reserve
- After the conclusion of this section you should be able to
  - Allocate and deallocate blocks of memory dynamically on the head
  - Use pointers to functions in order to pass a function as a parameter to another function
  - Use generic pointers to write more general functions

Dynamic Memory Allocation

- C's data structures normally fixed in size
- C supports dynamic memory allocation to allocate storage during execution
  - needed for dynamic arrays, lists, strings, ...
- Dynamically allocated memory stored on the heap

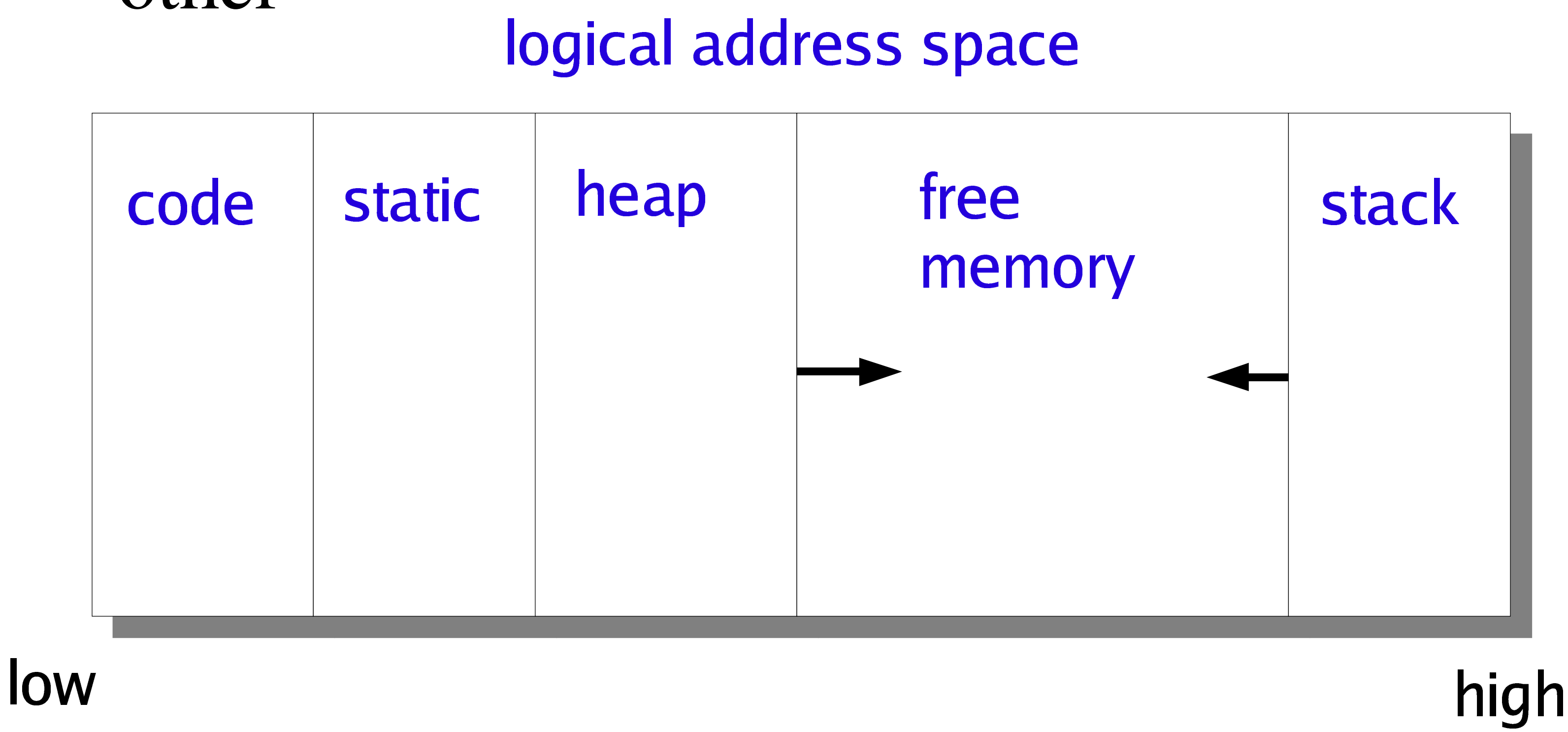
Recall: Memory Management

- Dynamically allocated variables
  - Memory allocated and destroyed at run time, under control of programmer!
  - No guarantee that first variable to be destroyed is last created
  - This area of memory can have holes and is called the **heap**



Recall: Memory Management

- Usually heap and stack begin at opposite ends of the program's memory, and grow towards each other



Dynamic Memory Allocation

`malloc` Allocates a block of memory, but doesn't initialize it

`calloc` Allocates a block of memory and clears it

- Note: arguments of **`malloc`** and **`calloc`** of type **`size_t`**, which is type returned by sizeof operator (normally **`unsigned long`**)

Dynamic Memory Allocation

Two primary methods of allocating memory:

```
void *malloc(size_t requestedSize);
void *calloc(size_t requestedCount,
             size_t requestedSize);
```

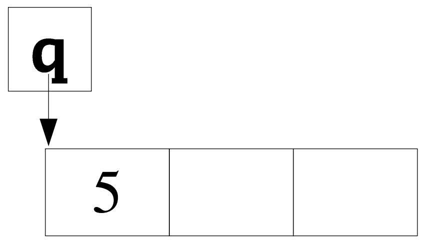
```
T *p;
p = malloc(sizeof(T));      /* or: */
p = calloc(1, sizeof(T));
```

You should always remember to check if a call to a memory allocation function was *successful*.

Dynamic Memory Allocation

```
int *p;
/* A block to store one int */
if((p = malloc(sizeof(int))) == NULL)
   exit(EXIT_FAILURE);
*p = 12;

int *q;
/* a block to store 3 ints */
if((q = malloc(3*sizeof(int))) == NULL)
   exit(EXIT_FAILURE);
*q = 5;
```



Dynamic Memory Allocation

- Note that malloc returns void*, whereas the left hand side of **`p = malloc(sizeof(int))`** is of type **`int*`**. Some programmers use an explicit cast, but this is not required:

```
p = (int*)malloc(sizeof(int))
```

- Always pass **`sizeof(type)`** as a parameter to malloc, rather than the absolute value
  - use **`malloc(sizeof(int))`** instead of **`malloc(4)`**

Memory Deallocation

Memory should be deallocated once the task it was allocated for has been completed.

```
int *p;

if((p = malloc(sizeof(int))) == NULL)
   exit(EXIT_FAILURE);
*p = 12;
...
free(p);
/* p not changed; don't use *p */
```

Memory Deallocation

Always follow the call to

```
free(p)
```

with

```
p = NULL
```

Errors

- Memory deallocation using **`free()`** should only be used if memory has been previously allocated with **`malloc()`**:

```
int i, *p;
p = &i;
free(p);
```

  - always remember where memory came from: heap or stack

- Don't create garbage objects!

```
p = malloc(sizeof(int));
p = malloc(sizeof(int));
```

  - The first object created is now inaccessible

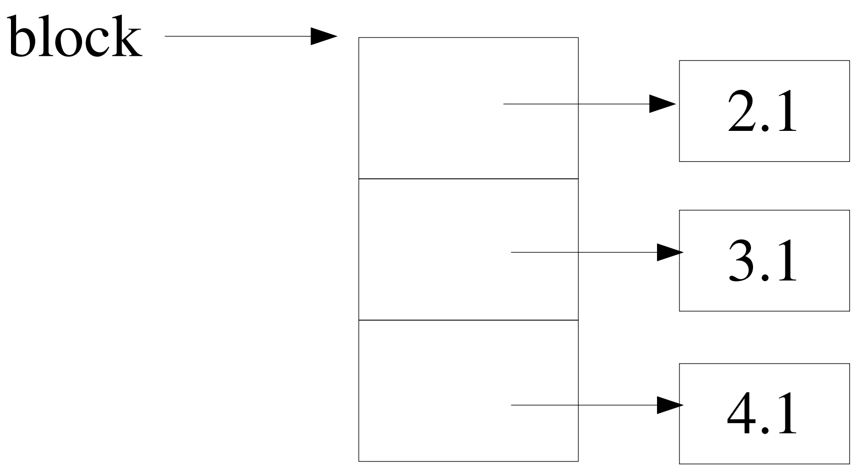
Errors

- Given two pointers `p` and `q`, the assignment `p = q` does not copy the block of memory pointed to by `q` into a block of memory pointed to by `p`

- Remember that after `p = q`; `p` and `q` share the value; if you call `free(p)` this would also deallocate `q`, now you must not call `free(q)`

Pointers to Blocks Containing Pointers

- A block containing three pointers to **`double`** objects. In order to access a single object, the code has to apply dereferencing twice



Pointers to Blocks Containing Pointers

```
double **block;
#define SIZE 3
if((block=calloc(SIZE, sizeof(double*)))
  == NULL)
      error;

for(i = 0; i < SIZE; i++)
  if((block[i]=calloc(1, sizeof(double)))
   == NULL)
      error;

*(*block) = 2.1;
block[0][0] = 2.1;
```

Pointers to Blocks Containing Pointers

The complete code to initialize the block:

```
for(i = 0; i < SIZE; i++)
    block[i][0] = 2.1 + i;
```

To free memory :

```
for(i = 0; i < SIZE; i++)
    free(block[i]);
free(block);
block = NULL;
```

```
#define SIZE 3  /* Triangular block of memory */
if((block=calloc(SIZE, sizeof(double*)))== NULL)
  error
for(i = 0; i < SIZE; i++)
 if((block[i]=calloc(i+1, sizeof(double)))==
  NULL)
    error
/* read in values */
for(i = 0; i < SIZE; i++)  /* for each row */
   for(j = 0; j <= i; j++)
     if(scanf("%lf", &block[i][j]) != 1)
        error
/* find the sum */
for(i = 0, sum = 0; i < SIZE; i++)
   for(j = 0; j <= i; j++)
      sum += block[i][j];
```

Pointers to Functions

- A pointer to a function determines the prototype of this function, but it does not specify its implementation:

```
int (*fp)(double); /* a pointer to a function */
int *fp(double); /* a function returning ... */
```

- You can assign an existing function to the pointer *as long as* both have identical parameter lists and return types:

```
int f(double); /* another function */
fp = f;
```

- You can call the function `f()` through the pointer `fp`:

```
int i = fp(2.5);
```

Functions as Parameters

```
void tabulate(double low, double high,
              double step, double (*f)(double)){
   double x;

   for(x = low; x <= high; x += step)
      printf("%13.5f %20.10f\n", x, f(x));
}
double pol1(double x) {
  return x + 2;
}

tabulate(-1.0, 1.0, 0.01, pol1);
```

Functions as Parameters

```
void tabulate(double low, double high,
              double step, double (*f)(double));
```

`f` is called:

- a *virtual* function; its implementation is not known to **`tabulate()`** but will be provided when **`tabulate()`** is called.
- a **callback** function, because it *calls back* the function supplied by the client.

Generic Search

C does not support polymorphic programming, but it can be simulated using generic pointers (i.e. **void***) .

A function prototype may specify that a block of memory and the value it is looking for are *not* typed:

```
int searchGen(const void *block,
              size_t size, void *value);
  /* non-typed values: Need more parameters: */
int searchGen(const void *block, size_t size,
  void *value, int (*compare)(const void *,
                               const void *));
```

Implementation of Generic Search (incorrect)

```
int searchGen(const void *block,
  size_t size, void *value,
  int (*compare)(const void *, const void *)) {
   void *p;

   if(block == NULL)
      return 0;
   for(p = block; p < block+size; p++)
      if(compare(p, value))
         return 1;
   return 0;
}
```

Implementation of Generic Search

```
int searchGen(const void *block,
  size_t size, void *value, size_t elSize,
  int (*compare)(const void *, const void *)) {
   void *p;

   if(block == NULL)
      return 0;
   for(p = block; p < block + size*elSize; p +=
                        elSize)
      if(compare(p, value))
         return 1;
   return 0;
}
```

Application of Generic Search

The client's responsibilities:

```
int comp(const double *x, const double *y) {
   return *x == *y;
}

int comp(const void *x, const void *y) {
   return *(double*)x == *(double*)y;
}
```

Note that this callback is sufficient for search, but not for sort.

```
/* Application of a generic search */
#define SIZE 10
double *b;
double v = 123.6;
int i;
if((b = malloc(SIZE*sizeof(double))) == NULL)
   exit(EXIT_FAILURE);
for(i = 0; i < SIZE; i++) /* initialize */
   if(scanf("%lf", &b[i]) != 1) {
        free(b);
        exit(EXIT_FAILURE);
   }
printf("%f was %s one of the values\n",
 v, searchGen(b, SIZE, &v, sizeof(double), comp)
  == 1 ? "" : "not");
```

NAME

qsort – sorts an array

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
        int(*compar)(const void *, const void *));
```

DESCRIPTION

The qsort() function sorts an array with nmemb elements of size size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.