Dynamic Memory

Chapter 13

Pointers and Addresses

The foundation for dynamic memory and structures
- `*` pointer-type specifier (variable definitions & declarations)
  - `type t;`
  - `*type p;` // same as above
  - `*t;` // same as above
- `&` address of operator returns the address of variables
- `*` dereference or indirection operator (expressions)
- Operator relationships
  - `*p` is assignment compatible with `t` (i.e., `t = *p`)
  - `&t` is assignment compatible with `p` (i.e., `p = &t`)
  - `*(&variable) = variable`
- Special pointers (that point to nothing)
  - `NULL` (null pointer)

Address Operators

Memory content vs. memory address

```
int i;
int* p = &i;

i = 10;

*p is now 10
```

```
i undefined 0x0a000010
p 0x0a000014

0x0a000010

0x0a000010

*p is undefined

i is undefined

p is 0x0a000010

*p is undefined

i is 10

*p is 0x0a000010

*p is 10
```

Pointer Syntax

Simple examples

```
int i;
int* ip1;
int* ip2;

i = 10;

ip1 = &i;  // *ip1 == 10
ip2 = ip1; // *ip2 == 10

printf("%d %d %d\n", &i, ip1, ip2); // prints addresses
printf("%d %d %d\n", i, *ip1, ip2); // prints variable values

ip1 = 0;  // *ip1 == nothing
ip2 = NULL;  // *ip2 == nothing

while (ip1 != NULL) ....
while (ip2 != 0) ....
if (ip1 == ip2) ....
if (ip1 != ip2) ....
```

Pointer Arithmetic

Addition, subtraction, and comparison
- Assumes a contiguous block of memory (i.e., an array)
- Allowed operations
  - Add/subtract an integer to/from a pointer: `p1 + i = p2`
  - Difference between two pointers: `p2 - p1 = i` (i, # of elements)
  - Test for (in)equality:
    - `if (p1 == p2)` if `(p1 != p2)`
- No check for addressing out of bounds
- The compiler automatically multiplies and divides by the size (in bytes) of the referenced object in pointer arithmetic.

Memory Layout

Data storage by class

- `heap`
- `dynamic variables`
- `parameters`
- `auto variables`
- `stack`
- `uninitialized data`
- `initialized data`
- `text`
- `low memory`
- `global and static data`
Dynamic Memory Allocation

- Used when the size or number of objects is unknown
- Allocated as a contiguous block from the heap
  - Amount of memory requested is measured in bytes
- The address of the beginning of the block is returned
  - NULL (or 0) is returned if the memory was not available
  - Must be cast to a pointer of the appropriate type
- Remains allocated until explicitly deallocated (i.e., freed)
- Becomes “garbage” if the address is lost (a memory leak)
- The C memory management system
  - #include <stdlib.h>
  - void* malloc(size_t nBytes);
  - void free(void* ptr);

Simple Dynamic Memory Examples

Allocating and deallocating

- int* ip;
- int* ap;
- char* s;

```
ip = (int*) malloc(sizeof(int));
ap = (int*) malloc(100 * sizeof(int));
s = (char*) malloc(100);
```

```
free(ip);
free(ap);
free(s);
```

Dynamic Allocation Example #1

Dynamically allocated structs

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

struct Emp {
    char name[25];
    char addr[25];
};

void main(void)
{
    struct Emp* E = (struct Emp*) malloc(sizeof(struct Emp));
    strcpy(E->name, "Albert Einstein");
    strcpy(E->addr, "Princeton, NJ");
    printf("%s %s\n", E->name, E->addr);
    free(E);
}
```

Dynamic Allocation Example #2

Dynamically allocated arrays of structs

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

struct Emp {
    char name[25];
    char addr[25];
};

void main(void)
{
    struct Emp E = (struct Emp) malloc(100 * sizeof(struct Emp));
    strcpy(E[0].name, "Albert Einstein");
    printf("%s\n", E[0].name);
    free(E);
}
```

Dynamic Allocation Example #3

Error detection

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

void main(void)
{
    int i;
    int* ip;

    if ((ip = (int*) malloc(100 * sizeof(int))) == NULL)
    {
        printf(stderr, "ERROR: out of memory\n");
        exit(1);
    }

    for (i = 0; i < 100; i++)
    {
        printf("%d: \n", ip[i]);
        for (i = 0; i < 100; i++)
            printf("%d:\n", ip[i]);
    }
}
```

Linked Data Structures

Trees and lists

- circularly linked list
- binary tree
- linked list