CS61C Course Notes

Anmol Parande

Fall 2019 - Professors Dan Garcia and Miki Lustig

Contents

Disclaimer: These notes reflect 61C when I took the course (Spring 2019). They may not accurately reflect current course content, so use at your own risk. If you find any typos, errors, etc, please report them on the GitHub repository.

1 Binary Representation

Each bit of information can either be 1 or 0. As a result, N bits can represent at most 2^N values.

1.1 Base conversions

Binary to Hex conversion

- Left pad the number with 0's to make 4 bit groups
- Convert each group its appropriate hex representation

Hex to Binary Conversion

- Expand each digit to its binary representation
- Drop any leading zeros

1.2 Numeric Representations

When binary bit patterns are used to represent numbers, there are nuances to how the resulting representations are used.

Definition 1 *Unsigned integers are decimal integers directly converted into binary. They cannot be negative.*

Definition 2 *Signed integers are decimal integers whose representation in binary contains information denoting negative numbers.*

Binary math follows the same algorithms as decimal math. However, because computers only have a finite number of bits allocated to each number, if an operation's result exceeds the number of allocated bits, the leftmost bits are lost. This is called **overflow**

1.2.1 Sign and Magnitude

In the sign and magnitude representation, the leftmost bit is the sign bit. A 1 means the number is negative where as 0 means it is positive. The downside to this is when overflow occurs, adding one does not wrap the bits properly.

1.2.2 One's Complement

To fix the bit wrapping, to form the negative number, we can flip each bit. This solves the wrapping issue so even with overflow, when adding 1, the result will continue increasing.

1.2.3 Two's Complement

To convert from decimal to two's complement:

- Invert each bit of the positive version of the number
- Add one to the result

The same process can be utilized to go from the negative version of a number to the positive version.

1.2.4 Bias Encoding

With bias encoding, the number is equal to its unsigned representation plus a bias turn. With a negative bias, we can center a positive range of $0 \to 2^N - 1$ on 0.

2 C

2.1 C Features

- C is a compiled language $\implies$ executables are rebuilt for each new system
- Every variable holds garbage until initialization
- Function parameters are pass by value

2.2 Bitwise Operators

Bitwise operators are operators which change the bits of integer-like objects (ints, chars, etc)

- &: Bitwise AND. Useful for creating masks
- |: Bitwise OR. Useful for flipping bits on
- $\wedge$: Bitwise XOR. Useful for flipping bits off
- <<: Left shifts the bits of the first operand left by the number of bits specified by the second operand.
- >>: Right shifts the bits of the first operand right by the number of bits specified by the second operand.
- : Inverts the bits.

2.3 Pointers

Definition 3 *The address of an object is its location in memory*

Definition 4 *A pointer is a variable whose value is the address of an object in memory*

2.3.1 Pointer Operators

- &: Get the address of a variable
- $*$: Get the value pointed to by a pointer

Because C is pass by value, pointers allow us to pass around objects without having them copied. They can also lead to cleaner, more compact code. However, always remember that declaring a pointer creates space for an object but **does not** put anything in that space (i.e it will be garbage).

Pointers can point to anything, even other points. For example, $int * *a$ is a pointer to an integer pointer. This is called a handle

2.4 Arrays

In C, arrays are represented as adjacent blocks of memory. The way that we interact with them is through pointers. Consider

```
int a[];
int *b;
```

Both of these variables represent arrays in C. They point to the first memory location of the array. C arrays can be indexed into using [] subscripts like other programming languages. They can also be index into using pointer arithmatic. For example, $*(a + 2) \equiv a[2]$. By adding 2 to a, C knows to look two memory locations into the array (i.e the third element).

C is able to do this because it automatically computes the size of the objects in the array to know how much to advance the pointer by.

```
int a[] = {1, 2, 3, 4, 5}; //ints are 4 bytes;
printf("%u", &a); // 0x2000
printf("%u", &(a+2)); // 0x2008
char *c = "abcdef"; //ints are 1 byte
printf("%u", &a); // 0x3000
printf("%u", &(a+2)); // 0x3002
```

One important thing to remember is that declared arrays are only allocated while the scope is valid. That means once a function returns, their memory is free to be taken.

Another important consideration in C is that C arrays do not know their own lengths and they do not check their bounds. This means whenever you are passing an array to a function, you should always be passing its size as well.

2.5 Structs

Structs are the basic datastructures in C. Like classes, they are composed of simpler data structures, but there is no inheritance.

2.5.1 Struct Operators

- =>: dereference a struct and get a subfield

typedef can be a useful command with structs because it lets us name them cleanly.

2.6 Memory Management

In C, there are four sections of memory: the stack, the heap, static, and code.

Definition 5 *The Stack is where local variables are stored. This includes parameters and return addresses. It is the "highest" level of memory and grows "downward" towards heap memory.*

Definition 6 *The Heap is where dynamic memory is stored. Data lives here until the programmer deallocates it. It sits below the stack and grows upwards to toward it.*

Definition 7 *Static storage is where global variables are stored. This storage does not change sizes and is mostly permanent.*

Definition 8 *Code storage is where the "code" is located. This includes preprocessing instructions and function calls.*

Definition 9 *Stackoverflow is when the stack grows so large that it intersects with heap memory. This is mostly unavoidable.*

Definition 10 *Heap pileup is when the heap grows so large that it starts to intersect with the stack. This is very avoidable because the programmer manages it.*

Definition 11 *All of the memory which a program uses is collectively referred to as the address space of the program.*

2.6.1 The Stack

The stack is named that way because every time a function call is made, a stack frame is created. A stack frame includes the address of the return instruction and the parameters to the function. As the function executes, local variables are added to the frame. When the function returns, the frame is popped off. In this way, frames are handled in Last-In-First-Out (LIFO) order.

Definition 12 *The stack pointer is a pointer which points to the current stack frame.*

Important: Deallocated memory is not cleared. It is merely overwritten later.

2.6.2 The Heap

The heap is a larger pool of memory than the stack and it is not in contiguous order. Back to back allocations to the heap may be very far apart.

Definition 13 *Heap fragmentation is when most of the free memory in the heap is in many small chunks*

Fragmentation is bad because if we want to allocate space for a large object, we may have enough cumulative space on the heap, but if none of the remaining contiguous spaces are open, then there is no way to create our object.

Implementation:

Every block in the heap has a header containing its size and a pointer to the next block. The free blocks of memory are stored as a circular linked list. When memory needs to be allocated to the heap, this linked list is searched. When memory is freed, adjacent empty blocks are coalesced into a single larger block. There are three different strategies which can be used to do this allocation/freeing.

Definition 14 *Best-fit allocation is where the entire linked list is searched to find the smallest block large enough to fit the requirements*

Definition 15 *First-fit allocation is where the first block that is large enough to fit the requirement is returned*

Definition 16 *Next-fit allocation is like first-fit allocation except the memory manager remembers where it left off in the free list and resumes searching from there.*

2.6.3 Heap Management

As a C programmer, it is up to us to manage the heap memory. This is done through the *malloc* function.

$$malloc(size_t size)$$

Malloc takes a size in bytes and returns a pointer to the allocated memory in the heap. If there is no space in the heap, then it returns $NULL$. Any space created with *malloc* must be freed using the $free$ function.
Things to watch out for:

- Dangling reference (A pointer used before malloc)
- Memory Leak (When $free$ isn't called or the pointer is lost but $free$ wasn't called)

- Do not free the same memory twice
- Do not use free on something not created with malloc
- malloc does not overwrite what was in the current memory location.

Because we need to tell *malloc* how much space we need allocated, we will often use the *sizeof* function. This returns the size in bytes of whatever type or object you give it. This allows us to write code for different architectures (i.e 32bit vs 64bit)