CS-200

Computer Organization and Assembly Language

C/C++ Pointers and Memory Allocation

C/C++ Tidbits

- C has a standard syntax but types may be implementation dependent
- Attempts have been made to make types explicit

  longint, int32, int64, uint, ulong, etc.
- C soon adopted linking model
  - Code compiles to .obj, .lib
  - .lib, .obj links to .com, .exe
  - References can be made to routines/data not in the current code

C/C++ Tidbits

- Many C implementations allow inline assembly
  - Finer control of code optimization
  - Hardware level interfacing – bypass the OS
  - Not platform independent!
- Headers define implementation details
  - Different libraries may use different type definitions for the same thing
  - Organization is essential
  - Even so, things quickly get complex

Computer Memory

- Memory 8-bits (byte) wide
- Organized as a numbered list
- Range depends on hardware (0 to ???)
- Addresses can be manipulated just like other integers
  - They can be added, subtracted, etc.
  - They can also be stored in memory

Computer Memory

- Are the contents at 0FFDh an address or data?
- If it is data, it can be interpreted many ways – up to the programmer
- If it is an address, it points to the data in 1003h.
- In this example, addresses are 2 bytes long.

…	
0FFDh	**10h**
0FFEh	**03h**
0FFFh	
1000h	
1001h	
1002h	
1003h	**C2h**
1004h	**FFh**
1005h	**57h**
1006h	**9Ah**
…	

Computer Memory

- Most compilers hide the details from you by substituting addresses for symbolic names

  long int mynum;

- With static naming, the compiler decides the storage ahead of time and makes the appropriate substitutions.

  mynum = 1003h to 1006h

…	
0FFDh	10h
0FFEh	03h
0FFFh	
1000h	
1001h	
1002h	
1003h	C2h
1004h	FFh
1005h	57h
1006h	9Ah
…	

Computer Memory

- If the variable is dynamic, the compiler often creates a pointer.
- The pointer is null until the variable is actually created at run-time.
- You still write the code the same; the compiler handles all substitutions.

mynum = 0FFDh -> 1003h

…	
0FFDh	10h
0FFEh	03h
0FFFh	
1000h	
1001h	
1002h	
1003h	C2h
1004h	FFh
1005h	57h
1006h	9Ah
…	

Computer Memory

- So why worry about pointers?
  - Efficiency
  - Compiler writing
  - Machine language
- Pointers are powerful but dangerous
  - Can point to random memory, even program memory
  - Can sometimes point outside allowable memory

Computer Memory

Some terms:

- Reference: address stored in memory (1003h)
- Pointer: variable containing a reference (mynum = 0FFDh)
- Referenced value: data at an address stored in a pointer

  (C2FF579Ah)

…	
0FFDh	10h
0FFEh	03h
0FFFh	
1000h	
1001h	
1002h	
1003h	C2h
1004h	FFh
1005h	57h
1006h	9Ah
…	

C/C++ Pointers and Memory Allocation

C/C++ POINTERS

C/C++ Pointers

- C language handles symbolic names for you
  - But it also allows you to specifically use pointers
  - This is why C/C++ is considered both powerful and dangerous
- Before we delve into why pointers in C/C++ are useful, let’s learn how to use them…

C/C++ Pointers

In C/C++ you must declare a variable before you can use it.

- uint mynum1; <- creates an unsigned byte integer
- uint* mynum2; <- creates a pointer to a uint
- The compiler allocates 1 byte for mynum1 and assigns the address as a substitution for mynum1
- The compiler allocates 4 bytes for mynum2 but nothing for the unsigned integer

C/C++ Pointers

- mynum1 contains a byte value.
- mynum2 contains a 2-byte address.
- At this point all the memory is empty. We say mynum2 = null.

	…	
mynum1	0FFDh	??
mynum2	0FFEh	??
V	0FFFh	??
loopcnt	1000h	??
sum	1001h	??
msg1	1002h	??
	1003h	??
	1004h	??
	1005h	??
	1006h	??
	…	

C/C++ Pointers

We can go ahead and use our variables.

- mynum1 = 2; <- puts 02h at mynum1
- mynum2 = 15h; <- puts 0015h at mynum2
  - 0015h is now a reference for mynum2
  - We'd better hope that's a good address
  - Maybe it's better to let the compiler help us

C/C++ Pointers

Method 1: use an existing variable.

- If we have a variable already allocated, we can get and use its address.

  mynum2 = &mynum1;

- The ampersand is shorthand for ‘address of’
- Now mynum2 points to the value of mynum1; that is, mynum2 contains 0FFDh.

C/C++ Pointers

Method 2: explicitly allocate memory.

- We can allocate new memory dynamically by using malloc() (new or new[] for C++).

  ```
  mynum2 = malloc(1);
  ```

- malloc or new returns the address of the new memory, which is put into mynum2.
- We passed 1 to malloc to tell it to reserve 1 byte for us but could have reserved more.
- We could also use the sizeof function.

C/C++ Pointers

Ok, so we have a pointer variable and it has an address. Now what?

- If we use the variable directly, we only get the address (reference).

newvar = mynum2; <- newvar now holds the address

- If we want the referenced value, we have to dereference the pointer.

newvar = *mynum2; <- newvar now holds the ref value

- The asterisk means ‘point to’ the address in mynum2

C/C++ Pointers

We can also assign directly to the reference value

- Remember, assigning to the pointer changes the address.

mynum2 = 25; < -Oops, we just pointed to 0025h;

- But using the asterisk does what we want.

*mynum2 = 25; <- now the reference value is 25

- Read it as ‘point to the address in mynum2 and change it to 25’

C/C++ Pointers

Now we can mention Method 3: array declarations

- Array declarations automatically allocate memory

char myarray[5]; <- allocates 5 bytes for the array

- The variable is actually a pointer to the first element of the array.
  - *myarray = 5;   // sets the first array value to 5
  - myarray[0] =5; // same thing

C/C++ Pointers

- Remember, we can do math on pointer values
- The bracketed number just adds an offset to the beginning of the array.
  - *(myarray + 3) = 5;   // sets the third array value to 5
  - myarray[3] =5;   // same thing
- The bracket notation is safer, especially if you go past the array bounds
- You can’t replace the asterisk with a [offset] unless you’ve declared an array; otherwise the compiler will complain.

C/C++ Pointers and Memory Allocation

MEMORY ALLOCATION

Memory Allocation

- Remember, allocate memory using malloc or new
- You can allocate as much as you want
- The sizeof() function will return the size, in bytes, of any type you pass it – including types you've defined
- You can do math inside the function parens:

```
Byte mystring[25];           //25 byte strings
 malloc(40 * sizeof(*mystring)); // allocate 1000 bytes
 Beware of allocating pointers, though
 malloc(40 * sizeof(mystring));  // allocate 80 bytes
```

Memory Allocation

- Variables allocated for you will also be automatically deallocated
- You must, on the other hand, deallocate anything you allocated explicitly.
- free() is the command to deallocate (delete or delete[] in C++):   free(mynum2);
- Timing is important
  - Too soon and the data isn't there when you need it
  - Too late and you could run out of space

C/C++ Pointers and Memory Allocation

POINTER USAGE

Pointer Usage

- Again, why pointers?
- Array Efficiency
  - Lookup vs. Addition vs. Inc.
  - Assembly has no built-in facility
- C/C++ pointers can point to anything
- Linked lists
- Arrays of pointers
- Function pointers

Pointer Usage

Non-homogenous arrays

- Index math can’t work
- But pointers are all same size, so array of pointers can be indexed and still point to variable objects
- (myvartype*) myarray[25]; // array of 25 pointers
- myarray[15] points to the 15th pointer
- *myarray[15] points to the 15th object

Pointer Usage

Non-homogenous arrays (cont.)

- You have to keep track of the type of data but you can allocate each pointer individually

```
myarray[12] = malloc(sizeof(mytype1));
myarray[13] = malloc(sizeof(mytype2));
```

- Caution: mixing up types could lead to trouble:

```
mytype2 *newvar = myarray[12];  // !!!
```

  - Works (pointer = pointer) but …
  - The types pointed to are different and even if size is the same, data fields may not match

C/C++ Pointers and Memory Allocation

Structs

Structs

- Structs are mentioned here because they introduce a potentially confusing pointer notation.
- Structs are generally an aggregation of different data types into a single *record*.

```
struct student{
        char *firstname;
        char *lastname;
        uint    age;
        float   average;
};
```

Structs

- Now we can create a pointer to the struct type:

```
struct student *studentptr;
```

- Next we allocate the storage (note that the name strings are not allocated):

```
studentptr = malloc(sizeof(student));
```

- Now we can access fields in studentptr in 2 ways:

```
(*studentptr).age = 22;
studentptr->age = 22;
```

Structs

- Think about: studentptr->firstname = “Eric”; or (*studentptr).firstname = “Eric”;
- Nesting levels of pointers, whether in structs or normal types, can lead to confusion
- As in algebra with nested parenthesis, it’s best to simplify where possible.
- With programming, that means the use of intermediate variables:

```
char *tempfirstname = &(studentptr->firstname);
*tempfirstname = “Eric”;
```

C/C++ Pointers and Memory Allocation

END OF SECTION