CPSC 213

Introduction to Computer Systems

Unit 1f

C, Pointers, and Dynamic Allocation

Reading

- Textbook
  - *New to C, Understanding Pointers, The malloc and free Functions, Why Dynamic Memory Allocation*
  - 2ed: "New to C" sidebar of 3.4, 3.10, 9.9.1-9.9.2
  - 1ed: "New to C" sidebar of 3.4, 3.11,10.9.1-10.9.2

C vs. Java

Java Hello World...

```
import java.io.*;
public class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello world");
  }
}
```

C Hello World...

```
#include <stdio.h>
main() {
    printf("Hello world\n");
}
```


Java Syntax... vs. C Syntax

Java Syntax...

- source files
  - .java is source file
- including packages in source
  - import java.io.*
- printing
  - System.out.println("blah blah");
- compile and run
  - javac foo.java
  - java foo
  - at command line (Linux, Windows, Mac)
- edit, compile, run, debug (IDE)
  - Eclipse

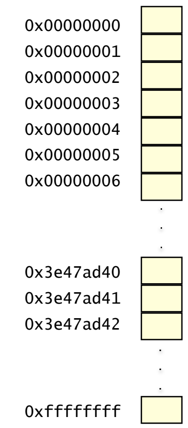
vs. C Syntax

- source files
  - .c is source file
  - .h is header file
- including headers in source
  - #include <stdio.h>
- printing
  - printf("blah blah\n");
- compile and run
  - gcc -g -o foo foo.c
  - ./foo
  - at Unix command line shell prompt (Linux, Mac Terminal, Sparc, Cygwin on Windows)
- debug
  - gdb foo

Pointers in C

New in C: Pointers

- pointers: addresses in memory
  - locations are first-class citizens in C
  - can go back and forth between location and value!
- pointer declaration: <type>*
  - int* b; // b is a POINTER to an INT
- getting address of object: &
  - int a; // a is an INT
  - int* b = &a; // b is a pointer to a
- de-referencing pointer: *
  - a = 10; // assign the value 10 to a
  - *b = 10; // assign the value 10 to a
- type casting is not typesafe
  - char a[4]; // a 4 byte array
  - *((int*) a) = 1; // treat those four bytes as an INT



C and Java Arrays and Pointers

- In both languages
  - an array is a list of items of the same type
  - array elements are named by non-negative integers start with 0
  - syntax for accessing element i of array b is b[i]
- In Java
  - variable a stores a pointer to the array
  - b[x] = 0 means *m[m[b] + x * sizeof(array-element)] ← 0*
- In C
  - variable a can store a pointer to the array or the array itself
  - b[x] = 0 means *m[b + x * sizeof(array-element)] ← 0* or *m[**m[b]** + x * sizeof(array-element)] ← 0*
  - dynamic arrays are just like all other pointers
    - stored in TYPE*
    - access with either a[x] or *(a+x)

Example

- The following two C programs are identical

```
int *a;
a[4] = 5;
```

```
int *a;
*(a+4) = 5;
```

- For array access, the compiler would generate this code

```
r[0]        ← a
r[1]        ← 4
r[2]        ← 5
m[r[0]+4*r[1]] ← r[2]
```

```
ld $a, r0
ld $4, r1
ld $5, r2
st r2, (r0,r1,4)
```

  - multiplying the index 4 by 4 (size of integer) to compute the array offset
- So, what does this tell you about pointer arithmetic in C?

Example

- The following two C programs are identical

```
int *a;
a[4] = 5;
```

```
int *a;
*(a+4) = 5;
```

- For array access, the compiler would generate this code

```
r[0]        ← a
r[1]        ← 4
r[2]        ← 5
m[r[0]+4*r[1]] ← r[2]
```

```
ld $a, r0
ld $4, r1
ld $5, r2
st r2, (r0,r1,4)
```

  - multiplying the index 4 by 4 (size of integer) to compute the array offset
- So, what does this tell you about pointer arithmetic in C?

Adding X to a pointer of type Y*, adds X * sizeof(Y) to the pointer's memory-address value.

Pointer Arithmetic in C

- Its purpose
  - an alternative way to access dynamic arrays to the a[i]
- Adding or subtracting an integer *index* to a pointer
  - results in a new pointer of the same type
  - value of the pointer is offset by index times size of pointer's referent
  - for example
    - adding 3 to an int* yields a pointer value 12 larger than the original
- Subtracting two pointers of the same type
  - results in an integer
  - gives number of referent-type elements between the two pointers
  - for example
    - (& a[7]) - (& a[2])) == 5 == (a+7) - (a+2)
- other operators
  - **&** X   the address of X
  - * X   the value X points to

Question (from S3-C-pointer-math.c)

```
int *c;

void foo () {
  // ...
  c = (int *) malloc (10*sizeof(int));
  // ...
  c = &c[3];
  *c = *&c[3];
  // ...
}
```

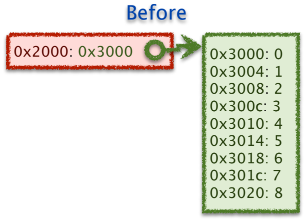
- What is the equivalent Java statement to
  - [A] c[0] = c[3];
  - [B] c[3] = c[6];
  - [C] there is no typesafe equivalent
  - [D] not valid, because you can't take the address of a static in Java

Looking more closely

```
c = &c[3];
*c = *&c[3];
```

```
r[0]    ← 0x2000          # r[0] = &c
r[1]    ← m[r[0]]         # r[1] = c
r[2]    ← 12              # r[2] = 3 * sizeof(int)
r[2]    ← r[2]+r[1]       # r[2] = c + 3
m[r[0]] ← r[2]            # c    = c + 3

r[3]    ← 3               # r[3] = 3
r[4]    ← m[r[2]+4*r[3]]  # r[4] = c[3]
m[r[2]] ← r[4]            # c[0] = c[3]
```

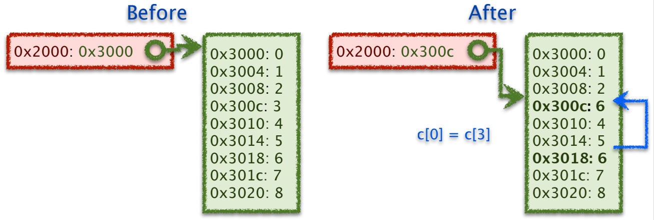
Before



Looking more closely

```
c = &c[3];
*c = *&c[3];
```

```
r[0]    ← 0x2000          # r[0] = &c
r[1]    ← m[r[0]]         # r[1] = c
r[2]    ← 12              # r[2] = 3 * sizeof(int)
r[2]    ← r[2]+r[1]       # r[2] = c + 3
m[r[0]] ← r[2]            # c    = c + 3

r[3]    ← 3               # r[3] = 3
r[4]    ← m[r[2]+4*r[3]]  # r[4] = c[3]
m[r[2]] ← r[4]            # c[0] = c[3]
```



- And in assembly language

```
r[0]    ← 0x2000          # r[0] = &c
r[1]    ← m[r[0]]         # r[1] = c
r[2]    ← 12              # r[2] = 3 * sizeof(int)
r[2]    ← r[2]+r[1]       # r[2] = c + 3
m[r[0]] ← r[2]            # c    = c + 3

r[3]    ← 3               # r[3] = 3
r[4]    ← m[r[2]+4*r[3]]  # r[4] = c[3]
m[r[2]] ← r[4]            # c[0] = c[3]
```

```
ld $0x2000, r0        # r0 = &c
ld (r0), r1           # r1 = c
ld $12, r2            # r2 = 3*sizeof(int)
add r1, r2            # r2 = c+3
st r2, (r0)           # c  = c+3

ld $3, r3             # r3   = 3
ld (r2,r3,4), r4      # r4   = c[3]
st r4, (r2)           # c[0] = c[3]
```


Example: Endianness of a Computer

```
#include <stdio.h>

int main () {
  char a[4];

  *((int*)a) = 1;

  printf("a[0]=%d a[1]=%d a[2]=%d a[3]=%d\n",a[0],a[1],a[2],a[3]);
}
```

Dynamic Allocation

Dynamic Allocation in C and Java

- Programs can allocate memory dynamically
  - allocation reserves a range of memory for a purpose
  - in Java, instances of classes are allocated by the **new** statement
  - in C, byte ranges are allocated by call to **malloc** function
- Wise management of memory requires deallocation
  - memory is a scare resource
  - deallocation frees previously allocated memory for later re-use
  - Java and C take different approaches to deallocation
- How is memory deallocated in Java?
- Deallocation in C
  - programs must explicitly deallocate memory by calling the **free** function
  - **free** frees the memory immediately, with no check to see if its still in use

Considering Explicit Delete

- Let's look at this example

```
struct MBuf * receive () {
  struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));
  ...
  return mBuf;
}

void foo () {
  struct MBuf* mb = receive ();
  bar (mb);
  free (mb);
}
```

- is it safe to free mb where it is freed?
- what bad thing can happen?

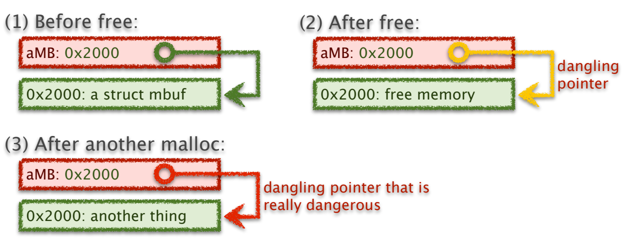
- Let's extend the example to see
  - what might happen in bar()
  - and why a subsequent call to bat() would expose a serious bug

```
struct MBuf * receive () {
  struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));
  ...
  return mBuf;
}

void foo () {
  struct MBuf* mb = receive ();
  bar (mb);
  free (mb);
}

void MBuf* aMB;

void bar (MBuf* mb) {
  aMB = mb;
}

void bat () {
  aMB->x = 0;
}
```

This statement writes to unallocated (or re-allocated) memory.

Dangling Pointers

- A dangling pointer is
  - a pointer to an object that has been freed
  - could point to unallocated memory or to another object
- Why they are a problem
  - program thinks its writing to object of type X, but isn't
  - it may be writing to an object of type Y, consider this sequence of events



Avoiding Dangling Pointers in C

- Understand the problem
  - when allocation and free appear in different places in your code
  - for example, when a procedure returns a pointer to something it allocates
- Avoid the problem cases, if possible
  - restrict dynamic allocation/free to single procedure, if possible
  - don't write procedures that return pointers, if possible
  - use local variables instead, where possible
    - since local variables are automatically allocated on call and freed on return through stack
- Engineer for memory management, if necessary
  - define rules for which procedure is responsible for deallocation, if possible
  - implement explicit reference counting if multiple potential deallocators
  - define rules for which pointers can be stored in data structures
  - use coding conventions and documentation to ensure rules are followed

Avoiding dynamic allocation

- If procedure returns value of dynamically allocated object
  - allocate that object in ***caller*** and pass pointer to it to ***callee***
  - good if caller can allocate on stack or can do both malloc / free itself

```
struct MBuf * receive () {
  struct MBuf* mBuf = (struct MBuf*) malloc (sizeof (struct MBuf));
  ...
  return mBuf;
}

void foo () {
  struct MBuf* mb = receive ();
  bar (mb);
  free (mb);
}
```

```
void receive (struct MBuf* mBuf) {
  ...
}

void foo () {
  struct MBuf mb;
  receive (&mb);
  bar (mb);
}
```


Reference Counting

- Use reference counting to track object use
  - any procedure that stores a reference increments the count
  - any procedure that discards a reference decrements the count
  - the object is freed when count goes to zero

```
struct MBuf* malloc_Mbuf () {
  struct MBuf* mb = (struct MBuf* mb) malloc (sizeof (struct MBuf));
  mb->ref_count = 1;
  return mb;
}

void keep_reference (struct MBuf* mb) {
  mb->ref_count ++;
}

void free_reference (struct MBuf* mb) {
  mb->ref_count --;
  if (mb->ref_count==0)
    free (mb);
}
```


- The example code then uses reference counting like this

```
struct MBuf * receive () {
  struct MBuf* mBuf = malloc_Mbuf ();
  ...
  return mBuf;
}

void foo () {
  struct MBuf* mb = receive ();
  bar (mb);
  free_reference (mb);
}

void MBuf* aMB = 0;

void bar (MBuf* mb) {
  if (aMB != 0)
    free_reference (aMB);
  aMB = mb;
  keep_reference (aMB);
}
```


Garbage Collection

- In Java objects are deallocated implicitly
  - the program never says free
  - the runtime system tracks every object reference
  - when an object is unreachable then it can be deallocated
  - a *garbage collector* runs periodically to deallocate unreachable objects
- Advantage compared to explicit delete
  - no dangling pointers

```
MBuf receive () {
  MBuf mBuf = new MBuf ();
  ...
  return mBuf;
}

void foo () {
  MBuf mb = receive ();
  bar (mb);
}
```


Discussion

- What are the advantages of C's explicit delete
- What are the advantages of Java's garbage collection
- Is it okay to ignore deallocation in Java programs?

Memory Management in Java

- Memory leak
  - occurs when the garbage collector fails to reclaim unneeded objects
  - memory is a scarce resource and wasting it can be a serous bug
  - its huge problem for long-running programs where the garbage accumulates
- How is it possible to create a memory leak in Java?
  - Java can only reclaim an object if it is unreachable
  - but, unreachability is only an approximation of whether an object is needed
  - an unneeded object in a hash table, for example, is never reclaimed
- The solution requires engineering
  - just as in C, you must plan for memory deallocation explicitly
  - unlike C, however, if you make a mistake, you can not create a dangling pointer
  - in Java you remove the references, Java reclaims the objects
- Further reading
  - http://java.sun.com/docs/books/performance/1st_edition/html/JPAppGC.fm.html