Hash Tables (HashMaps)

Implementing the Map interface with Hash Tables
Imagine you want to look up your neighbors’ names, based on their house number

- House numbers: 2555 through 10567 (roughly 4000 houses)
- Names: one last name per house
Array vs Tree

- You could store them in a balanced TreeMap of some kind
- Log(n) to do get, put, delete

- Or you could store them in an array
- Array is really fast lookup! O(1)
- Just look in myarray[housenumber] to get the name
Hash Table is just a modified, more flexible array

- Keys don’t have to be integers 0-(size-1)
- (Ideally) avoids big gaps like our gap from 0 to 2555 in the house numbers

- Hash function is what at makes this all work:
Hash key collisions

- Hash function takes key and maps it to an integer
- Sometimes will map two DIFFERENT keys to the same integer
  - “Collision”
- We can NOT overwrite the value the way we would if it really were the same key
- Need a way of storing multiple values in a given “place” in the hash table
Closed Addressing with Linear Probing

- Where does “Annie” go if hashkey(“Annie”) = 3?
  A. 0
  B. 1
  C. 2
  D. 3
  E. Other

<table>
<thead>
<tr>
<th>Array index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Closed Addressing with Linear Probing

Where does “Juan” go if hashkey(“Juan”) = 4?

A. 1
B. 2
C. 3
D. 4
E. Other

<table>
<thead>
<tr>
<th>Array index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Annie</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Where does “Julian” go if hashkey(“Julian”) = 3?

A. 1
B. 2
C. 3
D. 4
E. Other

<table>
<thead>
<tr>
<th>Array index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Annie</td>
</tr>
<tr>
<td>4</td>
<td>Juan</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Closed Addressing with Linear Probing

Where does “Solange” go if hashkey(“Solange”) = 5?

A. 3
B. 4
C. 5
D. 6
E. Other

<table>
<thead>
<tr>
<th>Array index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Annie</td>
</tr>
<tr>
<td>4</td>
<td>Juan</td>
</tr>
<tr>
<td>5</td>
<td>Julian</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
C for Java Programmers

Slides based on those by Philip Papadopoulos
A short history of C

• Invented in the early 1970s by Brian Kernighan and Dennis Ritchie (AT&T)
  • Affectionately known as “K&R” C (1977)
• The core of the Unix operating systems and was co-developed as Unix was being developed.
• In the mid/late 80s it became an ANSI standard language 1988
  • Book is 274 pages.
• C as a language is stable, and has highly-optimized compilers.
• It is a small
Why C, Why Learn C?

• C is sometimes thought of as a “high-level” assembly language
  • It’s structure mirrors the fundamental architecture of CPUs
  • It’s portable across different CPUs
    • for the most part, C programs are unchanged when compiled for
      • x86 (PC’s), MC68000 (Older Macs), PPC (Some IBM mainframes), ARM (what’s in your Android phone), Sparc (Sun/Oracle), PDP-11, RS600, and other CPUs

• Why learn it?
  • C is a “low level” language. It’s very flexible
  • Its close relationship to the processor (especially in memory management) enables one to understand and debug higher-level languages like Java/Python
  • If you want to drive fast, you need to understand how the engine is built – C is the engine that underpins many modern languages.
Even Simple things can be measurably faster when coded in C

• HW1 (Reverse and lines of a file using an Array and List implementation)

• Input: 200K lines random-strings.txt from your homework

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
<th>Java vs. C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>0.065s</td>
<td>2.018s</td>
<td>31x Slower</td>
</tr>
<tr>
<td>List</td>
<td>0.079s</td>
<td>1.037s</td>
<td>13X slower</td>
</tr>
</tbody>
</table>

Goal of these lectures is to learn enough C to be able to read and understand the C implementation
C and Java are related, but they are very different languages:

<table>
<thead>
<tr>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Oriented</td>
<td>Procedural</td>
</tr>
<tr>
<td>Interpreted</td>
<td>Compiled</td>
</tr>
<tr>
<td>Memory Management</td>
<td>No Memory Management</td>
</tr>
<tr>
<td></td>
<td>(User is responsible)</td>
</tr>
<tr>
<td>References</td>
<td>Pointers</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Error Codes</td>
</tr>
<tr>
<td>--</td>
<td>Pre-processor</td>
</tr>
</tbody>
</table>
Object Oriented vs. Procedural

• A Java Program is a collection of objects
  • Objects contain data and methods
  • Methods operate on their data

• A C program is a collection of procedures
  • Data used by the procedures can be internal to or external to the procedures
  • A notion of objects does not exist in C

• Both Java and C use the special method/function called `main()` to start the program
Interpreted vs. Compiled

- **Java is interpreted**
  - The java compiler converts source to *bytecode*
  - The java virtual machine (JVM) interprets bytecode
  - Once a java class is compiled, it (theoretically) does not need recompiling to run on different physical hardware (“compile once, run everywhere”)

- **C is compiled**
  - The C compiler converts source code directly to machine code
  - The machine code is executed directly by the processor
  - C programs are portable, but must be re-compiled to run on different operating systems/architecture
Memory management vs. No Memory Management

• Java performs most memory management automatically for the user
  • When more memory is required, a user creates a new object instance with the `new` keyword
  • When objects are no longer being used, the garbage collector frees memory for recycling

• C provides no memory management
  • Users create new memory with `malloc`
  • Program logic must determine when it is safe to recycle memory and then call `free`
  • When you don’t properly free no-longer needed memory, you create a memory leak
References vs. Pointers

- Java
  - A reference “points to” an Object.
  - Only Objects can be referenced (primitive variables cannot be referenced)

- C
  - A pointer “points to” memory
  - what’s at the end of the pointer depends upon what is stored at that location in memory
  - One can point to primitive variables (e.g. ints), more complex data types (C structures), functions, and pointers.
Exceptions vs. Error Codes

• Java
  • When an error occurs, an exception is thrown
  • Exceptions can be caught

• C
  • When an error occurs, a function *may* return an error code (integer)
  • When an error condition goes unnoticed, the program behaves oddly or crashes later
The C preprocessor

- The C preprocessor is a text-to-text conversion process that is integral to C compilation.
- A C source file is preprocessed, then compiled
- A variety of capabilities
  - Define untyped constants (#define) or undefined (#undef)
  - Conditionally compile sections of code
    - #if ... #else ... #endif
    - #ifdef
    - #ifndef
  - Include source code from other files (#include)
  - Parameterized macros (#define)
# The C Keyword Universe

<table>
<thead>
<tr>
<th>auto</th>
<th>break</th>
<th>case</th>
<th>char</th>
<th>const</th>
<th>continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>do</td>
<td>double</td>
<td>else</td>
<td>enum</td>
<td>extern</td>
</tr>
<tr>
<td>float</td>
<td>for</td>
<td>goto</td>
<td>if</td>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td>register</td>
<td>return</td>
<td>short</td>
<td>signed</td>
<td>sizeof</td>
<td>static</td>
</tr>
<tr>
<td>struct</td>
<td>switch</td>
<td>typedef</td>
<td>union</td>
<td>unsigned</td>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
<td>while</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bold keywords** in both C and Java. `static` has very different meanings
Many Java keywords that are NOT defined in C

<table>
<thead>
<tr>
<th>abstract</th>
<th>assert</th>
<th>boolean</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>catch</td>
<td>class</td>
<td>extends</td>
<td>finally</td>
</tr>
<tr>
<td>implements</td>
<td>imports</td>
<td>instanceof</td>
<td>interface</td>
</tr>
<tr>
<td>package</td>
<td>private</td>
<td>protected</td>
<td>public</td>
</tr>
<tr>
<td>super</td>
<td>synchronized</td>
<td>this</td>
<td>throw</td>
</tr>
<tr>
<td>throws</td>
<td>transient</td>
<td>try</td>
<td></td>
</tr>
</tbody>
</table>

These concepts are not part of the C language
A first C program

$ cat first.c
#include <stdio.h>
int main(int argc, char *argv[])
{
    float x = 34.5;
    int i = 4;
    printf("x = %f, i = %d, i * x = %f\n", x, i, i * x);
}
$ cc first.c
$ ./a.out
x = 34.500000, i = 4, i * x = 138.000000
A first C program

```c
$ cat first.c
#include <stdio.h>
int main(int argc, char *argv[])
{
    float x = 34.5;
    int i = 4;
    printf("x = %f, i = %d, i * x = %f\n", x, i, i * x);
}

$ cc first.c
$ ./a.out
x = 34.500000, i = 4, i * x = 138.000000
```

What is a.out?
A. The compiled first.c program
B. A file that stores the output of running the first.c program
C. A file that stores a class named a
D. I have no idea!
#include <stdio.h>

- This is a preprocessor directive
- include the contents of the file stdio.h
  - often referred to as an “include file”
    - stdio.h, stdlib.h, string.h are common ones
- <stdio.h> -- indicates to the compiler, look in system areas for the file

- man stdio.h to see details.
int main (int argc, char *argv[])

• Declaration of the main procedure. There can only be one.
• argc – the number of arguments on the command line
• *argv[]
  • an array of strings (a string in C is an array of char)
    • each string holds an argument
    • argv[0] is the string that is the name of file being executed
  • * indicates “pointer” (more on this later)
• Not used in first.c
$ cc first.c

• cc is the compiler (often it is soft-linked to gcc, the gnu cc compiler)

• without any flags, the output file is called a.out

• There is no relationship between the name of your c source code file and the file generated
  • Note in java, the name of .java file is the name of the public class (if a public class is declared)

• can create an executable file with the –o flag
  • cc –o first first.c
  • will compile first. c and create the executable first
printf( .... )

• formatted print statement
  • Prints out the formatted string to the standard output
    • stdout
    • Same as System.out in java

stdin = System.in  (standard input)
stderr = System.err (standard error)
A first C program

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    float x = 34.5;
    int i = 4;
    printf("x = %f, i = %d, i * x = %f\n", x, i, i * x);
}
```

$ cc first.c

$ ./a.out

x = 34.500000, i = 4, i * x = 138.000000

What will print as the value of i if I change the %d in the printf statement to %f?
A. Nothing, there will be a compile error
B. 0.00000
C. 4.00000
D. 4
E. Something else
Computer Memory is a linear array addresses

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bit</td>
<td>8 bit</td>
<td>8 bit</td>
<td>8 bit</td>
<td>8 bit</td>
<td></td>
</tr>
</tbody>
</table>

byte
word (16 bits)
double word (32 bits)
quad word (64 bits)
## C vs. Java: built-in Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>16 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>16,32 or 64 bits</td>
</tr>
<tr>
<td>long</td>
<td>64 bits</td>
<td>32 or 64 bits</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>(use int)</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>(use char)</td>
</tr>
</tbody>
</table>

* This doesn’t say anything about pointers or references. That answer is more complicated in C*
sizeof

• returns, in bytes, the size of a particular type in C
  • enables one to write portable code without “hard coding” sizes for different architectures.
  • Very important when dealing with memory allocation.

• We’ll use this a great deal
Operators (e.g., & (bitwise and) vs && (logical and))

What is the value of a && b?
A. 1
B. True
C. 0
D. False
E. This is an error
Operators (e.g., & (bitwise and) vs && (logical and))

What is the value of `a & b`?
A. 1  
B. True  
C. 0  
D. False  
E. This is an error

```
$ cat second.c
#include <stdio.h>
#define T_F(e) ((e) == 0 ? "False" : "True")
int main(int argc, char *argv[])
{
    int a = 1, b = 2;
    printf("a = %d -- b = %d\n", a, b );
    printf("a & b = %s -- a && b = %s\n", 
            T_F(a & b), T_F(a && b) );
}
$ ./second
```
Operators (e.g., & vs &&)

• A large number of operators are identical between C and Java
• Difference between C and Java if statements
  • Java: expressions must evaluate to Boolean
  • C: expressions are “ints”
    • “0” is false
    • Non-zero is true

```c
#include <stdio.h>
#define T_F(e) ((e) == 0 ? "False" : "True")
int main(int argc, char *argv[])
{
    int a = 1, b = 2;
    printf("a = %d -- b = %d\n", a, b);
    printf("a & b = %d -- a && b = %d, a & b, a && b \);
    printf("a & b = %s -- a && b = %s\n", T_F(a & b), T_F(a && b) );
}
$ ./second
a = 1 -- b = 2
a & b = 0 -- a && b = 1
a & b = False -- a && b = True
second.c
```
if (a = b) { ... }

• This is an often-used construction, but care is warranted
  • a is assigned the value of b
  • Then, the result is checked for zero (false) or non-zero (true)

• = is not the equality operator (just like Java),
  • == is the equality operator
  • Java would only allow this assignment (without a Boolean operator) inside an if statement, if a and b were Boolean types.
Four variable scopes in C

• Automatic
  • Allocated on the stack (just like java). Uninitialized.
    • Duration: only during execution of procedure (function/subroutine)

• Global
  • Any function (no matter where defined) has visibility
    • Duration: lifetime of program

• Static extern (or file global)
  • Only available to functions in the file where variable is declared
    • Duration: lifetime of program

• Static (local static)
  • A variable declared local to a function. Persists between function invocations.
  • permanent storage, local to a function.
Sample code to illustrate static, automatics

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

int g = 100; /* global variable, declared outside of every function */
static int gs = 200; /* variable only available to procedures in this file */

int myfunc()
{
    char indent[10]; indent[0] = '\0';
    static int ls; /* persist between function calls */
    int a = 1, i; /* automatic variables */
    g += 10; /* use the global */
    gs += 20; /* use the global static */
    if (g <= 110) /* first time through */
        ls = 300; /* assign local static */
    for (i = 200; i < gs; i += 20) strcat(indent,"---");
    printf("myfunc: %s (a,g,gs,ls) (%d,%d,%d,%d) \n",indent, a, g, gs, ls);
    if (g == 130) return; /* base case */
    myfunc();
}

int main(int argc, char *argv[])
{
    printf("a = automatic, g=global, gs=global static, ls=local static\n");
    myfunc();
}
```

scope.c
Sample code to illustrate static, automatics

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

int g = 100;         /* global variable, declared outside of every function */
static int gs = 200; /* variable only available to procedures in this file */

int myfunc()
{
    char indent[10]; indent[0] = '\0';
    static int ls;       /* persist between function calls */
    int a = 1, i;         /* automatic variables */
    g += 10;              /* use the global */
    gs += 20;             /* use the global static */
    if (g <= 110 )        /* first time through */
        ls = 300;        /* assign local static */
    for (i = 200; i < gs; i += 20) strcat(indent,"---");
    printf("myfunc: %s (a,g,gs,ls) (%d,%d,%d,%d) \n",indent, a,g,gs,ls);
    if (g == 130) return;    /* base case */
    myfunc();
}

int main(int argc, char *argv[])
{
    printf("a = automatic, g=global, gs=global static, ls=local static\n");
    myfunc();
}
```

How many times is `myfunc()` called (i.e. how many lines will this program print?)

A. 0   B. 1   C. 2   D. 3   E. 4
Sample code to illustrate static, automatics

```c
#include <stdio.h>
#include <string.h>
int g = 100; /* global variable, declared outside of every function */
static int gs = 200; /* variable only available to procedures in this file */

int myfunc()
{
    char indent[10]; indent[0] = '\0';
    static int ls; /* persist between function calls */
    int a = 1, i; /* automatic variables */
    g += 10; /* use the global */
    gs += 20; /* use the global static */
    if (g <= 110 ) /* first time through */
        ls = 300; /* assign local static */
    for (i = 200; i < gs; i += 20) strcat(indent,"---");
    printf("myfunc: %s (a,g,gs,ls) (%d,%d,%d,%d) \n",indent, a,g,gs,ls);
    if (g == 130) return; /* base case */
    myfunc();
}

int main(int argc, char *argv[])
{
    printf("a = automatic, g=global, gs=global static, ls=local static\n");
    myfunc();
}
```

How many times is `ls` assigned a value?

A. 0  B. 1  C. 2  D. 3  E. 4
#include <stdio.h>
#include <string.h>

int g = 100;       /* global variable, declared outside of every function */
static int gs = 200; /* variable only available to procedures in this file */

int myfunc()
{
    char indent[10]; indent[0] = '\0';
    static int ls;       /* persist between function calls */
    int a = 1, i;        /* automatic variables */
    g += 10;             /* use the global */
    gs += 20;            /* use the global static */
    if (g <= 110 )       /* first time through */
        ls = 300;      /* assign local static */
    for (i = 200; i < gs; i += 20) strcat(indent,"---");
    printf("myfunc: %s (a,g,gs,ls) (%d,%d,%d,%d) \n",indent, a,g,gs,ls);
    if (g == 130) return;    /* base case */
    myfunc();
}

int main(int argc, char *argv[])
{
    printf("a = automatic, g=global, gs=global static, ls=local static
myfunc: --- (a,g,gs,ls) (1,110,220,300)
myfunc: ------ (a,g,gs,ls) (1,120,240,300)
myfunc: --------- (a,g,gs,ls) (1,130,260,300)
myfunc():
");
    myfunc();
}
Arrays

• Work similar to java
  • indexed beginning at 0
  • array[ index ], gives you the [index + 1] element in the array.
  • there is no .length to find how long an array is
    • Keep track of it yourself
• e.g., int iarray[20];
Strings in C

• Strings in C are “arrays of char”
• There is no “length” description, instead the string is null-terminated
• \rightarrow \text{to store a string of length N, the char array must be at least N+1 in length}
  • N to store the characters of the string
  • ‘\0’ - to store the null terminator

• All string manipulation is performed in a library, e.g.,
  • `strcpy` – copy strings
  • `strdup` – duplicate a string
  • `strcmp` – compare two strings (just like `compareTo()`!)
  • `strlen` – get the length of a string, counts characters until null terminator
  • ...
Defining your own data types in C

• three important keywords
  • typedef
    • Defines a new type
  • struct
    • Defines a data structure
  • union
    • not common, used when the same memory is used as a different data type

• Let’s look at struct first
struct - building block of data structures

• Similar to instance variables of java classes, e.g.,

```
struct contact
{
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
};
```

```
struct contact phonebook[100];
```
Sample code to illustrate a simple data structure (struct contact)

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

struct contact
{
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
};

int main(int argc, char *argv[])
{
    int i;
    struct contact phonebook[100];
    strcpy(phonebook[1].lastname,"Papadopoulos");
    strcpy(phonebook[0].lastname,"Alvarado");
    phonebook[0].areaCode=phonebook[1].areaCode = 858;

    printf("sizeof(struct contact): %d \n", sizeof(struct contact));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    for (i = 0; i <= 1; i++)
        printf("phonebook[%d] (lastname,areaCode) (%s,%d)\n",i,
                   phonebook[i].lastname, phonebook[i].areaCode);
    return 0;
}
```
Access structure members

• `<variable name>.<structure member>`

• This is the same as java, but only works for structure members

• There are NO methods associated with a C data structure
  • This is a fundamental difference between a C struct and a Java class
  • C doesn’t have member functions
  • (Advanced C programmers make this work with function pointers, so one can make “objects” in C).
Typedef

- C has the ability to define new data types.
  - Once declared, that data type behave as if natively defined by the language itself.
    - You can create arrays
    - you can find out how much memory the data type requires on a particular system
- Let’s slightly redo the phonebook program with a typedef declaration
Sample code to illustrate a simple data structure (struct contact)

```c
#include <stdio.h>
#include <string.h>
typedef struct
{
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
} CONTACT;

int main(int argc, char *argv[])
{
    int i;
    CONTACT phonebook[100];
    strcpy(phonebook[1].lastname, "Papadopoulos");
    strcpy(phonebook[0].lastname, "Alvarado");
    phonebook[0].areaCode = phonebook[1].areaCode = 858;

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    for (i = 0; i <= 1; i++)
        printf("phonebook[%d](lastname,areaCode) (%s,%d)\n", i,
                phonebook[i].lastname, phonebook[i].areaCode);
}
```
Three Syntax Variants

1. Anonymous structure typedef
2. Named structure typedef
3. Structure definition followed by typedef

```
1. typedef struct
   {
       char lastname[25];
       char firstname[25];
       int areaCode;
       int phoneNumber;
   } CONTACT;

2. typedef struct contact
   {
       char lastname[25];
       char firstname[25];
       int areaCode;
       int phoneNumber;
   } CONTACT;

3. struct contact
   {
       char lastname[25];
       char firstname[25];
       int areaCode;
       int phoneNumber;
   };

typedef struct contact CONTACT;
```
Assignment with structures

• If a and b are valid data types, then b is copied into a
  • This is different than java
    • Java deals (almost exclusively) with Object references

• For example, in the previous code, If you added the statement
  • Then, the entire contents (a CONTACT) in array location 1 would be copied to array location 2.

• C deals with references as a separate concept (pointers)
Function invocation/Function return

• Every subroutine is a function in C, and can return any valid datatype
  • this includes struct types

• One declares exactly what the method returns
  • Just like java

• See next slide for code the fuses several of these
Sample code that defines functions that return structs, copies structs

```c
#include <stdio.h>
#include <string.h>
#include "structure.h"

CONTACT formFill(char last[], int code)
{
    CONTACT rval;
    strcpy(rval.lastname, last);
    rval.areaCode = code;
    return rval;
}

void printbook(CONTACT book[], int nentries)
{
    int i;
    for (i = 0; i < nentries; i++)
        printf("book[%d](lastname, areaCode) (%s,%d)\n", i,
                book[i].lastname, book[i].areaCode);
}

... Continued next slide
```
Sample code that defines functions that return structs, copies structs

```c
int main(int argc, char *argv[]) {
    int i;
    CONTACT phonebook[100];
    phonebook[0] = formFill("Papadopoulos", 858);
    phonebook[1] = formFill("Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */
    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook,3);

    /* Modify entry 2 last name */
    printf("Modify entry 2\n");
    phonebook[2] = formFill("Smith", 619);
    printbook(phonebook,3);
}
```

- Call function to fill out a CONTACT, copy the returned data structure
- copy data structures, not references
- modify [2], then print again
Functions can return references (aka pointers)

• So far, we’ve explicitly shown returns of built-in types (e.g., void, int, ...) AND struct types (e.g. struct contact)
• But there are times, when one wants just a reference to a complex data structure, without a (deep) copy
  • C can do this
• Foreshadowing: if we wanted a function myFunc to return a reference to a CONTACT, one would declare
  • CONTACT * myFunc( ... )
  • The “*” means “pointer to”
• And then, only the pointer would be returned (Which is what Java does)
How are arguments passed to C?

• by VALUE!

• The compiler makes a COPY of what is passed in the argument list and places it on the stack.
  • This is what Java does with primitive types
  • This is NOT what Java does with Object types!
/* This does NOT modify the form of the caller */
void modifyForm(CONTACT form, char last[], int code)
{
    form = formFill(last, code);
}

int main(int argc, char *argv[])
{
    int i;
    CONTACT phonebook[100];
    phonebook[0] = formFill("Papadopoulos", 858);
    phonebook[1] = formFill("Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook, 3);

    /* Modify entry 2 last name */
    printf("(Improperly) Modify entry 2\n");
    modifyForm(phonebook[2], "Smith", 619);
    printbook(phonebook, 3);
}
Pointers!

• What ARE **pointers** in C?
  • A variable that contains the *memory address* of another variable

• How do you get the address of an existing variable?
  • the ‘&’ operator
  • `int i; int * ipointer = &i; /* ipointer has the memory address of i */`

• Can you get the memory address of a variable of primitive type?
  • **YES!**
  • You can even get the memory address of a variable that itself holds a memory address
    • So-called “a pointer to a pointer”

• Let’s revisit our “failed” modifyForm() method of the previous
Sample code that will modify phonebook[2] (diff output)

```
$ diff structure4.c structure5.c
18,19c18,19
< /* This does NOT modify the form of the caller */
< void modifyForm(CONTACT form, char last[], int code)
---
> /* This does modify the form of the caller */
> void modifyForm(CONTACT * form, char last[], int code)
21c21
<     form = formFill(last,code);
---
>     * form = formFill(last,code);
37,38c37,38
<     printf("(Improperly) Modify entry 2\n")
<     modifyForm(phonebook[2], "Smith", 619);
---
>     printf("(properly) Modify entry 2\n")
>     modifyForm(& (phonebook[2]), "Smith", 619);
```
Pointers and Arrays

• A pointer knows what kind of datatype it points to, e.g
  • int * intPointer;
  • double * floatPointer;
  • CONTACT * contactPointer;

• Arrays Know what kind of datatypes they hold
  • int intArray[30];
  • double doubleArray[80];
  • CONTACT phonebook[100];

• intArray (without brackets) is a pointer an int
  • Happens to be the first address of the indexed data structure (array of ints)
• Equivalences (if intPointer = &(intArray[0]))
  • *intPointer == intArray[0]
  • *(intPointer +k) == intArray[k]
  • *(++intPointer) == intArray[1]

• Equivalences (if contactPointer = &(phonebook[0]))
  • *contactPointer == phonebook[0]
  • *(contactPointer +k) == phonebook[k]
  • *(++contactPointer) == phonebook[1]

• This is called POINTER ARITHMETIC.
  • ++, --, +=, -= all index the pointer to the proper memory location based upon the size of data structure being pointed to.
Since C passes arguments by value, how do you change a variable passed to the routine?

- Pass a POINTER to the function

```c
#include <stdio.h>

void square (int *iarg) {
    *iarg = (*iarg) * (*iarg);
}

main() {
    int i = 10;
    square(&i);
    printf("%d\n", i);
}
```

- *iarg: dereference the pointer (get its actual contents)
- *iarg: dereference the pointer (store in its memory)
- &i: address of i (pointer to i)
- prints "100"
decoding \texttt{int main(int argc, char *argv[])}

• argc – an integer describing the number of command line arguments passed to main by the command shell.

• char *argv[]
  • This takes some explaining
  • argv[] == char *argv \leftarrow \text{Pointer to characters (an array)}
  • *argv[] == char * (*argv) = char **argv
    • pointer to an array of character arrays
    • Or in “Java-nese” an array of Strings.

• Commonly, you will see
  • \texttt{int main (int argc, char **argv)}
Printing command-line arguments

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d]:'%s'
", i, argv[i]);
}
```

$ cc -o printArgs printArgs.c
$ ./printArgs first "second" "third in quotes"

argv[0]:'./printArgs'
argv[1]:'first'
argv[2]:'second'
argv[3]:'third in quotes'

note that argv[0] is the name of the program itself (not the first command argument as in java)
The Generic Pointer (void *)

• Pointers know what data type the point to
  • This makes pointer arithmetic work well.

• What do you do if you don’t know exactly what the data type is, but you still want a reference to it?

• void *
  • pointer to a void data type

• Why would you want this?
  • Think of (void *) as the equivalent of (Object)
  • Every pointer can be safely cast to/from a (void *)
The (dreaded) Null Pointer

• void * ptr = NULL;
  • Set a pointer to all zeros
  • This is the first memory location and is RESERVED

• In If tests
  • if (ptr != NULL) is often shortened to if (ptr)
The most awful “Segmentation Fault”

• What is a segmentation fault?
• Memory (in Unix/Linux) is protected.
  • One process cannot access/change another processes memory
  • Memory is said to be “segmented” so that each process has its own protected area
• Nothing in C enforces what you can put into a pointer, but
  • If it points off to memory that is outside of your processes memory segment you generate a “Segmentation” fault
  • This is a catastrophic error
The (dreaded) Null Pointer and other errors

• A null point
Syntactic Sugar: the “->” operator

• Having a pointer to a data structure is extremely common in C
• One can access members of the data structure in two different ways (supposing a CONTACT * contactPointer)
  • (*contactPointer).lastname
  OR
  • contactPointer -> lastname

  “lastname member of the structure that contactPointer points to”

• The second form is MUCH preferred for readability.
Pointers to functions: Advanced C

• The syntax for a pointer to function (e.g. a function that returns an int and has a double as an argument
  • int (*funcPtr) (float);

• Let’s pick apart some code to see all this in action
  • pointer arithmetic
  • pointers and arrays
  • pointers to functions
/* Phonebook structure header file */
typedef struct contact
{
    char lastname[25];
    char firstname[25];
    int areaCode;
    int phoneNumber;
    void (*print)(struct contact *); /* pointer to func that prints */
} CONTACT;

/* prototype declarations */
void formFill(CONTACT * t, char last[], int code);
void printbook(CONTACT book[], int nentries);
```c
#include <stdio.h>
#include <string.h>
#include "structure2.h"

static void printIt(CONTACT * t)
{
    printf("%s (%d)\n", t -> lastname, t -> areaCode);
}

void formFill(CONTACT * t, char last[], int code)
{
    strcpy(t -> lastname, last);
    t -> areaCode = code;
    t -> print = &printIt;
}

void printbook(CONTACT *book, int nentries)
{
    int i;
    for (i = 0; i < nentries; i++)
    {
        printf("book[%d]lastname,(areaCode):",i);
        (book -> print)(book);
        book++;
    }
}
```

“static” printing function. Private. Takes CONTACT pointer as an argument

formFill uses the -> syntax for ease of reading

records the specific print function (very similar to a class method)

Call the function as the end of the pointer

structure6.c
void modifyForm(CONTACT * form, char last[], int code) {
    formFill(form, last, code);
}

int main(int argc, char *argv[]) {
    int i;
    CONTACT phonebook[100];
    formFill(phonebook, "Papadopoulos", 858);
    formFill(phonebook + 1, "Alvarado", 858);
    phonebook[2] = phonebook[1]; /* copy entries */

    printf("sizeof(CONTACT): %d \n", sizeof(CONTACT));
    printf("sizeof(phonebook): %d \n", sizeof(phonebook));
    printbook(phonebook, 3);

    /* Modify entry 2 last name */
    printf("(properly) Modify entry 2\n");
    modifyForm(phonebook + 2, "Smith", 619);
    printbook(phonebook, 3);
}
Print out of the code in previous slides

```c
./structure6
sizeof(CONTACT):  64
sizeof(phonebook):  6400
book[0].lastname,(areaCode):Papadopoulos (858)
(properly) Modify entry 2
book[0].lastname,(areaCode):Papadopoulos (858)
```
Memory Management

• C puts the onus of dynamic memory management onto the user
  • In java, `new + garbage collector (System.gc())` handles this
  • In C, you explicitly allocate and de-allocated memory as needed

• memory allocators
  • `malloc`
    • Allocate memory from the heap. Memory is uninitialized. Program must explicitly deallocate with a call to `free`
  • `calloc`
    • Allocate memory from the heap. Memory is initialized to all zero. Program must explicitly deallocate with a call to `free`
  • `alloca`
    • Allocate memory from the stack. Memory is uninitialized. Memory is automatically freed when function returns.
**Typical Memory Organization**

Stack
- Cannot have holes in it

Memory Heap.
- Can have holes in it

---

**Memory of a Process**

- **Stack**
  - Call stack (Activation Frames) are stored here.
  - `alloca()` allocates memory here
  - Stack grows down

- **Heap**
  - `malloc()`, `calloc()` allocate memory from this part of memory
  - Heap grows up
  - `malloc()`, `calloc()` allocate memory from this part of memory
More on the heap

• Unlike the stack, the Heap can have “holes” in it
  • heap-allocated memory does not have to be “freed” in the order it was allocated.
  • Efficient heap allocators will attempt to reuse previously freed regions of memory to keep the heap from growing too large.
    • Can you think of how a heap data structure might be used for this?

• In Java, Objects are allocated from the heap with the `new` keyword
What do you get when call malloc()?

• A pointer.
  • actually, a void * pointer

• Allocating an array of 200 contacts
  • CONTACT * cptr = malloc (200 * sizeof(CONTACT));
  • (cast to a CONTACT pointer is done implicitly)

OR

• Allocating an array of 200 contacts
  • CONTACT * cptr = (CONTACT *) malloc (200 * sizeof(CONTACT));
  • *(cptr + k) accesses the k+1 element of the array
  • so does cptr[k]
Free – memory deallocation

• When your program no longer needs the memory that was allocated on the heap, you should call `free` to recycle it.
• If you don’t call free, you may exhaust memory.
• When you lose reference to allocated memory e.g.,
  ```c
  int * iptr = malloc(10*sizeof(int));
  iptr = malloc (20 * sizeof(int));
  free (iptr);
  • The reference to the first malloc’ed block of memory is lost
  • This is called a `memory leak`
  • memory is only recovered when the program exits
  ```
How not to give yourself a flat tire

- Returning a pointer to `alloca()`’ed (or other stack allocated) memory to a caller of a function
  - The memory is “freed” and will be overwritten at the next function call

- Attempting to free memory that wasn’t heap-allocated
  - You must call `free` with the memory address that was returned by `malloc/calloc`

```c
int * iptr = malloc(10*sizeof(int));
free ( iptr ); OK!
free ( iptr + 1); ❌
```
realloc()

• one can reallocate existing memory (usually to expand it).

“realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized.”
Singly Linked List in C

- Four files
  - linkedList.h - dataType definition
  - linkedList.c – implementation of singly Linked List of voids
  - reverseList.c – a version of file line reversal using a singly linked list
  - reverseArray.c – a version of file line reversal using an array
preprocessor defines

Data Structure Definitions: NODE, SLLIST

"forward" function declarations

linkedList.h – header file

/* linkedList.h – header file for simple linked list of strings */
#define OK 0
#define FAIL -1
#define TRUE 1
#define FALSE 0

/* Define Linked List Structure of Node, and List */
struct Node
{
    void *elem; // data element
    struct Node *link; // link to next node
};
typedef struct Node NODE;
struct SLList
{
    int size; // size of the list
    struct Node *head; // head of list
};
typedef struct SLList SLLIST;

NODE *createNode(void *val);
void destroyNode(NODE *node, int freeElem);
SLLIST *createList();
void destroyList(SLLIST *list, int freeElem);
int insertElem(SLLIST *list, void *elem);
int insertElemIndex(SLLIST *list, void *elem, int idx);
void *removeElem(SLLIST *list);
void *removeElemIndex(SLLIST *list, int idx);
NODE, SLLIST

• NODE is a typedef of struct Node
• SLLIST is a typedef of struct SLList
• Very similar in concept to Nodes (inner class) of List classes developed earlier in the quarter
  • Node – has a pointer to the actual data (notice use of void *)
  • Node – has a pointer to its next next (link)
  • SLList – has a size (keep track of # of nodes)
  • SLList – has a head (ptr to the head Node of the list)
**Create a node in the list**

* @param val pointer to data referenced by this node
  * val - the */

```c
NODE * createNode(void * val)
{
    NODE * rNode;
    rNode = (NODE *) malloc(sizeof(NODE));
    rNode->elem = val;
    rNode->link = (NODE *) NULL;
    return rNode;
}
```

Explicit memory allocation
LinkedList.c – Implements a Linked List

• `destroyNode( ...)`

```c
/** destroy a Node
 * @param node - the NODE to destroy
 * @param freeElem - if nonzero, free the element referenced
 */
void destroyNode(NODE * node, int freeElem)
{
    if (! node ) return;
    if (freeElem && node -> elem) free( node -> elem );
    free(node);
}
```

Optionally, free memory associated with data

Explicit memory deallocation
LinkedList.c – Implements a Linked List

• `createList(...)`

```c
/** Create a Singly Linked List
 * This uses a sentinel for the head node */
SLLIST * createList()
{
    SLLIST *rval;
    rval = (SLLIST *) malloc(sizeof(SLLIST));
    rval -> size = 0;
    rval -> head = createNode( (void *) NULL);
    return rval;
}
```

Explicit memory allocation
Sentinel head node
LinkedList.c – Implements a Linked List
• insertElemIndex( ... )

```c
int insertElemIndex(SLLIST * list, void * elem, int idx)
{
    int i;
    if (!list || !elem) return FAIL;
    NODE * new;
    NODE * head = list->head;
    for (i = 0; i < idx; i ++)
        head = head->link;
    if (!head) return FAIL;
    new = createNode(elem);
    new->link = head->link;
    head->link = new;
    list->size ++;
    return OK;
}
```
LinkedList.c – Implements a Linked List

- removeElemIndex( ... )

```c
void * removeElemIndex(SLLIST * list, int idx)
{
    int i;
    if (! list) return (void *) NULL;
    NODE * r; NODE * head = list->head;
    void * rval;
    for (i = 0; i < idx; i++)
        head = head->link;
    if (!head) return (void *) NULL;

    if (r = head->link){
        rval = r->elem;
        head->link = r->link;
        destroyNode(r, FALSE);
        list->size --;
        return rval;
    }

    return (void *) NULL;
}
```
reverseList.c – reverse print lines of a text file

• Major sections
  • Read the command line arguments
    • Fail if not the correct number
  • Open the file specified on the command line
    • Fail if file cannot be opened for reading
  • Read in file line-by-line
    • Add to list at position 0
  • Iterate through list
    • print lines
reverseList.c – uses a Linked List

• main code loop

```c
SLLIST list = createList();

while (fgets(linebuf, MAXLINE-1, rfile) != NULL) {
    linebuf[MAXLINE-1] = '\0'; // force string termination
    str = malloc(strlen(linebuf)+1);
    strcpy(str, linebuf);
    insertElem(list, str);
}

NODE * n;
n = list -> head;
while ( (n = n -> link))
    printf("%s", n->elem);
destroyList(list, TRUE);
```

- Read file, line-by-line until the end
- Allocate storage for the string just
- Insert into list (this version inserts at 0)
- Iterate through the list
- Destroy the list
How does this differ Java?

• Explicit memory allocation/deallocation
• pointers ~= references
• SLLIST implements list of generic elements
  • Not TYPED, could mix pointers to all kinds of things (that’s good and bad)
  • Closest to having a list of Objects
Compiling

• Option 1 (all at once)
  • `cc -o reverseLink linkedList.c reverseLink.c`

• Option 2 (first `linkedList`, then `reverseLink`)
  • `cc -c linkedList.c`  // creates `linkedList.o` object file
  • `cc -o reverseLink linkedList.o reverseLink.c`

• Option 3 (create object files, then link)
  • `cc -c linkedList.c`
  • `cc -c reverseLink.c`
  • `cc -o reverseList reverseList.o linkedList.o`
Running

$ ./reverseList
reverse <filename>
$ ./reverseList somefile
Could not open file: somefile
$ ./reverseList short.txt
Line 4
Line 3
Line 2
Line 1
$ cat short.txt
Line 1
Line 2
Line 3
Line 4
$
Make

• Subject of another class, but a Makefile is supplied.
• to compile and run the first example, Should be able to do
  • make first
  • ./first