CS 138 - Data Abstraction and Implementation

Kevin Carruthers

Winter 2013

Unix and the Unix Shell

Unix and C were developed in the 1970s at Bell Labs / AT&T in New Jersey by Thompson, Ritchie, Kernighan, and Pike. Later, C++ was also developed there by Stroustroup et al.

BSD came from UC-Berkeley. Free/Net/OpenBSD forked from this. NeXTSTEP is based on this, MAC OSX is based on NeXTSTEP.

Solaris/SunOS was developed by Sun Microsystems in the 1980s and is now owned by Oracle.

There exist AIX, HP-UX, SCO, and other commerical Unices.

Linux (1990s, open-source). Android and KindleOS are based on this.

BB10 is based on QNIX.

GNU/Linux

GNU utilities run on top of Unix. Some are re-implementations of Unix tools, but GNU's Not Unix. Linux is technically just the kernel, but is mostly useless without a shell, compiler, CL utilities... thus a Linux distribution contains mostly GNU free software, and should properly be termed GNU/Linux.

The Shell

sh is the Bourne Shell, the first practical, widely-used shell. Later implementations include bash, ksh, csh, tcsh.

Filesystem

Pretty much all OSs have a file system, though you can't see it in iOS. Your home directory is where your personal stuff is stored. You can reference it by a ~. For example, user 'god' has a home directory ~god.

Some commands in Unix

```
$ pwd
$ cd
$ ls (-laFt)
$ g++ -o test test.cc
$ ./test
$ echo
$ cat
$ less
$ make
$ cp/mv/rm (-iv)
$ (s)diff x y
$ find [dir] (-name) (-type f | d) (-not | -a | -o expr) patt
$ grep [-irnvEF] pattern-string file-list
$ sort (-n | -c)
```

or more generally

```
$ cmd -opt1 -opt2 arg1 arg2 arg3
```

Commands may be built into the shell, external, or aliases. The shell looks for aliases, then built-ins, then externals. External programs are found by a predefined path (/usr/bin, etc)

What does the current command do in local context?

```
$ which cmd
```

File Permissions

Each file or directory has three sets of permissions: user, group, and other. Groups are arbitrary collections of userIDs defined under admin control. A user can belong to multiple groups.

The three types of permissions are read (4), write (2), and execute (1). For directories, read means "see what files are inside", write means "add/delete files inside", execute means "cd into it".

```
$ chmod (-r) permissionsChanges nameOfFileOrFolder
```

Use ugo(a) and +/- or octal

```
$ ssh (-Y) (-1 userID) (user@)hostname
```

Globbing

- $*$ matches zero or more characters
- ? matches one character
- , matches any alternative in the set
- [] matches one character in the set
- ! not
- $-$ creates ranges. Escape it by putting it at the end or beginning of the set
- $*$ doesn't match dotfiles
- single quotes protect everything, double quotes protect everything except doublequotes, backquotes, variables.

IO Redirection

- $<$ means take input from file
- $>$ means overwrite output to file
- $>>$ means append output to file
- | means pipe the first output into the second input
- $2 >$ &1 means pipe stderr to stdout

C++

```
// hi mom!
using namespace std;
#include <string>
#include <iostream>

const string kidDrinks = "juice";
string adultDrink = "coffee";

int main(int age; char argv[]) {
  int age = 100;
  while(age > 0) {
    cout << "How old are you?";
    cin >> age;
    cout << "Would you like some ";
    if(age < 18) {
```

```
    cout << kidDrink;
  } else {
    cout << adultDrink;
  }
  cout << "?" << endl;
  if(age == 49) {
    adultDrink = "La chouffe";
  }
  return 0;
}
```

Boolean Type

Boolean types have two special cases, true and false.

```
bool done = false;
while(!done) {
  // do stuff
  if( ... ) {
    done = true;
  }
}
```

You can also use numbers as boolean values: zero means false, other values mean true.

```
if(n)
  (n == 0)
  (p = NULL)
  (NULL == p)
```

C++ Strings

Much nicer, more convenient, and safer than char*. We only use char when dealing with legacy C compatibility. Need to include string.

```
#include
int main( ... ) {
  string fruit = "apple";
  string tree = "pine";
  cout << tree + fruit << endl;
  string f = tree + fruit;
  if(f == "pineapple") {
    cout << "yup, same";
  }
  int n = f.length();
```

```
  cout << "Beyond the " << f[0] << f.at(4) << f.substr(n-2) << endl;
  cout << "Beyond the" << f[0] + f.at(4) << endl;
}
```

Types

```
cout << "n" << endl;                \\ n
        'n'                         \\ n
        (int)'n'                    \\ 109
        (string)"m" + (string)"g"   \\ mg
        (string"m" + 'g'            \\ mg
        'm' + 'g'                   \\ 212
        "m" + "g"                   \\ error
```

CLI

```
int main(int argc, char* argv[]) {
  cout << argc << endl;
  for(int i = 0; i < argc; i++) {
    cout << "arg " << i << " is" << argv[i] << endl;
  }
  string s = argv[1];
  cout << s + "hi mom" << endl;
  \\ argv[2] = s;                   \\ This is wrong and stupid
  argv[2] = (char *) s.c_str();
}
```

IO

Sample IO

```
#include <iostream>
```

gives cin, cout, cerr: three special variables you can do IO with.

IO overloads the double arrow operators.

Input and Whitespace

```
cin >> foo >> bar;
```

gives the first two tokens, ignoring all whitespace.

```
string line;
getline(cin, line);
```

gives the entire line of input, newlines are removed.

Input and EOF

```
.eof()  // true is EOF
.fail() // true if EOF
```

both can be used with other streams and neither will trigger until you go too far.

Example:

```
int main(...) {
  double sum = 0;
  int count = 0;
  while(true) {
    double next;
    cin >> next;
    if(cin.fail()) {
      break;
    }
    sum += next;
    count++;
  }
}
```

Note that we can also use

```
if(!cin)
```

Or more concisely

```
double next;
while(cin >> next) {
  sum += next;
  count++;
}
```

File IO

ifstream and ofstream are used for file IO. We can't use the file directly, so we associate it with a stream.

```
ifstream ifstr("foo.txt");
```

Stream constructors expect char* not strings.

Note that we need to immedaitely check whether stream creation was successful.

```
#include <iostream>
#include <fstream>

int main(int agrc; char* argv[]) {
  if(argc <= 1) {
    cerr << "Error" << endl;
    exit(1);
  }
  ifstream is_raw_grades(argv[1]);
  if(!is_raw_grades) {
    cerr << "Error" << endl;
    exit(2);
  }
  ofstream pass("passes");
  ofstream fail("fails");
  // should check errors here

  int grade;
  string name;
  while(is_raw_grades >> grade >> names) {
    if(grade >= 50) {
      pass << grade << " " << name << endl;
    } else {
      fail << grade << " " << name << endl;
    }
  }

  is_raw_grades.close();
  pass.close();
  fail.close();

  return 0;
}
```

Note that

```
ifstream in("input.txt");
```

is alright, but

```
string mFilename = "input.txt"
ifstream in(mFilename);
```

is not, because streams can be opened only with char* but not string. Of course, this means we can use

```
string mFilename = "input.txt"
ifstream in(mFilename.c_str());
```

There are two ways to open files:

1. Call the constructor (ifstream in(”foo”);)
2. Create stream and connect later (ofstream out; out.open(”bar”);)

Regardless, we must close all streams with out.close();

```
int main(int argc, char* argv[]) {
  ifstream in1;
  in1.open("in1.txt");
  ifstream in2("in2.txt");
  if(!in1 || !in2) {
    cerr << "ERROR" << endl;
    exit(1);
  }
  ofstream out;
  out.open("out.txt");
  if(!out) {
    cerr << "Error" << endl;
    exit(2);
  }
  string s;
  in1 >> s;
  out << s " to 1" << endl;
  out.close();
  out.open("out2.txt");
  if(!out) {
    // error stuff
  }
  in2 >> s;
  out << s << " to 2" << endl;
  out.close();
  in1.close();
  in2.close();
}
```

cin, cout, cerr

They are global variables from the C++ standard libraries, included in <iostream>. cin is an instance of the C++ class istream, cout and cerr are instances of the C++ class ostream.

Inheritance and Polymorphism

```
void log(ostream output) {
  output << "42" << endl;
}

int main (...) {
  ostream myout("foo");
  log(cerr);
  log(cout);
  log(myout);
}
```

ofstream inherits from ostream, so any aspects of ostream are also aspects of ofstream.

C++ Arrays

Almost exactly like C arrays

```
int A[15];        // OK
const int N = 20;
int B[N];         // OK
int M;
cin >> M;
int C[M];         // Not OK, you can not have variable length arrays
```

References to arrays are unchecked. We have to pass array.extent(size); to procedures. For example:

```
int findMax(int arr[], int N) {
  int max = 0;
  for(int i = 0; int < N; i++) {
    if(arr[i] > max) {
      max = arr[i];
    }
  }
  return max;
}
```

Vectors

A **vector** is a container data structure from the C++ STL. It is like a C-style array except that

- it is generic and type safe

- index bound checking is possible (optional - array.at(i) instead of array[i])
- it can be resized on the fly, either by element (push_back) or by chunk

Other STL data structures are deque, list, map, and set.

Iteration

Simple, normal, numeric approach.

```
for(int i = 0; i < v.sive(); i++)
```

Abstract, powerful, confusing. (STL iterators)

```
for(vector<string>::const_iterator i = v.begin(); i != v.end(); i++)
```

Structs and Pointers

We can create **structs** through either direct (static) instantiation or dynamic instantiation (i.e. through a pointer). For example:

```
struct coord {
  int x,y;
};

int main(...) {
  coord a;      // static
  a.x = 3;
  a.y = 5;

  coord *b = new coord;
  b->x = 3;
  b->y = 5;

  coord *c = &a;

  delete b;
  delete c;

  return;
}
```

Pointers are basically a strange number with strange arithmetic. They can point to any object, pointer, procedure, etc, or can be assigned a NULL value (pointing to nothing). You can see their usage in the previous example.

Static and Dynamic Memory Allocation

Memory comes from one of two places

- run-time stack - automated allocation and deallocation for parameters and local variables as procedures are called. The storage is cleared when a procedure completes.
- the heap = handles all programatic requests for storage via `malloc` (C-style) and `new` (C++-style, objects/structs/etc). This storage must be returned with `free` or `delete`.

```
Balloon *b = new Balloon;
```

Pointer b is stored on the stack, the object it points to is on the heap.

```
#include ...

struct Node {
  string val;
  Node *next;
};

int main(...) {
  Node *p;
  p = new Node;
  p>val = "first";
  p->next = NULL;

  Node *q,r;
  r.val = "fluble";
  q = new Node;
  q->next = p;
  q->val = "second";

  Node *s = new Node;
  s->val = "third";
  s->next = q;

  Node *temp = s;

  while(NULL != temp) {
    cout << temp->val << endl;
    temp = temp->next;
  }
}
```

Abstract Data Types (ADTs)

An abstract data type is a structure that has a well-defined recognizable behaviour. It contains data which can only be accessed by a set of pre-written operations. Each of these operations has

- a signature (interface) which describes the parameters and return type
- a precondition (logic statement) which states what must be true before the operation may be applied
- a post-condition (logic statement) which describes the effects of evaluating the operation

Note that the formality and rigour of these interfaces varies greatly.

There are many common ADTs:

- a **vector** is an ordered data container that allows random access to individual elements
- a **stack** is an ordered data container with a Last-In-First-Out policy on reads/writes
- a **queue** is an ordered data container with a First-In-First-Out policy on reads/writes
- a **set** is an unordered data container that can retain a given value at most once (a multi-set can have multiples)
- a **map** is an unordered data container of pairs. i.e. if (a,b) and (a,c) are in this map, then $b == c$

There are multiple ways of implementing these, but the memory should be the same for each of them. Idealy, you could use any of these containers in the same way and receve the same output. C++ (but not C) provides mechanisms and run-time support for defining your own ADTs, as well.

Designing an interface which does exactly what we want *and nothing more* leads to one of the most important principles of software design: **information hiding**.

Information hiding is the act of separating interfaces from implementations and hiding those implementation details. Clients depend only on well-designed interfaces. Note that C does not provide support for information hiding.

The Stack

A linked list is a data container for objects for which we reference the next object from the previous one. This is how we could use one to implement a stack:

```
#include<iostream>
#include<string>
#include<cassert>
```

```
struct Node {
  string val;
  Node* next;
}

typedef Node* = stack;
bool isMT(Node *s) {
  return (NULL==s);
}

Node *push(Node *s, string val) {
  Node * newNode = new Node;
  newNode->val=val;
  newNode->next=s;
  return newNode;
}

string peek(Node *s) {
  assert(!isMT(s));
  return s->val;
}

void initStack() {
  return NULL;
}

Node *pop (Node *s) {
  assert(!isMT(s));
  Node *p = s->next;
  delete s;
  return p;
}

Node *nuke (Node *s) {
  while (!isMT(s)) s = pop(s);
}

int main(int argc, char *argv[]) {
  Node *s = stackInit();
  s = push(s, "abc");
  s = push(s, "ghi");
  cout << peak(s) << endl;
  s = pop(s);
}
```

Queues

We can implement a queue with the following:

```
#include<iostream>
#include<string>
#include<cassert>

struct Node {
  string val;
  Node * next;
}

struct Queue {
  Node *first;
  Node *last;
}

Queue initQ() {
  Queue q;
  q.first = NULL;
  q.last = NULL;
  return q;
}

bool isMTQ(q) {
  return NULL == q.last;
}

Queue enter (Queue q, string val) {
  Node *newNode = new Node;
  newNode->val = val;
  newNode->next = NULL;

  if (NULL == q.first) {
    q.first = newNode;
  } else {
    q.last->next = newNode;
  }

  q.last = newNode;
  return q;
}

string first Queue(q) {
```

```
  assert(!isMT(q));
  return q.first->val;
}

Queue leave(Queue q) {
  assert(!isMTQ(q));
  Node *p = q.first;
  q.first = q.first->next;

  if(q.first==q.last) {
    q.first=NULL;
    q.last = NULL;
  } else {
    q.first = q.first->next;
  }

  if(NULL == q.first) {
    q.last=NULL;
  }

  delete p;
  return q;
}

int main(...) {
  Queue ql;
  ql = enter(ql, "alpha");
  ql = enter(ql, "bravo");
  ql = enter(ql, "charlie");
  cout << first(q) << endl;
  ql = leave(ql);
  cout << first(ql) << endl
}
```

Variables and Recursion

There are three kinds of C++ variables:

- Global - these are preborn when declared and die at the end of program execution
- Instance/member variables - sub variables which are a struct class
- Local variables and parameters - these come into existence when they are declared during program execution and dieat the end of the enclosing scope (end of {} block, procedure, loop...)

Scope and Activation Records (ARs)

When a new scope is extended, a new AR is created on the run-time stack. When the current scope exits, the AR is popped and storage for any variables contained within it is destroyed.

Recursion

Recursion is a technique for solving a large problem by breaking it into smaller and smaller pieces until each is solveable. It usually contains three parts:

1. Trivial base case which can be solved easily and directly
2. Reduction operator which makes the input smaller
3. Composition operator which composes the answer from the smaller pieces

For example, to calculate a factorial we can do

```
int factorial(int n) {
  if(n <= 1) {
    return 1;
  } else {
    return n*factorial(nl);
  }
}

int main(...) {
  int k;
  cin >> k;
  cout << k << "!=" << factorial(k) << endl;
}
```

Reference Parameters

A reference parameters is an alias to some other variable. In C, we can cheat by passing a pointer; in C++, call-by-value and call-by-reference are both supported

```
void swap (int &x, int &y{...}          \\ by-value
void push (Stack &s, string val{...}    \\ by-reference
```

Reference parameters are the normal way to pass variables such that the changes are properly propogated.

```
void swap1 (int x, int y) {
  const int temp=x;
  x=y;
```

```
  y=temp;
}

void swap2 (int px, int *py) {
  const int temp=*px;
  *px=*py;
  *py = temp;
}

void swap3(int &x, int &y) {
  const int temp = x;
  x=y;
  y=temp;
}

int main(...) {
  int x=5;
  int y=37;
  swap1(x,y);
  swap2(&x,&y);
  swap3(x,y);
}

// WRONG, reference changes it
string peek1 (List &first) {
  string ans= first->val;
  first= NULL;
  return ans;
}

// first set to NULL on local copy, doesn't affect
string peek2(List first) {
  string ans= first->val;
  first= NULL;
  return ans;
}

// will not allow you to change value of first (not a guarantee, can use pointer tricks)
string peek3(const List &first) {
  string ans= first->val;
  first= NULL;
  return ans;
}
```

Dynamically Allocated Memory

Const reference parameters are used like reference paramenters, but the compiler will prevent you from changing the parameter inside the function. Effectively, it's similar to a value parameter, though they are more efficient.

Dynamic arrays such as vectors can dynamically allocate memory during run-time, through repeated usage of functions such as `push_back()`

Linked Lists

A linked list has a special pointer to its first element. Each element then has a pointer to the next or to NULL (if that element was the last item). Linked lists may be unordered by arrival time (e.g. stacks, queues), ordered by data (e.g. sorted by name), or ordered by some mixture (e.g. priority generic).

```
struct Node {
  string val;
  otherStuff mumble;
  Node *next;
}
```

Variant: Doubly-Linked List

```
struct Node {
  string val;
  otherStuff mumble;
  Node *prev;
  Node *next;
}
```

Binary Trees

A binary tree is a special construct such that each node has a value and up to two children nodes (i.e. "left" and "right"). There exists a special "root" node which has no parents.

A common subset is the **binary search tree (BST)**, which has the properties such that the left child has a value less than or equal to both the node's value and the right child's value. This is true recursively (i.e. the value of *all* left descendants is less than or equal to the value of the node...).

We can insert into a BST by searching for the proper parent, and pointing its left or right child to our new node. Note that insertion order affects how balanced our tree is. Printing can be done easily with a recursive algorithm. Deletion is more complicated.

To delete a node from a BST, we use the following cases:

1. no child: delete node, point parent to NULL
2. one child: delete node, point parent to child
3. two children:
    - largest key in the left is the parent's new child
    - largest key in the left node's smallest child points to its parents
    - largest key in the left node's left side points to its parents

For example:

```
#include<iostream>
#include<string>
#include<cassert>
#include<stack>

using namespace std;

struct BST_Node {
  string key;
  string stuff;
  BST_Node *left;
  BST_Node *right;
};

typedef BST_Node* BST;

void BST_init(BST &root) {
  root=NULL;
}

bool BST_isEmpty(const BST &root) {
  return NULL==root;
}

bool BST_has(const BST& root, string key) {
  if (BST_isEmpty(root)) {
    return false;
  } else if (key == root->key) {
    return true;
```

```
  } else if (key < root->key) {
    return BST_has(root->left, key);
  } else {
    return BST_has(root->right, key);
  }
}

string BST_lookup (BST& root, string key) {
  if (BST_isEmpty(root)) {
    return "";
  } else if (key == root->key) {
    return root->stuff;
  } else if (key < root->key) {
    return BST_has(root->left, key);
  } else {
    return BST_has(root->right, key);
  }
}

void BST_insert(BST& root, string key, string stuff) {
  assert(key != root->key);
  if (BST_isEmpty(root)) {
    root = new BST_Node;
    root->key = key;
    root->stuff = stuff;
    root->left = NULL;
    root->right = NULL;
  } else if (key < root->key) {
    BST_insert(root->left, key, stuff);
  } else if (key > root->key) {
    BST_insert(root->right, key, stuff);
  }
}

void BST_print(const BST& root) { //In order traversal version
  if(!BST_isEmpty(root)) {
    BST_print(root->left);
    cout << root->key << endl;
    BST_print(root->right);
  }
}

void BST_print(const BST& root) { //iterative version
  Stack<BST> nodeStack;
  BST cur = root;
```

```
  while (true){
    if(cur != NULL) {
      nodeStack.push(cur);
      root = cur->left;
    } else if(nodeStack.size == 0) {
      return;
    } else {
      cur = nodeStack.peak();
      nodeStack.pop();
     cout << cur->key << endl;
      cur = cur->right
    }
  }
}

void BST_delete(BST& root, string key) {}

int main() {
  return 0;
}
```

Iterative vs Recursive

Recursion is simpler, more elegant, and easier to read, though it can sometimes be inneffi-cient.

Priority Queues

A priority queue is a queue for which each element has both a value and a non-negative integer priority. `leave` means "remove the element of highest/lowest priority, if there are several of the same priority then remove the oldest". This is often used in network routing, since some types of data have higher priority (e.g. streaming video, VoIP, paid QoS).

C++ provides an STL container class called priority-queue. Alternatively, we can implement our own by using a linked list.

Object-Oriented Programming (OOP)

OOP involves **declarations** ("this is what it looks like") and **definitions** ("this is what it means"). These can either occur together or the declaration can occur first.

For example, **classes** can be implemented in the following fashion:

```
class Balloon {
  public:
    Balloon();
    Balloon (string colour);
    virtual ~Balloon();
    void speak() const;
  private:
    String colour;
};

Balloon::Balloon() {
  this->colour = "transparent";
}

Balloon::Balloon(string colour) {
  this->colour=colour;
}

Balloon::~Balloon() { }

void Balloon:: speak() {
  cout<<"I'm a " << colour << " balloon" << endl;
}
```

Classes include permission by default.

- "public": the API expected by clients. This should contain no variables or implementation details. This is the default for struct fields
- "private": the secret implementation details. This should contain all of the variable and some of the methods
- "protected": these are inherited by all descendents. This is the default for class fields

Classes also contain **constructors**, which are special methods used to create new instances. They have the same name as the class and no return type/value. These are called exactly once at the beginning of each instance's lifetime.

We also have **destructors**, which are called upon object death. These are usually trival (e.g. declared as `virtual`). These are most often used for memory cleanup.

Inheritance

```
class P {
  public:
    P();
```

```
    virtual ~P();
    virtual void m1();
    virtual void m2();
    int cv;
};

P::P() {}
P::~P() {}

void P::m1() {
  cout << "P::m1" << endl;
  m2();
}

void P::m1() {
  cout << "P::m1" << endl;
}

//C is concrete too
class C: public P {
  public:
    C();
    virtual ~C();
    virtual void m2();
    virtual void m3();
    int dv;
};

C::C() {}
C::~C() {}

void C::m2() {
  cout << "C::m2" << endl;
}

void C::m2() {
  cout << "C::m3" << endl;
}

int main() {
  P *f, *g;
  f = new P;
  f->m2(); //*
  g = new C;
  g->m2(); //*
```

```
  f->m1(); //*
  g->m1(); //*
  f->m3(); //ERROR
  g->m3(); //ERROR
  g->pv = 5; //Legal
  g->cv = 5; //NOT legal
  C *h = new C;
  h->m3();
  h->pv = 5;
  h->cv = 5;
}
```

will have an output of

- `P::m2`
- `C::m2` (if `P::m2` is not virtual, that would be called instead)
- `P::m1`, `P::m2`
- `P::m1`, `C::m2`