Ve 280
Programming and Introductory Data Structures

Improve ADT Efficiency;

Midterm Review

Abstract Data Types

Improving Efficiency

```
void IntSet::insert(int v) {
  if (indexOf(v) == MAXELTS) { // duplicate not found
    if (numElts == MAXELTS) throw MAXELTS; // no room
    int cand = numElts-1; // largest (last) element
    while ((cand >= 0) && elts[cand] > v) {
      elts[cand+1] = elts[cand];
      cand--;
    }
    // Now, cand points to the left of the "gap".
    elts[cand+1] = v;
    numElts++; // repair invariant
  }
}
```

Question: What is the situation when the loop terminates due to `cand < 0`? Is our implementation correct?

Abstract Data Types

Improving Efficiency

- **Question**: Do we have to change `indexOf`?

```
int IntSet::indexOf(int v) {
  for (int i = 0; i < numElts; i++) {
    if (elts[i] == v) return i;
  }
  return MAXELTS;
}
```

Abstract Data Types

Improving Efficiency

- **Question**: Do we have to change `indexOf`?
- **Answer**: No, but it can be made more efficient with the new representation.
- **How?** Using **binary search**! (The array is sorted)

```
int IntSet::indexOf(int v) {
  for (int i = 0; i < numElts; i++) {
    if (elts[i] == v) return i;
  }
  return MAXELTS;
}
```

Abstract Data Types

Complexity

	Unsorted	Sorted
query	O(N)	?
insert	?	?
remove	?	?

Abstract Data Types

Complexity

	Unsorted	Sorted
query	O(N)	O(log N)
insert	O(N)	O(N)
remove	O(N)	O(N)

`insert` and `remove` are still **linear**, because they may have to "swap" an element to the beginning/end of the array.

Abstract Data Types

Complexity

	Unsorted	Sorted
query	O(N)	O(log N)
insert	O(N)	O(N)
remove	O(N)	O(N)

- If you are going to do more searching than inserting/removing, you should use the "sorted array" version, because query is faster there.

- However, if query is relatively rare, you may as well use the "unsorted" version. It's "about the same as" the sorted version for insert and remove, but it's MUCH simpler!

Midterm Review

Midterm

- 10:00 am – 11:40 am, June 27th, 2017

- Find your seat on Canvas

- Closed book and closed notes

- No electronic devices are allowed
  - These include laptops and cell phones
  - We will show a clock on the screen

Midterm

- Written exam
  - A number of questions which only require you to provide a very short answer
  - A few questions which require you to write code on the paper. Be clear!

- Abide by the **Honor Code**!

Midterm Topics

- Linux Commands
- Compiling and Developing Program on Linux
- C++ Basics: Pointers, References, const Qualifier
- Procedural Abstraction and Specification Comments
- Recursion
- Function Pointers
- enum Type

- Program Taking Arguments
- I/O Streams
- Testing/Debugging
- Exception
- Class Basics

Lecture 1 to this lecture

Linux Commands

- cd; ls; mkdir; rmdir;
- cp; mv; rm;
- nano; gedit;
- less;
- diff; man; …
- I/O redirection
  - <, >
- Command options
  - ls -l; cp -r dir1 dir2; …
- Wildcard: *
  - cp *.h dir/

Compiling Program on Linux

- Write the source code, for example, using **gedit**

- Compile the program: g++ -o program source.cpp

- Run the program: ./program

- Compile multiple source files:

  - g++ -o program src1.cpp src2.cpp src3.cpp

  - E.g., g++ -o run_add run_add.cpp add.cpp

- Header guard: avoiding multiple inclusions

```
// add.h
#ifndef ADD_H
#define ADD_H
int add(int a, int b);
#endif
```

- What happens if the .h file is included **first** time?
- What happens if the .h file is included **second** time?

A Better Way of Compiling: Makefile

all: run_add

run_add: run_add.o add.o
        g++ -o run_add run_add.o add.o

run_add.o: run_add.cpp
        g++ -c run_add.cpp

add.o: add.cpp
        g++ -c add.cpp

clean:
        rm -f run_add *.o

- The file name is "Makefile"
- Type "make" on command-line

A Rule

Target: Dependency
<Tab> Command

Don't forget the Tab!

Dependency: A list of files that the target depends on

Function Call Mechanisms

There are two function call mechanisms:
1. Call-by-value
2. Call-by-reference

What will `a` be?

```
void f(int x)
{
    x *= 2;
}
```

```
void f(int& x)
{
    x *= 2;
}
```

```
int main()
{
    …
    int a=4;
    f(a);
    …
}
```
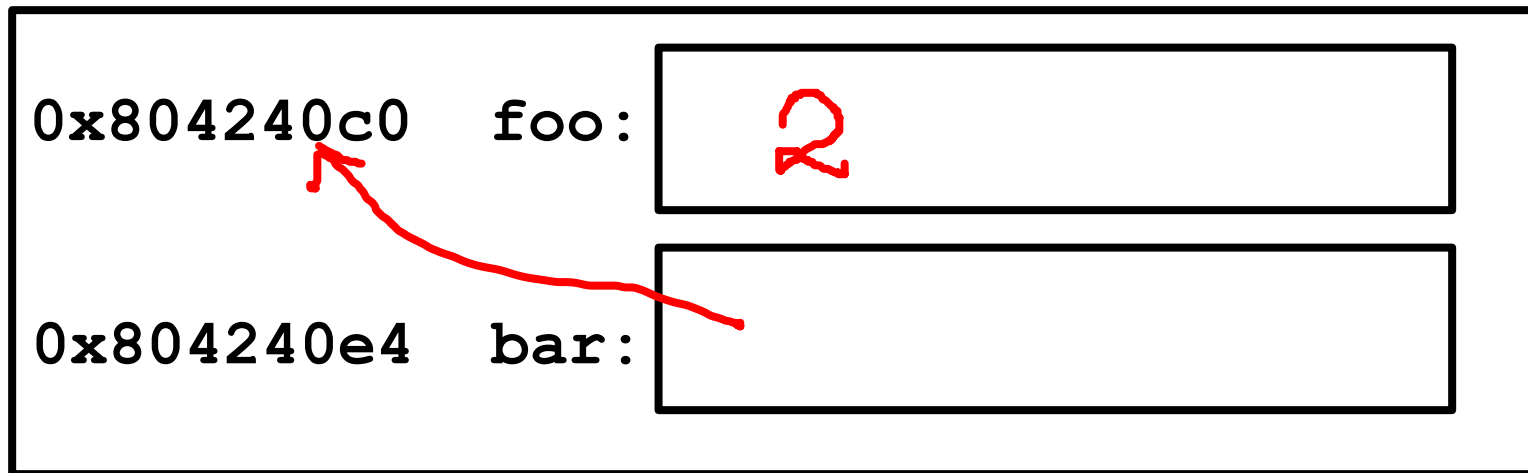
```
int main()
{
    …
    int a=4;
    f(a);
    …
}
```

Pointers

```
int foo = 1;
int *bar;
bar = &foo;  // addressing operation
*bar = 2;  // dereference operation
```

0x804240c0 foo: 2

0x804240e4 bar:

References

- An alternative name for an object

```
int iVal = 1024;
int &refVal = iVal;
```

- Reference **must be initialized** using a **variable** of the same type.

References Versus Pointers

Example

```
int x = 0;
int &r = x;
int y = 1;
r = y;
r = 2;
```

```
int x = 0;
int *p = &x;
int y = 1;
p = &y;
*p = 2;
```

What's the final values of
x, y, and r?

x = 2, y = 1, r = 2

What's the final values of
x, y, and *p?

x = 0, y = 2, *p = 2

const Qualifier

- Once you defined a constant variable, it cannot be modified later on.

  - ```
    const int a = 10;
    a = 11;  // Error
    ```

- Because we cannot subsequently change the value of an object declared to be const, we must initialize it when it is defined:

  - ```
    const int i;
    // Error: i is an uninitialized const
    ```

const Reference

```
int avg_exam(const struct Grades & gr){
    return (gr.midterm+gr.final)/2;
}
```

- It gives us the best of both worlds:
  - We don't have the expense of a copy.
  - We have the safety guarantee that the function cannot change the caller's state. Compiler will catch the error of accident change!

const Pointers

- When you have pointers, there are two things you might change:
  1. The value of the pointer.
  2. The value of the object to which the pointer points.

- Either (or both) can be made unchangeable:

```
const T *p;   // "T" (the pointed-to object)
pointer to const   // cannot be changed by pointer p
T *const p;   // "p" (the pointer) cannot be
const pointer   // changed
const T *const p; // neither can be changed.
```

Pointers to const

Example

```
int a = 53;
const int *cptr = &a;
    // OK: A pointer to a const object
    // can be assigned the address of a
    // nonconst object
*cptr = 42;
    // ERROR: We cannot use a pointer to
    // const to change the underlying
    // object.
a = 28 // OK
int b = 39;
cptr = &b; // OK: the value in the pointer
           // can be changed.
```

const Pointers

Example

```
int a = 53;
int *const cptr = &a;
   // OK: initialization
*cptr = 42;
   // OK: We can use a const pointer to
   // change the underlying object.
int b = 39;
cptr = &b;
   // ERROR: We cannot change the value of
   // a const pointer.
```

Pointer to const versus Normal Pointer

- Pointers-to-const-T are not the same type as pointers-to-T.

- You can use a pointer-to-T anywhere you expect a pointer-to-const-T, but NOT vice versa.

```
int const_ptr(const int *ptr)
{
    …
}


int main()
{
    int a = 0;              ✓
    int *b = &a;
    const_ptr(b);
}
```

```
int nonconst_ptr(int *ptr)
{
    …
}


int main()
{
    int a = 0;              ✗
    const int *b = &a;
    nonconst_ptr(b);
}
```

Abstraction

- Abstraction
  - Provides only those details that matter.
  - Eliminates unnecessary details and reduces complexity.

- Example: Multiplication algorithm
  - Many ways to do: table lookup, summing, etc.
  - Each looks quite different, but they do the **same** thing.
  - In general, a user won't care how it's done, just that it multiplies.

Procedural Abstraction

- Two important properties of procedural abstraction
  - Local: the implementation of an abstraction does not depend on any other abstraction implementation.
  - Substitutable: you can replace one (correct) implementation of an abstraction with another (correct) one, and no callers of that abstraction will need to be modified.

Procedural Abstraction

Specification Comments

- We describe procedural abstraction by specification comments.

- There are three clauses of specification comments:
  - **REQUIRES**: the pre-conditions that must hold, if any.
  - **MODIFIES**: how inputs are modified, if any.
  - **EFFECTS**: what the procedure computes given legal inputs.

- Note that the first two clauses have an "if any", which means they may be empty, in which case you may omit them.

Call Stacks

How a function call really works

- When a function is called, an activation record (also known as stack frame) is created. It holds the function's formal parameters and local variables.

- The **activation record** for **the current** invocation is added to the "top" of the stack.

- When that function returns, its **activation record** is removed from the "top" of the stack.

double add(double a, double b): $a = 1$, $b = 0$, $result = 0$
double sin(double x): $x = 1$, $result = 0$
int main(): $x = 1$, $sinResult = 0$

Recursion

$$n! = \begin{cases} 1 & (n == 0) \\ n * (n-1)! & (n > 0) \end{cases}$$

```
int factorial (int n)
    // REQUIRES: n >= 0
    // EFFECTS:  computes n!
1.  {
2.    if (n == 0) {
3.      return 1;  // 'base case'
4.    } else {
5.      return n*factorial(n-1); // 'recursive step'
6.    }
    }
```

Recursion

Writing a function for the general case

- Treat it like an inductive proof.

- To <u>write</u> a correct recursive function, do two things:
  1. Identify the "trivial" case (or cases), and write them explicitly.
  2. For all other cases, first assume there is a function that can solve smaller versions of the same problem, then figure out how to get from the smaller solution to the bigger one.

Recursive Helper Function

- Sometimes it is easier to find a recursive solution to a problem if you change the original problem slightly, and then solve that problem using a **recursive helper function**.

```
soln()
{
   …
   soln_helper();
   …
}
```

```
soln_helper()
{
   …
   soln_helper();
   …
}
```

Function Pointers

Motivation

- If you were asked to write a function to add all the elements in a list, and another to multiply all the elements in a list, your functions would be almost exactly **the same**.

- Writing almost the exact same function twice is almost certainly a bad idea

    Function pointers to the rescue!

Function Pointers

A first look

```
int min(int a, int b);
  // EFFECTS: returns the smaller of a and b.
int max(int a, int b);
  // EFFECTS: returns the larger of a and b.
```

- These two functions have precisely the same type signature:
  - They both take two integers, and return an integer.
- Of course, they do completely different things:
  - One returns a min and one returns a max.
  - **However, from a syntactic point of view, you call either of them the same way.**

Function Pointers

Basic Format

- Declaration

```
int  (*foo)(int, int);
```

- Once defined, we can assign it to a function that has **the same type signature**

```
int min(int a, int b);
foo = min;
```

- Furthermore, after assigning min to foo, we can just call it as follows:

```
foo(3, 5)
```

…and we'll get back 3!

Enum Type

- Define an enumeration type as follows:

    ```
    enum Suit_t {CLUBS, DIAMONDS,
                    HEARTS, SPADES};
    ```

- Define variables of this enum type:

    ```
    enum Suit_t suit;
    ```

- You can initialize them as:

    ```
    enum Suit_t suit = DIAMONDS;
    ```

- Once you have such an enum type defined, you can use it as an argument for a function.

Enum Type

- If you write

  enum Suit_t {CLUBS, DIAMONDS,
  HEARTS, SPADES};

  then numerically

  ```
  CLUBS = 0, DIAMONDS = 1,
  HEARTS = 2, SPADES = 3
  ```

- Using this fact, it will sometimes make life easier

  ```
  enum Suit_t s = CLUBS;
  const string suitname[] = {"clubs",
      "diamonds", "hearts", "spades"};
  cout << "suit s is " << suitname[s];
  ```

Passing Arguments to a Program

- Programs can take arguments.

  `diff file1 file2`

- Arguments are passed to the program through main() function.

- We need to change the argument list of main():
  - `int main(int argc, char *argv[])`

- `argv` stores the array of C-strings that user inputs.
  - `argv[0]` is the name of the program being executed.

- `argc` is the number of strings in the array

I/O Streams

- Output Stream `cout`
  - Insertion operator `<<`

- Input Stream `cin`
  - extraction operator `>>`
  - `getline(cin, str)`
  - `cin.get(ch)`
  - Failed input stream: check stream state `if(cin)`

- `cout` and `cin` streams are buffered.

I/O Streams

- File Stream
  - `ifstream; ofstream`
  - Opening a file: `iFile.open("myText.txt");`
  - extraction >> ; insertion <<

- String Stream
  - `istringstream; ostringstream`
  - extraction >> ; insertion <<
  - Assign a string to an input string stream `iStream.str(line);`
  - fetch the string value from an output string stream `oStream.str();`

Testing

- Be skeptical!
- Incremental testing
- Five Steps:
1. Understand the specification
2. Identify the required behaviors
3. Write specific tests
   - **Simple inputs**
   - **Boundary conditions**
   - **Nonsense**
4. Know the answers in advance
5. Include stress tests

Debugging Using Assert

- Using the `assert` function
  - The `assert` function is a special function, which takes a Boolean argument.
  - If the argument is **true**, `assert()` does nothing.
  - If the argument is **false**, `assert()` causes your program to stop, printing an **error message** to the `cerr` stream.

- `assert` for the condition that should hold.

Exceptions

- Exceptions and exception handling mechanism

- **Exception propagation** mechanism: where to find the handler
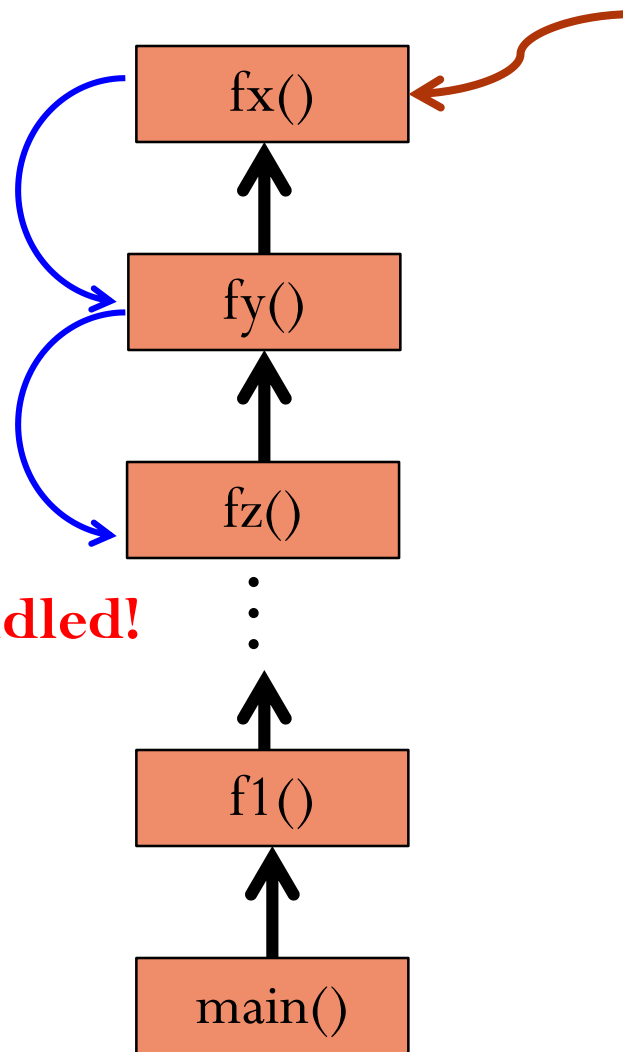
Exceptions
Exception Handling

handler in fx()? **No!**

handler in fy()? **No!**

handler in fz()? **Yes!**

Exception handled!

fx()

fy()

fz()

f1()

main()

exception occurs

Exceptions

- Throwing an exception

- Catching an exception

- Exceptions have **types** and **objects**.
  - **throw errorObj;**

- Exceptions Handling in C++

```
void foo() {
        try {            }
        catch (Type var) {            }
    }
```

Abstract Data Types

- The role of a type:
  - The set of values that can be represented by items of the type
  - The set of operations that can be performed on items of the type.

- An abstract data type provides an **abstract description** of **values** and **operations**.

- Advantages: <u>Information hiding</u> and <u>encapsulation</u>.

C++ Classes

- Data members and function members are defined in a single entity.

- **Public** versus **private** members.

- Defining a class type.

- Class object as a function argument: pass by value

C++ Classes

- **Constructor** for intilization: `IntSet();`

- Initialization syntax:
  `IntSet::IntSet(): numElts(0)`
  `{}`

- const member function: `int size() const;`

  - <u>Means</u>: the member function **`size()`** cannot change the object on which **`size()`** is called.

  - Syntax: if a const member function calls other **member** functions, they must be **const** too!

    `void A::g() const { f(); }`

`void A::f() {...}` ✘   **`void A::f() const {...}`** ✔