(5 - 1) Object-Oriented Programming (OOP) and C++

Instructor - Andrew S. O’Fallon
CptS 122 (February 5, 2018)
Washington State University

Key Concepts

- Object-Oriented Design
- Object-Oriented Programming (OOP)
- Class and Objects
- Data Encapsulation
- Abstraction/Information Hiding
- C++ I/O
- References and Reference Parameters
- Unary Scope Resolution Operator
- Function Overloading

A. O'Fallon, J. Hagemeister

Object-Oriented Design (OOD)

- Model software in ways that are similar to how people view/describe real-world objects
- Descriptions and designs include properties or attributes of the real-world objects
- The Unified Modeling Language (UML) provides a specification for illustrating properties of objects along with interactions between them

Object-Oriented Programming (OOP) (I)

- Programming language model which institutes mechanisms to support implementing object driven software systems
  - C++, C#, Java
- Procedural programming, such as instituted by C, is action oriented
- In C, the unit of programming is a function
- In C++ the unit is a `class`

A. O'Fallon, J. Hagemeister

Object-Oriented Programming (OOP) (II)

- We'll explore OOP with classes, encapsulation, objects, operator overloading, inheritance, and polymorphism
- We'll also explore generic programming with function templates and class templates

A. O'Fallon, J. Hagemeister

Classes and Objects

- What is a `class`?
  - A user defined type or data structure
  - Contains data members (attributes) and member functions (operations)
  - A blueprint for an object
- What is an object?
  - An instantiation of a class
  - The class is the type and the object is the variable with allocated memory for that type

Data Encapsulation (I)

- A way of organizing or wrapping of data/attributes and methods/operations into a structure (or capsule)
  - Demonstrated by objects
- Objects naturally impose encapsulation – attributes and operations are closely tied together
- How does making a function or class a `friend` of another class impact encapsulation?

Abstraction/Information Hiding (I)

- A design principle which states a design decision should be hidden from the rest of the system
- In other words, objects should communicate with each other through well-defined interfaces, but not know how other objects are implemented

A. O'Fallon, J. Hagemeister

Abstraction/Information Hiding (II)

- Prevents access to data aside from the methods specified by the object
- Guarantees integrity of data
- Access specifiers in C++ control the access to information
  - `public`, `protected`, and `private`

A. O'Fallon, J. Hagemeister

Programming in C++

- When programming in an object-oriented language, we'll be exposed to encapsulation, abstraction, and information hiding in action
- We need to start thinking in an object-oriented way so that we can leverage the software design benefits of objects and the richness of C++!
- Always remember, objects contain data and associated operations!

A. O'Fallon, J. Hagemeister

Basics of C++ and I/O (I)

- In C++, just like in C, every program begins execution with function main ()
- To perform input and output (I/O) we need to include the C++ Standard Library `<iostream>`
  - Essentially replaces `<stdio.h>,` but with even more richness and convenience

A. O'Fallon, J. Hagemeister

Basics of C++ and I/O (II)

- In tandem with including `<iostream>`, we'll need to use the following:
  - A standard output stream object (std::cout) and stream insertion operator (<<) to display information on the screen
    - Replaces the need for printf ()
  - A standard input stream object (std::cin) and the stream extraction operator (>>) to read data from the keyboard
    - Replaces the need for scanf ()

A. O'Fallon, J. Hagemeister

Basics of C++ and I/O Example

```
#include <iostream>
using std::cin; // replaces need for std:: in front of cin
using std::cout; // replaces need for std:: in front of cout
using std::endl; // replaces need for std:: in front of endl
int main (void)
{
        int n1 = 0;
        cout << “Enter a number: “;
        cin >> n1; // Notice no address of (&) required!

        int n2 = 0, sum = 0; // Can declare variables right
                             // before their use in C++!
        cout << “Enter a second number: “;
        cin >> n2;
        sum = n1 + n2;
        cout << “The sum is: “ << sum << endl; // endl outputs a
                                               // newline, then flushes buffer
        return 0;
}
```


A. O’Fallon, J. Hagemeister

References and Reference Parameters (I)

- There are two ways to pass arguments to functions in C++
  - Pass-by-value (PBV) – a copy of the contents/value of each argument is made and passed (on the function call stack) to the called function
    - One disadvantage of pass-by-value is copying the contents of a large data item or object introduces longer execution times and memory space
    - In general, should only be used with simple types
    - Passing-by-pointer falls under this category
  - Pass-by-reference (PBR) – NO copy of the contents/value of each argument is made
    - The called function can access the caller's data directly, and modify the data

A. O'Fallon, J. Hagemeister

References and Reference Parameters (II)

- Thoughts: we don’t use pass-by-reference strictly so that we can modify the data in an object directly, in many cases we use it so that the overhead of copying data is circumvented
- We use the ampersand (&) to represent pass-by-reference
  - i.e. void cube (int **&n**); // this is a prototype
  - Don’t confuse with the address of (&) operator! Context determines which one’s in play!
- Check out: http://www.cplusplus.com/articles/z6vU7k9E/

A. O’Fallon, J. Hagemeister

References and Reference Parameters (III)

- We can return a reference to a variable as well – however we have to be very careful!
  - i.e. **int &** someFunction (int &n);
- If we return a reference to an automatic local variable, the variable becomes "undefined" when the function exits; unless the variable is declared as "static" (keyword)
  - References to undefined variables are called *dangling* references
  - Note: dangling references and dangling pointers are NOT the same!

A. O'Fallon, J. Hagemeister

References and Reference Parameters Example

```
…
void cubeByRef (int &n);
void cubeByPtr (int *pN);
int main (void)
{
		int n = 5;
		cubeByRef (n); // Don’t need &, the formal parameter list indicates PBR
		cubeByPtr (&n); // Need address of (&) operator to satisfy pointer; applying PBV
…
}
void cubeByRef (int &n)
{
		n = n * n * n; // We have direct access to n, don’t need to dereference;
				// changes are retained
}
void cubeByPtr (int *pN)
{
		*pN = (*pN) * (*pN) * (*pN); // Need to dereference to indirectly change value
}
```


A. O’Fallon, J. Hagemeister

Unary Scope Resolution Operator

- It's possible to declare local and global variables of the same name
  - Unary Scope Resolution Operator (::) allows a global variable to be accessed without confusing it with a local variable

```
…
int num = 42; // global variable
int main (void)
{
    double num = 100.25; // local variable
    cout << num << endl; // displays 100.25
    cout << ::num << endl; // displays 42
}
```

Function Overloading (I)

- The ability to define multiple functions with the same name
  - Requires that each function has different types of parameters and/or different number of parameters and/or different order of parameters
  - i.e. `int cube (int n);`

    `double cube (double n);`
- The C++ compiler selects the appropriate function based on the *number*, *types*, and *order* of arguments in the function *call*

A. O'Fallon, J. Hagemeister

Function Overloading (II)

- We use function overloading to increase readability and understandability
  - Of course, we only want to overload functions that perform similar tasks

A. O'Fallon, J. Hagemeister

C++ Standard Template Library (STL) Class Vector

- STL class `vector` represents a more robust array with many more capabilities
- May operate with different types of data because they're templated!
  - i.e.

```
vector<int> v1(10); // declares a 10 element
                    // vector of integers

vector<double> v2(5); // declares a 5 element vector
                      // of doubles
```


A. O'Fallon, J. Hagemeister

Closing Thoughts

- OOP and C++ opens us up to an entirely different world!
- We need to start thinking more in terms of data and “capsules” instead of just actions and logic
- Learning C++ is a challenge, but provides features that will increase levels of production!

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014.

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister

(5 – 2) Introduction to Classes in C++

Instructor - Andrew S. O'Fallon
CptS 122 (February 7, 2018)
Washington State University

Key Concepts

- Function templates
- Defining classes with member functions
- The Rule of Three, Law of The Big Three, or The Big Three
- Constructors
  - Default and copy
- Destructors
- Setters (mutators) and getters (accessors)

A. O'Fallon, J. Hagemeister

Function Templates

- Overloaded functions are generally defined to perform similar operations that involve different types and/or program logic
- What happens if the program logic and operations are identical for each type?
  - Function templates may be used to more concisely overload functions

A. O'Fallon, J. Hagemeister

Function Template Example

```
… // template function must be placed in .h files!
template <class T>
T add (T v1, T v2)
{
          T result;
          result = v1 + v2;
          return result;
}

… // start of .cpp file!
int main (void)
{
          int n1 = 10, n2 = 20, n3 = 0;
          double d1 = 35.75, d2 = 45.5, d3 = 0.0;
          // Single function template provide capability of defining
         // a family of overloaded functions!
          n3 = add (n1, n2); // C++ generates overloaded function for integers
          d3 = add (d1, d2); // C++ generates overloaded function for doubles
}
```

Classes w/ Member Functions (I)

- Recall, in C++ we can create a user-defined type using the keyword `class`
- Also, recall, an *object* is an instantiation of a `class`
- A class consists of *data members* (attributes) and *member functions* (operations)
- A `class` controls access to its members
- Typically you cannot call a *member function* unless an object of the `class` has been instantiated
  - One exception to this rule is when you declare a member function with the keyword `static`

A. O'Fallon, J. Hagemeister

Classes w/ Member Functions (II)

- Classes allow the developer to separate *interfaces* from *implementation*, which is a principle of good software engineering
  - We generally place our function prototypes for member functions in the class.h file and our implementation for these in our class.cpp file
  - The function prototypes describe the classes `public` interfaces without exposing the internal implementation of the member functions

A. O'Fallon, J. Hagemeister

Classes w/ Member Functions (III)

- Objects can interact with each other by sending *messages*
- *Messages* are sent from one object to another by calling a method on that object
  - These methods are generally `public` member functions

Example Class ComplexNumber w/ Member Functions (I)

- Let's define the class for a complex number
- Recall, a complex number consists of a real part and imaginary part in the form:
  - a + b*i*, where a and b are real numbers, and *i* is the imaginary unit $i = \sqrt{-1}$
- In our design `class` ComplexNumber will consist of two data members
  - double realPart // we choose double because these are real numbers
  - double imaginaryPart

A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (II)

- Let's add the data members to the class
  - To follow the good software engineering practice of *information hiding* we will make the data members `private`
  - `private` members may only be accessed directly by member functions of the class (or `friend`s)

```
class ComplexNumber
{
        private:
                // m - represents member of a class
                double mRealPart;
                double mImaginaryPart;
}; // Don't forget the semicolon!
```


A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (III)

- Now let's consider operations that we need to perform on the data members of the class
  - We should be able to add and subtract two complex numbers
  - We should also be able to print complex numbers in the form a + b*i*
  - We could perform each of these operations by using the C++ *operator overloading* capability, but we'll reserve that for another example

A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (IV)

- Let's add the member functions to the class
  - The member functions will represent the well-defined interfaces to the "outside" world, thus, we'll make them `public`
  - `public` functions may be accessed by other (non-member) functions in the program as well as member functions of other classes

```
class ComplexNumber
{
        public:
                // const forces the implementation to NOT allow
                // the operand object to be modified; pass-by-ref
                // so a copy of the operand object is not made!
                ComplexNumber add (const ComplexNumber &operand);
                ComplexNumber sub (const ComplexNumber &operand);
                // Remember since print () is a member function,
                // it has access to the private data members,
                 // so no parameters are required!
                void print ();
        private:
                double mRealPart; // m – represents member of a class
                double mImaginaryPart;
}; // Don't forget the semicolon!
```


A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (V)

- Now that we've seen how to define some parts of a `class`, let's focus on how to define the definitions for the member functions
- All member functions must be associated with a `class`
  - Since we'll separate our interface (.h) from our implementation (.cpp), we'll need to use the binary scope resolution operator (::) to provide this association
    - Don't confuse this operator with the unary scope resolution operator!

A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (VI)

- Let’s write the definition for the `add()` member function

```
// Prototype: ComplexNumber add (const ComplexNumber &operand);

// Definition – notice the binary scope resolution operator
ComplexNumber ComplexNumber::add (const ComplexNumber &operand)
{
          // This adds the real part of the “operand” object
          // to the real part of the object that invokes the
          // call to the function; it also adds the imaginary
          // parts
          ComplexNumber result; // Declare local ComplexNumber

          // Recall we use the dot member operator (.) to access
          // members of a class; no dot (.) denotes accessing the
          // instantiated object’s members; note we don’t have to apply “special”
          // operators to access an object passed by reference!
          result.mRealPart = mRealPart + operand.mRealPart;
          result.mImaginaryPart = mImaginaryPart + operand.mImaginaryPart;
           // Don’t want to pass back by reference; cause undefined behavior
          return result;
}
```

Example Class ComplexNumber w/ Member Functions (VII)

- Could you write the definition for the `sub()` function? Try it!

A. O'Fallon, J. Hagemeister

Example Class ComplexNumber w/ Member Functions (VIII)

- Let’s write the definition for the `print()` member function

```
// Prototype: void print ();
// Definition
void ComplexNumber::print ()
{
      // Print in the form -> real + imaginary i
      cout << mRealPart << “ + “ <<
              mImaginaryPart << “i” << endl;
}
```

The Rule of Three

- Also known as the Law of The Big Three or The Big Three
- The rule states that if one or more of the following are defined, then all three should be explicitly defined
  - Destructor
  - Copy constructor
  - Copy assignment operator

How to Instantiate Objects from main ()? (I)

- Continuing with our ComplexNumber example…

```
int main (void)
{
        // Instantiate three objects!
        ComplexNumber c1, c2, c3;
        // Some other code needs to be in place for this
        // to work in reality…
        c3 = c1.add (c2); // c1 invokes the add () call
        // c3 contains the result, so it invokes the
        // print () call
        c3.print ();

        return 0;
}
```

How to Instantiate Objects from main ()? (II)

- You should be asking yourself how do we know which values are stored in each of the ComplexNumber objects (c1, c2, c3) for each real and imaginary part
  - Right now we really don't know…most likely 0 for both data members though…
  - We need to create a means of initializing our objects
    - *Constructor* functions solve this problem for us!

A. O'Fallon, J. Hagemeister

Constructors for Initializing Objects! (I)

- Each `class` declared provides a *constructor* that may be used to *initialize* an object
- A *constructor* is a special member function because it MUST be named the same as the `class`, it cannot return a value, and it is called *implicitly* when an object is instantiated
- If a class does not *explicitly* provide a constructor, then the compiler provides a *default* constructor (a constructor with no parameters)
- Generally constructors are declared `public`
- When is an object instantiated?
  - When a variable of the type of `class` is declared
  - When the `new` operator is explicitly invoked
    - Note: `new` is used in place of malloc () for C++

Constructors for Initializing Objects! (II)

- Let's add a *default* constructor to our ComplexNumber `class`

```
class ComplexNumber
{
          public:
                    ComplexNumber (); // Default constructor
                    // const forces the implementation to NOT allow
                    // the operand object to be modified; pass-by-ref
                    // so a copy of the operand object is not made!
                    ComplexNumber add (const ComplexNumber &operand);
                    ComplexNumber sub (const ComplexNumber &operand);
                    // Remember since print () is a member function,
                    // it has access to the private data members,
                     // so no parameters are required!
                    void print ();
          private:
                    double mRealPart; // m - represents member of a class
                    double mImaginaryPart;
}; // Don't forget the semicolon!
```

Constructors for Initializing Objects! (III)

- Let's write the definition for the *default* constructor member function

```
// Prototype: ComplexNumber ();
// Definition
void ComplexNumber::ComplexNumber()
{
      // Initialize the data members
      mRealPart = 0.0;
      mImaginaryPart = 0.0;
}
```


A. O'Fallon, J. Hagemeister

Constructors for Initializing Objects! (IV)

- Notice the default constructor sets the real and imaginary parts to 0
- What if we want to set the parts to values other than 0?
  - We create another version of the constructor, which accepts parameters
    - This implies we need to overload our constructor!

```
ComplexNumber (double real, double imaginary);

ComplexNumber::ComplexNumber (double real, double imaginary)
{
        mRealPart = real;
        mImaginaryPart = imaginary;
}
```

How to Initialize Objects with a Constructor?

```
int main (void)
{
        // Instantiate three objects! Use a constructor that
        // supports arguments!
        ComplexNumber c1(2.5, 3.5), c2(1.25, 5.0), c3;

        // With the addition of constructors we now know the following:
        // c1 = 2.5 + 3.5i, c2 = 1.25 + 5.0i, c3 = 0.0 + 0.0i
        c3 = c1.add (c2); // c1 invokes the add () call
        // State of c3? It should be c3 = 3.75 + 8.5i

        // c3 contains the result, so it invokes the
        // print () call
        c3.print (); // Would print 3.75 + 8.5i

        return 0;
}
```


A. O'Fallon, J. Hagemeister

Copy Constructor

- A *copy* constructor always accepts a parameter, which is a reference to an object of the same `class` type

```
ComplexNumber (ComplexNumber &copyObject);
```

- Copy constructors make a *copy* of an object of the same type
- A copy constructor is *implicitly* invoked when an object is *passed-by-value*!
- A *shallow* copy is made if only the data members are copied directly over to the object
- A *deep* copy is made if new memory is allocated for each of the data members
- We will explore these constructors more in the future!

A. O'Fallon, J. Hagemeister

Destructors

- Each `class` declared provides a *destructor*
- A *destructor* is a special member function because it MUST also be named the same as the `class` (with a tilde (~) in front) it cannot return a value, and it is called *implicitly* when an object is destroyed

```
~ComplexNumber ();
```

- If a class does not *explicitly* provide a destructor, then the compiler provides an “empty” destructor
- When does an object get destroyed?
  - When the object leaves scope
  - When the `delete` operator is explicitly invoked
    - Note: `delete` is used in place of free () for C++

A. O’Fallon, J. Hagemeister

Setters and Getters

- These are `public` interfaces/functions to provide access to `private` data members
- *Setters* allow *clients* of an object to *set* or modify the data members
  - *Clients* include any statement that calls the object's member functions from *outside* the object
  - May be used to *validate* data
- *Getters* allow client to obtain/*get* a *copy* of the data members
- There generally should be 1 setter function per data member, and 1 getter function data member (of course this depends on whether or not a data member should be accessed by a client object)

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister

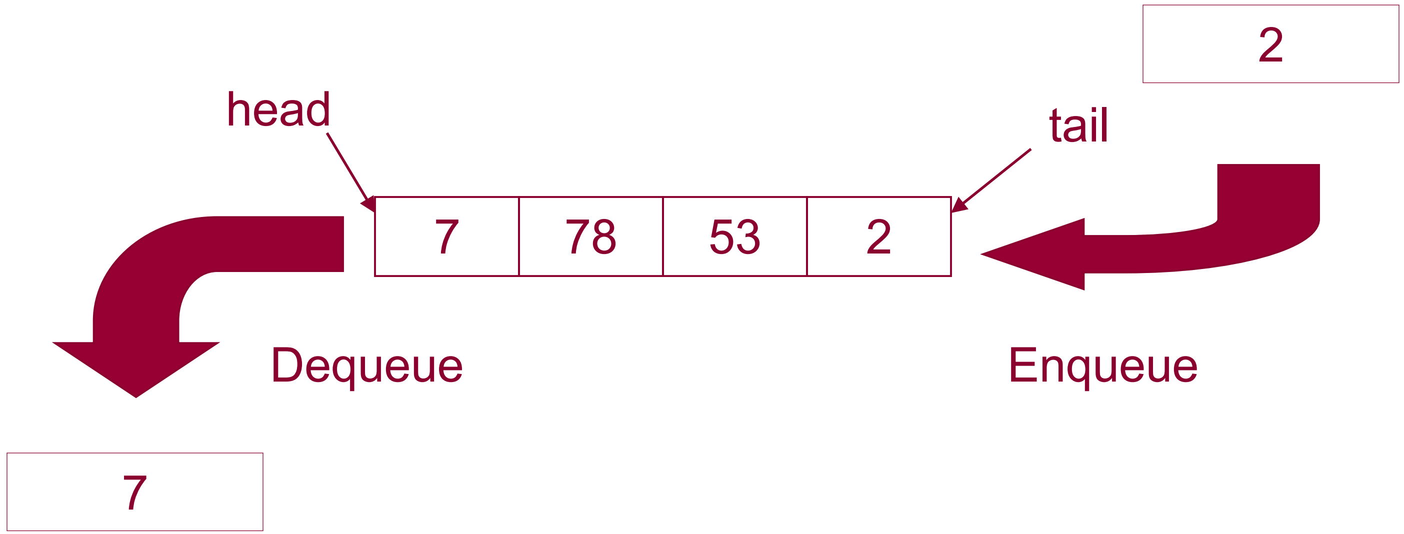
(6-1) Basics of a Queue

Instructor - Andrew S. O'Fallon
CptS 122 (February 12, 2018)
Washington State University

What is a Queue?

- A linear data structure with a finite sequence of nodes, where nodes are removed from the front or head and nodes are inserted at the back or tail
- A queue is referred to as a first-in, first-out (FIFO) data structure
  - Consider a grocery store line; as the line forms, people enter at the back or tail of the line; the person at the front or head of the line is always serviced before the others; once the front person is serviced, he/she leaves and the next in line is helped
- A queue is also considered a restricted or constrained list
- We will focus most of our attention on linked list implementations of queues

Typical Representation of Queue of Integers



Implementation of Queues in C

- The following slides will show how to implement Queues in C
- We will implement them in C++ during lecture

Struct QueueNode

- For these examples, we'll use the following definition for `QueueNode`:

```
typedef struct queueNode
{
    char data;
    // self-referential
    struct queueNode *pNext;
} QueueNode;
```

Initializing a Queue in C (1)

- Our implementation:

```
void initQueue (QueueNode **pHead,
                QueueNode **pTail)
{
    // Recall: we must dereference a
    // pointer to retain changes
    *pHead = NULL; // Points to front of queue
    *pTail = NULL; // Points to back of queue
}
```


A. O'Fallon, J. Hagemeister

Initializing a Queue in C (2)

- The `initQueue()` function is elementary and is not always implemented
- We may instead initialize the pointers to the front and back of the queue with `NULL` within `main()`

```
int main (void)
{
    QueueNode *pHead = NULL; // points to front
    QueueNode *pTail = NULL; // points to back
    …
}
```


A. O'Fallon, J. Hagemeister

Initializing a Queue in C (3)

- We can combine the two pointers (pHead and pTail) of a queue into a single struct called `Queue`

```
typedef struct queue
{
    QueueNode *pHead;
    QueueNode *pTail;
} Queue;
```

- We can then modify our `initQueue()` to accept a `Queue` struct type

```
void initQueue (Queue *pQueue)
{
    pQueue -> pHead = NULL;
    pQueue -> pTail = NULL;
}
```


A. O'Fallon, J. Hagemeister

Checking for Empty Queue in C (1)

- Only need to check the head pointer to see if the queue is empty
- Our implementation:

```
int isEmpty (Queue q)
{
    // Condensed the code into
    // one statement; returns 1 if
    // pHead is NULL; 0 otherwise
    return (q.pHead == NULL);
}
```

Checking for Empty Queue in C (2)

- Note: we could substitute the `int` return type with an enumerated type such as Boolean

```
typedef enum boolean
{
   FALSE, TRUE
} Boolean;
```

Checking for Empty Queue in C (3)

- Our implementation with `Boolean` defined:

```
Boolean isEmpty (Queue q)
{
    Boolean status = FALSE;

    if (q.pHead == NULL) // Queue is empty
    {
        status = TRUE;
    }

    return status;
}
```


A. O'Fallon, J. Hagemeister

Printing Data in Queue in C

- A possible implementation using recursion:

```
void printQueueRecursive (QueueNode *pHead)
{
     if (pHead != NULL) // Recursive step
     {
          printf ("%c ->\n", (pHead) -> data);
          // Get to the next item
          pHead = (pHead) -> pNext;
          printQueueRecursive (pHead);
     }
     else // Base case
     {
          printf ("NULL\n");
     }
}
```

Inserting Data into Back of Queue with Error Checking in C (1)

- Let's modify our code so that we can check for dynamic memory allocation errors
- We'll start with `makeNode()`:

```
QueueNode * makeNode (char newData)
{
      QueueNode *pMem = NULL;

      pMem = (QueueNode *) malloc (sizeof (QueueNode));
      if (pMem != NULL)
      {
              // Initialize the dynamic memory
              pMem -> data = newData;
              pMem -> pNext = NULL;
      }
      // Otherwise no memory is available; could use else, but
      // it's not necessary

      return pMem;
}
```

Inserting Data into Back of Queue with Error Checking in C (2)

- Now let's add some error checking to `enqueue()`:

```
Boolean enqueue (Queue *pQueue, char newData)
{
        QueueNode *pMem = NULL;
        Boolean status = FALSE; // Assume can't insert a new node; out of memory

        pMem = makeNode (newData);

        if (pMem != NULL) // Memory was available
        {
                // Insert the new node into the back of the queue
                if (isEmpty (*pQueue)) // Inserting first node into queue
                {
                        pQueue -> pHead = pMem;
                }
                else // Already at least one node in queue; update tail only
                {
                        pQueue -> pTail -> pNext = pMem;
                }
                pQueue -> pTail = pMem;
                status = TRUE; // Successfully added a node to the queue!
        }

        return status;
}
```


A. O'Fallon, J. Hagemeister

Removing Data from Front of Queue in C (1)

- We will apply defensive design practices and ensure the queue is not empty
- This implementation of `dequeue()` returns the data in the node at the front of the queue

```
char dequeue (Queue *pQueue)
{
	char retData = '\0';
	QueueNode *pFront = NULL;

	if (!isEmpty (*pQueue)) // Stack is not empty; defensive design
	{
		pFront = pQueue -> pHead; // Temp storage of front of queue
		retData = pQueue -> pHead -> data;
		pQueue -> pHead = pQueue -> pHead -> pNext;
		if (pQueue -> pHead == NULL) // Queue is now empty; update tail
		{
			pQueue -> pTail = NULL;
		}
		free (pFront); // Remove the front node
	}

	return retData;
}
```


A. O'Fallon, J. Hagemeister

Queue Applications

- Operating systems maintain queues of processes that are ready to execute
- Printers queue print requests; first-come, first-serve
- Simulations of real world processes, such as movie lines, grocery store lines, etc.

A. O'Fallon, J. Hagemeister

Closing Thoughts

- Can you build a driver program to test these functions?
- A queue is essentially a restricted linked list, where one additional pointer is needed to keep track of the back, tail, or rear of the queue
- You can implement a queue without using links; Hence, you can use an array as the underlying structure for the queue

References

- P.J. Deitel & H.M. Deitel, *C: How to Program* (7th ed.), Prentice Hall, 2013
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister

(6 – 2) Streams and File Processing in C++

Instructor - Andrew S. O’Fallon
CptS 122 (February 16, 2018)
Washington State University

What is a Stream? A Refined Definition

- A *sequence* of objects (generally just considered bytes) that flow from a device to memory or from memory to a device
- For *input* operations, the bytes flow from the device (i.e. keyboard, network connection, disk, etc.) to main memory
- For *output* operations, the bytes flow from main menu to the device (screen, printer, etc.)

Analogy for a Stream

- A conveyer belt
  – You can place an item in sequence on the belt, i.e. into the stream (insertion or output operation)
  – You can remove an item in sequence from the belt, i.e. take from the stream (extraction or input operation)

A. O'Fallon, J. Hagemeister

Classic Streams vs. Standard Streams

- The *classic* input/output streams for C++ supported byte-sized `char`s, which represented the ASCII characters
- Many alphabets require *more* characters than can be represented by a *byte* and the ASCII character set does not provide the characters
  - The *Unicode* character set provides these ones
- C++ provides *standard* stream libraries to process Unicode characters (`wchar_t`)

A. O'Fallon, J. Hagemeister

Standard Streams in C++ (1)

- For *standard* input/output streams, include `<iostream>`
  - `cin` is a predefined *object* of class `istream` and is connected to the standard input device (i.e. keyboard)
    - `cin >> var // cin applying stream extraction operator – stops at whitespace for strings`
  - `cout` is a predefined *object* of class `ostream` and is connected to the standard output device (i.e. screen)
    - `cout << var // cout applying stream insertion operator`

A. O'Fallon, J. Hagemeister

Standard Streams in C++ (2)

- *Member* function `getline()` will read a line from the stream
  - Inserts a null character at the end of the array of characters, removes and discards the ‘\n’ from the stream (i.e. stored as a C string)

A. O’Fallon, J. Hagemeister

Recall the File Processing Algorithm!

- Step 1: open the desired file
  - Opening is based on filename and permissions (read, write, or append)
  - Associates a file with a stream
- Step 2: process the file
  - Read data from the file
    - Does not affect file
  - Write data to the file
    - Completely overwrites existing file
  - Add data to the end of the file
    - Retains previous information in file
- Step 3: close the file
  - Disassociates a file from a stream

Files Streams in C++ (1)

- For input/output streams to work with *files*, include <fstream>
  - `ifstream` objects enable input from a file
  - `ofstream` objects enable output to a file
  - `fstream` objects for input from and output to a file
- Associate file with a file stream either during construction (applying the constructor or by calling `open()`)
  - `fstream fstr("filename.txt") // an instantiation of fstream object` or `fstr.open("filename.txt") // after instantiation`

Files Streams in C++ (2)

- Read from files using:
  - `fstr >> var; // applying the stream extraction operator – stops at whitespace for strings`
  - `fstr.getline () // to read entire line into a character array`
    - Stored as a C string
- Write to files using:
  - `fstr << var; // applying the stream insertion operator`

A. O'Fallon, J. Hagemeister

Files Streams in C++ (3)

- Each file ends with an end-of-file marker (EOF)
  - check if at end of file using `fstr.eof()`
- Close a file using:
  - `fstr.close();`

A. O'Fallon , J. Hagemeister

Closing Thoughts on Files

- Files are required for many applications
- Files may be created and manipulated in any manner appropriate for an application

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (8th Ed.)*, Addison-Wesley, 2016

Collaborators

- Jack Hagemeister

(7 – 1) Classes: A Deeper Look
D & D Chapter 9

Instructor - Andrew S. O’Fallon

CptS 122 (February 21, 2018)

Washington State University

Washington State University

World Class. Face to Face.

Key Concepts

- Composition relationship
- `const` objects
- `const` member functions
- The "`this`" pointer

Composition Relationship

- A class can have objects of other classes as members – this is composition
- Composition is also referred to as a *has-a* relationship (we will not distinguish between composition and aggregation at this point)
  - For example: a car *has-an* engine, a pencil *has-an* eraser, etc.

A. O'Fallon, J. Hagemeister

`const` Objects

- Some objects need to be *mutable* and some do not (*immutable*)
  - A *mutable* object's attributes may be modified (given different values) after creation of the object
  - An *immutable* object's attributes have to be set during construction and cannot be modified later
    - Objects can be declared as immutable using keyword `const`
    - For example, consider a *ComplexNumber* with an imaginary and real part:

ComplexNumber c1(2.5, 3.0) // mutable

`const` ComplexNumber c2(4.5, 6.0); // immutable

A. O'Fallon, J. Hagemeister

`const` Member Functions

- Getter/accessor functions in most cases should be declared as `const` member functions
  – For example:

    double getRealPart () const; // declaration in ComplexNumber
- `const` member function cannot modify members of the object
  – They also *cannot* call functions that try to modify members of the object
- NOTE: `const` objects *cannot* call non-`const` member functions!!! However non-`const` objects can call `const` member functions

A. O'Fallon, J. Hagemeister

Copy Constructors for `const` Objects

- How do we copy a `const` object?
  - We could use a copy constructor where the argument is a reference to a `const` object
  - ComplexNumber (`const` ComplexNumber &copy);
- For example:

```
const ComplexNumber c2(4.5, 6.0); // immutable
ComplexNumber c3(c2); // invokes the copy constructor with the const argument
ComplexNumber c4 = c3; // will actually invoke the copy constructor, not overloaded
                       // assignment because we are constructing (instantiating)
                       // an object here!
```


A. O'Fallon, J. Hagemeister

The “`this`” Pointer (1)

- Every object has access to a *pointer* called keyword `this`
- It stores the address of the object
- The pointer is not part of the object itself, but is an *implicit* argument (passed by the compiler) to each of the object’s *non-static* member functions
- It can be used *explicitly* to reference data members in order to avoid name *conflicts*

A. O’Fallon, J. Hagemeister

The “`this`” Pointer (2)

- Let’s say we named one of the private data members of class ComplexNumber *realPart*:

  private:

  double realPart; // of course we’ll generally name **m**RealPart

- We want to create a setter for the *realPart*. We need to avoid *ambiguous* statements!:

  public:

  void setRealPart (double realPart)

  {

  realPart = realPart; // ambiguous statement!

  this->realPart = realPart; // use “ this” explicitly instead!

  }

A. O’Fallon, J. Hagemeister

Type of "`this`" Pointer

- The type is dependent on the type of object
- For a non-`const` member function of *ComplexNumber,* the `this` pointer type would be *ComplexNumber **
  - For a `const` member function, the `this` pointer type would be `const` ComplexNumber * -- meaning it could not be used to modify members of the object!

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister

(7 - 2) Operator Overloading
D & D Chapter 10

Instructor - Andrew S. O’Fallon
CptS 122 (February 23, 2018)
Washington State University

Key Concepts

- Keyword `operator`
- Operator overloading

A. O'Fallon, J. Hagemeister

What is Operator Overloading?

- A generalization of function overloading
- An extension to C++ standard operators to define how they should work with user-defined types such as classes

Why Overload Operators?

- Improves readability
- Allows for a more natural way to implement code

A. O'Fallon, J. Hagemeister

Rules and Restrictions on Operator Overloading

- The precedence of an operator cannot be changed
- The associativity of an operator cannot be changed, i.e. left-to-right or right-to-left
- The "arity" of an operator cannot be changed, i.e. if the operator accepts one operand (unary) or two operands (binary)
- Only existing operators may be overloaded

A. O'Fallon, J. Hagemeister

Which Operators Cannot be Overloaded?

- .
- .* (pointer to member)
- ::
- ?:

A. O'Fallon, J. Hagemeister

Recall Class ComplexNumber's Add () Function

- Let's write the definition for the `add()` member function

```
// Prototype: ComplexNumber add (const ComplexNumber &operand);

// Definition – notice the binary scope resolution operator
ComplexNumber ComplexNumber::add (const ComplexNumber &operand)
{
          // This adds the real part of the "operand" object
          // to the real part of the object that invokes the
          // call to the function; it also adds the imaginary
          // parts
          ComplexNumber result; // Declare local ComplexNumber

          // Recall we use the dot member operator (.) to access
          // members of a class; no dot (.) denotes accessing the
          // instantiated object's members; note we don't have to apply "special"
          // operators to access an object passed by reference!
          result.mRealPart = mRealPart + operand.mRealPart;
          result.mImaginaryPart = mImaginaryPart + operand.mImaginaryPart;
           // Don't want to pass back by reference; cause undefined behavior
          return result;
}
```

We Can Replace Add () by Overloading + (Using a Friend Function)

- Let's write a function to overload the binary + operator; this function is a *non-member* function, but is a friend of `ComplexNumber`

```
// Prototype: friend ComplexNumber operator+ (const ComplexNumber &lhs,
                                              const ComplexNumber &rhs);

// Definition - notice the operator is not preceded by ComplexNumber::, because
// it's a non-member operator. Made this function a friend of ComplexNumber
// to efficiently access the private data members of ComplexNumber.
ComplexNumber operator+ (const ComplexNumber &lhs, const ComplexNumber &rhs)
{
         ComplexNumber result; // Declare local ComplexNumber

         result.mRealPart = lhs.mRealPart + rhs.mRealPart;
         result.mImaginaryPart = lhs.mImaginaryPart + rhs.mImaginaryPart;

          // Don't want to pass back by reference; cause undefined behavior
         return result;
}
```

We Can Replace Add () by Overloading + (without Using a Friend Function)

- Let's write a function to overload the binary + operator; this function is a *non-member* function, but is NOT a friend of `ComplexNumber`

```
// Prototype: ComplexNumber operator+ (const ComplexNumber &lhs,
                                       const ComplexNumber &rhs);

// Definition – notice the operator is not preceded by ComplexNumber::, because
// it's a non-member operator. This function is NOT a friend of ComplexNumber.
// Need to use setters/getters now!
ComplexNumber operator+ (const ComplexNumber &lhs, const ComplexNumber &rhs)
{
         ComplexNumber result; // Declare local ComplexNumber

         result.setRealPart (lhs.getRealPart() + rhs.getRealPart());
         result.setImaginaryPart (lhs.getImaginaryPart() +
                                  rhs.getImaginaryPart());
          // Don't want to pass back by reference; cause undefined behavior
         return result;
}
```


A. O'Fallon, J. Hagemeister

Why Non-Member Overloaded Operators?

- Enables "symmetry" and *communitivity* among operators, i.e.
  - `ComplexNumber operator+ (const ComplexNumber &lhs, int rhs);`
  - `ComplexNumber operator+ (int lhs, const ComplexNumber &rhs);`
    - Important to make non-member operator+ when lhs is not a class! Since lhs is not an object in this case!

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Jack Hagemeister

(8 – 1) Container Classes & Class Templates
D & D Chapter 18

Instructor - Andrew S. O’Fallon
CptS 122 (February 26, 2018)
Washington State University

Key Concepts

- Class and block scope
- Access and utility functions
- Container classes
- Iterators
- Class templates

Class Scope and Accessing Class Members Explored Further (I)

- A class' data members (attributes) and member functions (operations) belong to the *class*' *scope*
- Nonmember functions do not belong to any class' scope; they are *global* namespace *scope*
- Within a class' scope data members are directly accessible by the member functions

A. O'Fallon, J. Hagemeister

Class Scope and Accessing Class Members Explored Further (II)

- Outside of the class' scope, public members are accessed through one of three different handles:
  - An object *name*, a *reference* to an object, or a *pointer* to an object
  - Note: the "this" pointer is considered an implicit handle available only within an object
- Local variables declared inside of a member function have *block* scope

A. O'Fallon, J. Hagemeister

Access Functions

- Functions that can read or display data are considered *access* functions
- *Predicate* functions are access functions that test a condition and return true or false; generally we append “is” to the front of the name of the function
  - isEmpty (), isFull(), etc.

Utility Functions

- A *utility* or *helper* function is a private member function used to support other member functions' operations

A. O'Fallon, J. Hagemeister

Container Classes (I)

- Classes designed to hold and organize a collection of other classes
  - Examples of *sequence* containers include: lists, vectors, etc.
  - Example of container *adapters* include: stacks, queues, etc.
    - Container adapters are adaptations or interfaces designed to restrict functionality for an already existing container – they provide a different set of functionality
    - The Standard Template Library (STL) stack and queue adapt the double-ended queue (deque)

A. O'Fallon, J. Hagemeister

Container Classes (II)

- Container classes are generally separated into four categories:
  - Sequence containers – represent *linear* data structures
    - Array, deque, list (doubly-linked), vector, forward_list (C++ 11)
  - Container adapters
  - Ordered associative containers – represent *nonlinear ordered* data structures
  - Set, multiset, map, multimap (CptS 223!)
  - Unordered associative containers – represent *nonlinear unordered* data structures

A. O'Fallon, J. Hagemeister

Properties of STL Sequence Containers (I)

- Array
  - Fixed size; direct access to any element
- Deque
  - Rapid insertions and deletions at front or back; direct access to any element
- List
  - Doubly linked list; rapid insertions and deletions anywhere

Properties of STL Sequence Containers (II)

- Vector
  - Rapid insertions and deletions at back; direct access to any element
- Forward_list
  - Singly linked list, rapid insertions and deletions anywhere; C++ 11

A. O'Fallon, J. Hagemeister

Properties of STL Container Adapters

- Stack
  - Last-in, first-out (LIFO)
- Queue
  - First-in, first-out (FIFO)
- Priority_queue
  - Highest priority element is always the first one out

A. O'Fallon, J. Hagemeister

Functions Common to Container Classes (I)

- *Default constructor – initializes* an *empty* container
- *Copy constructor* – initializes the container to be a *copy* of an *existing* container of the same type
- *Move constructor* – available in C++ 11 – *moves* the contents of an existing container into a new container of the same type without copying each element of the argument container

A. O'Fallon, J. Hagemeister

Functions Common to Container Classes (II)

- *Destructor* – performs house keeping or *cleanup* when container is no longer needed
- *Empty* – returns *true* if there are no elements in the container; *false* otherwise
- *Insert* – *inserts* an item into the container
- *Size* – returns the number of elements in the container

A. O'Fallon, J. Hagemeister

Functions Common to Container Classes (III)

- *Copy operator (=)* – copies the elements of one container into another container of the same type
- *Move operator (=)* – available in C++ 11 – *moves* the contents of one container into another without copying each element of the argument container
- *Max_size* – returns the *maximum* number of elements for a container

A. O'Fallon, J. Hagemeister

Functions Common to Container Classes (IV)

- *Begin – overloaded* to return an *iterator* that refers to the *first* element of the container
- *End - overloaded* to return an *iterator* that refers to the *next* position after the *end* of the container
- *Erase – removes* one or more elements from the container
- *Clear – removes all* elements from the container
- Others exist!

A. O'Fallon, J. Hagemeister

Iterators

- Similar properties to a *pointer*
- An *iterator* is any object that points to some element in a sequence of elements, and has the ability to iterate through the elements using ++ and indirection (*) operators
- *Containers* support the use of iterators

 A. O'Fallon, J. Hagemeister

Class Templates

- We have already seen function templates, we will now extend the idea to classes
- *Class templates* allow for a way to easily specify a variety of related overloaded functions (*function-template specializations*) or classes (*class-template specializations*)
- Allows for *generic* programming
- Keyword `template` denotes the start of a class template
- STL containers are “templated”

A. O’Fallon, J. Hagemeister

Example using Class Templates

- Developed during lecture – see code posted to schedule

A. O'Fallon, J. Hagemeister

Next Lecture..

- More about class templates, data structures, and containers

References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister

(9-3) Efficiency of Algorithms
D & D Chapter 20

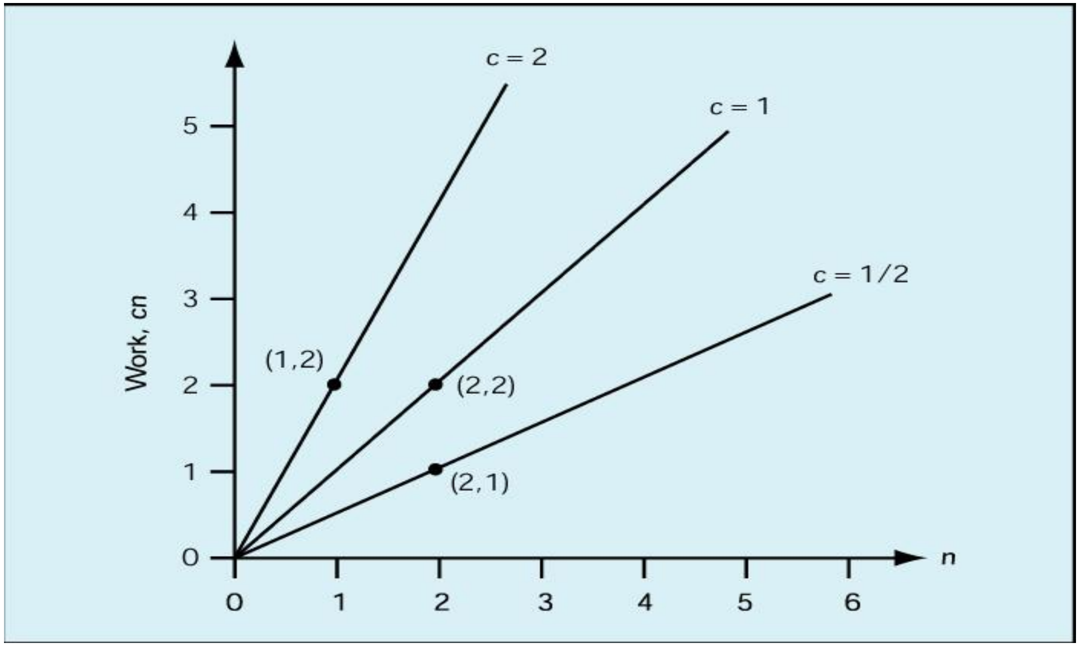
Instructor - Andrew S. O'Fallon
CptS 122 (March 9, 2018)
Washington State University

Analysis of Algorithms (1)

- In general, we want…
  - to determine central unit of work by considering the operations applied in the algorithm
  - to express unit of work as function of size of input data: How quickly does amount of work grow as size of input grows?
  - classify algorithms according to how their running *time* and/or *space* requirements grow as input size grows
- For example, recall Sequential Search algorithm
  - Get list of *n* names to search, and target name to search for
  - Examine each name in sequence
    - If all names have been examined, set found to false and stop
    - If name equals target, set found to true and stop
    - If name not equal to target, advance to next name
  - Main unit of work: *comparisons*
  - Analysis
    - In best case, one comparison must be made (target is first item in list)
    - In worst case, *n* comparisons must be made (target not found; all items examined)
    - In average case *n*/2 comparisons must be made

Analysis of Algorithms (2)

- Order of magnitude analysis ("Big-O")
  - Constant factors do not change shape of graph!



C. Hundhausen, A. O'Fallon

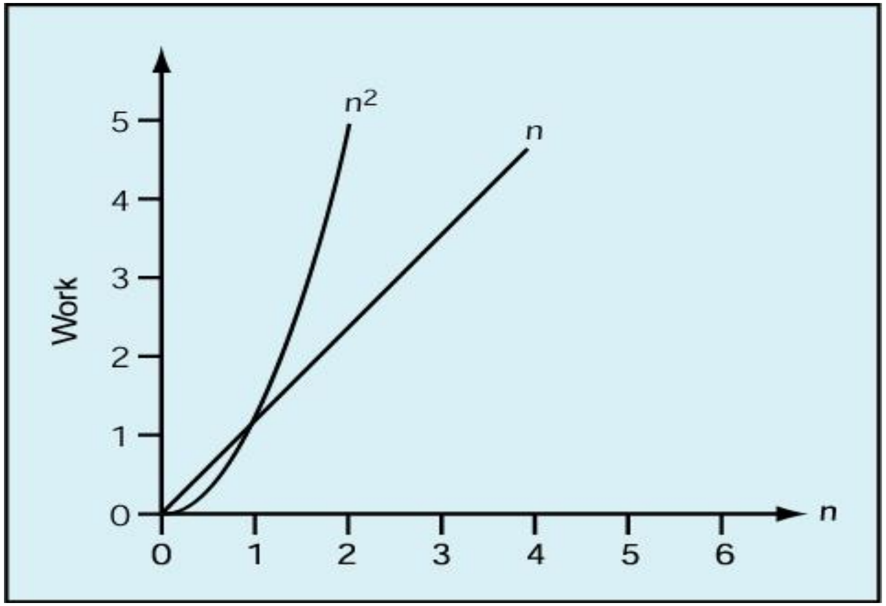
Analysis of Algorithms (3)

- Order of magnitude ("Big-O") (cont.)
  - Any algorithm whose work can be expressed as c * n where c is a constant and n is the input size is said to be "order of magnitude n", or O(n)
  - Likewise, any algorithm whose work varies as a constant times the square of the input size is said to be "order of magnitude n-squared", or $O(n^2)$

C. Hundhausen, A. O'Fallon

Analysis of Algorithms (4)

- Order of magnitude ("Big-O") (cont.)
  - $O(n^2)$ always gets bigger than $O(n)$ eventually!



C. Hundhausen, A. O'Fallon

Analysis of Algorithms (5)

- Big-O Analysis of Sequential Search
  - Best case: O(1)
  - Worst case: O(n)
  - Average case: O(n/2) = O(n)

Analysis of Algorithms (6)

- Recall Selection Sort…
  - *Input*: a list of numbers
  - *Output*: a list of the same numbers in ascending order
  - *Method*:
    - Set marker that divides “unsorted” and “sorted” sections of list to the end of the list
    - While the unsorted section of the list is not empty
      - Find largest value in “unsorted” section of list
      - Swap with last value in “unsorted” section of list
      - Move marker left one position

Analysis of Algorithms (7)

- Selection Sort (cont.)
  - Big-O Analysis
    - *Units of work*: comparisons and exchanges
    - In all cases, we need $n + (n - 1) + \ldots + 1$ comparisons = $[n * (n – 1)]/2$ comparisons = $1/2n^2 – 1/2n$ comparisons = $O(n^2)$ comparisons
    - In best case, items are already in order, so 0 exchanges needed: $O(n^2)$ comparisons + 0 exchanges = $O(n^2)$
    - In worst case, items are in reverse order, so n exchanges needed: $O(n^2)$ comparisons + n exchanges = $O(n^2)$

C. Hundhausen, A. O'Fallon

Analysis of Algorithms (8)

- Selection Sort (cont.)
  - Space Analysis
    - Major space requirement is list of numbers (n)
    - Other space requirements:
      - Extra memory location needed for marker between sorted and unsorted list
      - Extra memory location needed to store LargestSoFar used to find largest item in unsorted list
      - Extra memory location needed to exchange two values (why?)
      - Overall, space requirement is proportional to n.

Analysis of Algorithms (9)

- Recall Binary Search…
  - *Input*: a list of n sorted values and a target value
  - *Output*: True if target value exists in list and location of target value, false otherwise
  - *Method*:
    - Set startindex to 1 and endindex to n
    - Set found to false
    - While found is false and startindex is less than or equal to endindex
      - Set mid to midpoint between startindex and endindex
      - If target = item at mid then set found to true
      - If target < item then set endindex to mid – 1
      - If target > item then set to startindex to mid + 1 pointSet marker that divides "unsorted" and "sorted" sections of list to the end of the list
    - If found = true then print "Target found at location mid"
    - Else print "Sorry, target value could not be found."

Analysis of Algorithms (10)

- Binary Search (cont.)
  - Big-O Analysis
    - Unit of work: comparisons
    - Best case
      - target value is at first midpoint
      - O(1) comparisons
    - Worst case
      - target value is not found
      - list is cut in half until it is reduced to a list of size 0 (startindex is greater than or equal to endindex)
      - How many times can the list be cut in half? The number of times a number n is divisible by another number m is defined to be the logb(a), so the answer is $\log_2(n)$ = **O(lg n)**

C. Hundhausen, A. O'Fallon

Analysis of Algorithms (11)

		n			
Order		10	50	100	1000
lg *n*		0.0003 sec	0.0006 sec	0.0007 sec	0.001 sec
n		0.001 sec	0.005 sec	0.01 sec	0.1 sec
n^2		0.01 sec	0.25 sec	1 sec	1.67 min
2^n		0.1024 sec	3570 yrs	4 * 1016 centuries	Too big to compute

Summary of Orders of Magnitude

- $O(\lg n)$ = flying
- $O(n)$ = driving
- $O(n^2)$ = walking
- $O(n^3)$ = crawling
- $O(n^4)$ = barely moving
- $O(n^5)$ = no visible progress
- $O(2^n)$ = forget it, it will never happen

References

- P.J. Deitel & H.M. Deitel, *C++ How to Program (9th Ed.)*, Pearson Education , Inc., 2014.
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

Collaborators

- Chris Hundhausen