

CSCI-1200 Computer Science II — Spring 2006

Lecture 2 — Background II

Announcements

- HW 1 is available on-line through the course website.
- If you have not resolved issues with the C++ environment on your laptop, please do so immediately.
- Administrative questions?

Overview

- A fourth introductory example
- Functions and parameter passing
- While loops
- Scope
- Order notation: *This material will not be covered completely. What is not covered is left for homework reading. We will be more interested in the intuitions underlying the order notation than in its formal definition and use.*

2.1 Example: Julian Date to Month/Day

- The source code is attached to these notes. We'll examine some of the structure and then focus on the `month_and_day` function
- The `main` function is the first function in the file as opposed to the last. This is a stylistic choice.
- Function prototypes are inserted above the main function in the file.
- The name of the month is output using the function `output_month_name`, which is just a big `switch` statement.
 - Note the use of the `break` statement at the end of each `case`.
 - See Chapter 4 of Malik for detailed discussion of `switch`.

2.2 Month And Day Function

```
// Compute the month and day corresponding to the Julian day within
// the given year.
void month_and_day( int julian_day, int year, int & month, int & day ) {
    bool month_found = false;
    month = 1;

    // Loop through the months, subtracting the days in this month
    // from the Julian day, until the month is found where the
    // number of remaining days is less than or equal to the total
    // days in the month.
    while ( !month_found ) {
        // Calculate the days in this month by looking it up in the
        // array. Add one if it is a leap year.
        int days_this_month = DaysInMonth[month];
```

```

    if ( month == 2 && is_leap_year(year) )
        ++ days_this_month;

    if ( julian_day <= days_this_month )
        month_found = true;    // Done!
    else {
        julian_day -= days_this_month;
        ++ month;
    }
}
day = julian_day;
}

```

- We'll assume you know the basic structure of a `while` loop and focus on the underlying logic.
- A `bool` variable (which can take on only the values `true` and `false`) called `month_found` is used as a flag to indicate when the loop should end.
- The first part of the loop body calculates the number of days in the current month (starting at one for January), including a special addition of 1 to the number of days for a February (`month == 2`) in a leap year.
- The second half decides if we've found the right month.
- If not, the number of days in the current month is subtracted from the remaining Julian days, and the month is incremented.

Understanding the logic of functions such as this one is important for developing your programming skills.

2.3 Value Parameters and Reference Parameters

Consider the line in the main function that calls `month_and_day`:

```
month_and_day( julian, year, month, day_in_month );
```

and consider the function prototype:

```
void month_and_day( int julian_day, int year, int & month, int & day )
```

Note in particular the `&` in front of the third and fourth parameters.

- The first two parameters are *value parameters*.
 - These are essentially local variables (in the function) whose initial values are copies of the values of the corresponding argument in the function call.
 - Thus, the value of `julian` from the main function is used to initialize `julian_day` in function `month_and_day`.
 - Changes to value parameters do NOT change the corresponding argument in the calling function (`main` in this example).
- The second two parameters are *reference parameters*, as indicated by the `&`.
 - Reference parameters are just aliases for their corresponding arguments. No new variable are created.
 - As a result, changes to reference parameters are changes to the corresponding variables (arguments) in the calling function.

- “Rules of thumb” for using value and reference parameters:
 - When a function (e.g. `is_leap_year`) needs to provide just one result, make that result the return value of the function and pass other parameters by value.
 - When a function needs to provide more than one result (e.g. `month_and_day`, these results should be returned using multiple reference parameters.

We will see minor variations on these rules as we proceed this semester.

2.4 Arrays as Function Arguments

What does the following function do?

```
void do_it (double a[], int n) {
    for (int i = 0; i < n; ++i)
        if (a[i] < 0) a[i] *= -1;
}
```

- The important point about this function is that the changes made to array `a` are permanent, even though `a` is a value parameter!
- Reason: What’s passed by value is the memory location of the start of the array. The entries in the array are not copied, and therefore changes to these entries are permanent.
- The number of locations in the array to work on — the value parameter `n` — must be passed as well because arrays have no idea about their own size.

2.5 Exercises

1. What would be the output of the above program if the main program call to `month_and_day` was changed to

```
month_and_day( julian, year, day_in_month, month );
```

and the user provided input that resulted in `julian == 50` and `year == 2006`? What would be the additional output if we added the statement

```
cout << julian << endl;
```

immediately after the function call in the main function?

2. What is the output of the following code?

```
void swap( double x, double &y ) {
    double temp = x;
    x = y;
    y = temp;
}

int main() {
    double a = 15.0, b=20.0;
    cout << "a = " << a << ", b= " << b << endl;
    swap ( a, b );
    cout << "a = " << a << ", b= " << b << endl;
    return 0;
}
```

2.6 Scope Example

The following code will not compile. We want to understand why not, fix the code (minimally) so that it will, and then determine what will be output.

```
int main() {
    int a = 5, b = 10;
    int x = 15;
    {
        double a = 1.5;
        b = -2;
        int x = 20;
        int y = 25;
        cout << "a = " << a << ", b = " << b << '\n'
             << "x = " << x << ", y = " << y << endl;
    }
    cout << "a = " << a << ", b = " << b << '\n'
         << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

2.7 Scope

- The *scope* of a name (identifier) is the part of the program in which it has meaning.
- Curly braces, { }, establish a new scope — this includes functions and compound statements.
 - This means scopes may be nested.
 - Identifiers may be re-used as long as they are in different scopes.
- Identifiers (variables or constants) within a scope hide identifiers within an outer scope having the same name. This does not change the values of hidden variables or constants — they are just not accessible.
- When a } is reached, a scope ends. All variables and constants (and other identifiers) declared in the scope are eliminated, and identifiers from an outer scope that were hidden become accessible again in code that follows the end of the scope.
- The operator :: (namespaces) establishes a scope as well.

2.8 Algorithm Analysis: What, Why, How?

- What?
 - Analyze code to determine the time required, usually as function of the size of the data being worked on.
- Why?
 - We want to do better than just implementing and testing every idea we have.
 - We want to know why one algorithm is better than another.
 - We want to know the best we can do. (This is often quite hard.)
- How? There are several possibilities, including:
 - Don't do any analysis; just use the first algorithm you can think of that works.

- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

2.9 Exercise: Counting Example

Suppose `foo` is an array of `n` doubles, initialized with a sequence of values.

- Here is a simple algorithm to find the sum of the values in the vector:

```
double sum = 0;
for ( int i=0; i<n; ++i )
    sum += foo[i];
```

- How do you count the total number of operations?
- Go ahead and try. Come up with a function describing the number of operations.
- You are likely to come up with different answers. How do we resolve these differences?

2.10 Which algorithm is best?

An venture capitalist is trying to decide which of 3 startup company to invest in and has asked for your help. Here’s the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the “best” algorithm?

2.11 Order Notation

The following discussion emphasizes intuition. That’s all we care about in CS II. For more details and more technical depth, see any textbook on data structures and algorithms.

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- As a result, algorithms requiring $3n + 2$, $5n - 3$, $14 + 17n$ operations are all $O(n)$ (i.e. in applying the definition of order notation $f(n) = n$).
- Algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$ (i.e. in applying the definition of order notation $f(n) = n^2$).
- Intuitively (and importantly), we determine the order by finding the asymptotically dominant term (function of n) and throwing out the leading constant. This term could involve logarithmic or exponential functions of n .
- Implications for analysis:

- We don't need to quibble about small differences in the numbers of operations.
 - We also do not need to worry about the different costs of different types of operations.
 - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

2.12 Common Orders of Magnitude

Here are the most commonly occurring orders of magnitude in algorithm analysis.

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC*. e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR*. e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^2)$, $O(n^k)$, *a.k.a. POLYNOMIAL*. e.g., find closest pair of points.
- $O(2^n)$, *a.k.a. EXPONENTIAL*. e.g., Fibonacci, playing chess.

2.13 Back to Analysis: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`

```
int loc=0;
bool found = false;
while (!found && loc < n) {
    if (x == foo[loc])
        found = true;
    else
        loc ++ ;
}
if (found) cout << "It is there!\n";
```

- Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

2.14 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size vector, we might want to know:
 - The fewest number of operations (best case) that might occur.
 - The average number of operations (average case) that will occur.
 - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

2.15 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the “size” of the problem.
 - For arrays and other “container classes” this will generally be the number of values stored.
- Decide what to count. The order notation helps us here.
 - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
 - We might also count specific operations, such as comparisons.
- Do the count and use order notation to describe the result.

2.16 Exercises: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of n :

1.

```
int count=0;
for ( int i=0; i<n; ++i )
    for ( int j=0; j<n; ++j )
        ++count;
```
2.

```
int count=0;
for ( int i=0; i<n; ++i )
    ++count;
for ( int j=0; j<n; ++j )
    ++count;
```
3.

```
int count=0;
for ( int i=0; i<n; ++i )
    for ( int j=i; j<n; ++j )
        ++count;
```

2.17 Summary

- `switch` statements and `while` loops
- Value parameters and reference parameters
- Arrays as function parameters
- Scope
- Order notation