Background

Overview

• A fourth introductory example
• Functions and parameter passing
• While loops
• Order notation
• Scope

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`. 
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
  // 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 (\texttt{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.

**Value Parameters and Reference Parameters**

Consider the line in the main function that calls \texttt{month\_and\_day}

\begin{verbatim}
  month_and_day( julian, year, month, day_in_month );
\end{verbatim}

and consider the function prototype

\begin{verbatim}
  void month_and_day( int julian_day, int year,
                      int & month, int & day )
\end{verbatim}

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

• The first two parameters are value parameters.
  – These are essentially local variables whose initial values are copies of the values of the corresponding argument in the function call.
  – Thus, the value of \texttt{julian} from the main function is used to initialize \texttt{julian\_day} in function \texttt{month\_and\_day}.
  – Changes to value parameters do NOT change the corresponding argument in the calling function (\texttt{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.

Arrays as Function Arguments

Consider the following function

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

• What does it do?

• 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.
Exercises

We will work on the following exercises in class

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 == 2005`? 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;
   }
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:
  1. Don’t do any analysis; just use the first algorithm you can think of that works.
  2. Implement and time algorithms to choose the best.
  3. Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
  4. 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.

- What Happens In Practice?
  - 99% of the time: rough count similar to #4 as a function of the size of the data. Use order notation to simplify the resulting function and even to simplify the analysis that leads to the function.
  - 1% of the time: implement and time.

What follows is a quick review of counting and the order notation.
**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:

  ```java
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?

**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 \times 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.

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

• $O(1)$: The number of operations is independent of the size of the problem.
• $O(\log n)$
• $O(n)$
• $O(n \log n)$
• $O(n^2)$
• $O(n^2 \log n)$
• $O(n^3)$
• $O(2^n)$

Significance of Orders of Magnitude
• On a computer that performs $10^8$ operations per second:
  – An algorithm that actually requires $15n \log n$ operations requires about 3 seconds on a problem of size $n = 1,000,000$, and 50 minutes on a problem of size $n = 100,000,000$. 

8
– An algorithm that actually requires \( n^2 \) operations requires about 3 hours on a problem of size \( n = 1,000,000 \), and 115 days on a problem of size \( n = 100,000,000 \).

• Thus, the leading constant of 15 on the \( n \log n \) does not make a substantial difference. What matters is the \( n^2 \) vs. the \( n \log n \).

• Moreover, in practice the leading constants usually do not vary by a factor of 15.

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 \( \text{foo} \)

```cpp
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 \( \text{if} \) statement? What did you assume about where the value stored in \( x \) occurs in the array (if at all)?

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 \( O(n) \).
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, using order notation to describe the result.

Examples: Loops

In each case give an order notation estimate as a function of \( n \) which here does note

- Version A:

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

- Version B:

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

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

- Version C:
int count=0;
for ( int i=0; i<n; ++i )
    for ( int j=i; j<n; ++j )
        ++count;

- How many operations in each?

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.

```c
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 << \nendl;
    }
    cout << "a = " << a << " b = " << b << " \n"
        << "x = " << x << " y = " << y << \nendl;
    return 0;
}
```

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.

Summary

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