Organizational

- Reading: Chapter 7
- Reminder: project due on Monday 10/25 (noon)
- Agenda on web has been updated
- Midterm will be returned next week (graded on curve)

Software engineering (coding) standards

- Standards for you
- Standards for others
- Matching design with implementation
Example: control structures

benefit = minimum;
if (age < 75) goto A;
benefit = maximum;
goto C;
if (AGE < 65) goto B;
if (AGE < 75) goto C;
A: if (AGE < 65) goto B;
benefit = benefit * 1.5 + bonus;
C: next statement
if (age < 55) benefit = minimum;
elseif (AGE < 65) benefit = minimum + bonus;
elseif (AGE < 75) benefit = minimum * 1.5 + bonus;
else benefit = maximum;

Algorithms

■ Efficiency may have hidden costs
■ cost to write the code faster
■ cost to test the code
■ cost to understand the code
■ cost to modify the code

Keep the program simple (1)

For the first $10,000 of income, the tax is 10 percent.
For the next $10,000 of income above $10,000, the tax is 12 percent.
For the next $10,000 of income above $20,000, the tax is 15 percent.
For the next $10,000 of income above $30,000, the tax is 18 percent.
For any income above $40,000, the tax is 20 percent.

```java
int tax;
if (taxable_income == 0)
    goto EXIT;
if (taxable_income > 10000)
    tax = tax + 1000;
else{
    tax = tax + 0.10*taxable_income;
    goto EXIT;
}
if (taxable_income > 20000)
    tax = tax + 1200;
else{
    tax = tax + 0.12*(taxable_income-10000);
    goto EXIT;
}
if (taxable_income > 30000)
    tax = tax + 1500;
else{
    tax = tax + 0.15*(taxable_income-20000);
    goto EXIT;
}
if (taxable_income < 40000){
    tax = tax + 0.18*(taxable_income-30000);
    goto EXIT;
}
else
    tax = tax + 1800.0 + 0.20*(taxable_income-40000);
exit:
```
Keep the program simple (2)

Table 7.1. Sample tax table.

<table>
<thead>
<tr>
<th>Bracket Base</th>
<th>Base</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>2200</td>
<td>12</td>
</tr>
<tr>
<td>20,000</td>
<td>3700</td>
<td>15</td>
</tr>
<tr>
<td>30,000</td>
<td>5500</td>
<td>18</td>
</tr>
<tr>
<td>40,000</td>
<td>5500</td>
<td>18</td>
</tr>
</tbody>
</table>

for (int i=2; level=1; i <= 5; i++)
  if (taxable_income > bracket[i])
      level = level + 1;
  tax = base[level] + percent[level] * (taxable_income - bracket[level]);

General guidelines

- Localize input and output
- Include pseudocode
- Revise and rewrite, rather than patch
- Reuse

Documentation

- Internal documentation
  - header comment block
  - other program comments
  - meaningful variable names and statement labels
  - format to enhance understanding
  - document data
- External documentation
  - describe the problem
  - describe the algorithm
  - describe the data
Writing high-quality code

- Good code must be based on good design
  - choose right data & control structures
  - choose appropriate programming language(s)
  - program into a language not in it:
    - use your own conventions, standards & class libraries if language does not provide what you need
  - most programming principles are independent of the specific language but depend on how you use it

Good Construction Practices

- Start with a good design
  - decide how much (little!) to do at keyboard
  - set coding conventions for names, comments, layout
  - define how to handle errors, security, class interface standards, reuse standards, performance
  - program into not in

OO: Designing Classes

- Foundations: ADTs
- Interfaces
- Implementation issues
- When to create a class?
- Language dependency
- Packages
Think ADTs first - program into!

- **Benefits**
  - hide implementation details
  - changes will be localized
  - interface can be more informative (e.g., not `currentFont.size = 16` but use an ADT size in pixels or points)
  - easier to improve performance
  - correctness is more obvious
  - more self-documenting
  - less coupling (no need to pass data everywhere)
  - deal with high-level (real world) entities not low implementation details

---

Good Class Interfaces

- Present a consistent level of abstraction, e.g., not:
  ```
  class EmployeeCensus: public ListContainer {
  public:
    ...
    void AddEmployee(Employee employee);
    void RemoveEmployee(Employee employee);
    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
  }
  ```

Good Class Interfaces (2)

- Understand the abstraction
  - provide the right one not more not less
  - provide services in pairs with opposites
  - move unrelated information to another class
  - make interface programmatic not semantic
    - interface attributes (like types)
    - not invariants like routine A must be called before routine B
  - beware of interface abstraction erosion
    - class modifications may obfuscate the abstraction
Good Class Interfaces (3)
- don’t add public members that are inconsistent with the interface abstraction
- consider cohesion and abstraction in tandem
- both typically go together

Good Class Interfaces (4)
- Good encapsulation
  - minimize accessibility
  - don’t expose member data publicly
  - don’t put private implementation details into a class interface

Example
```cpp
class Employee {
  ...
private:
  String m_Name;
};
class Employee {
  ...
private:
  EmployeeImplementation *m_impl;
};
```
### Implementation Issues
- **Containment**
  - “has a” relationship
- **Inheritance**
  - “is a” relationship
- Inherit only what you want to inherit
- Avoid deep inheritance trees
  - create complexity

### Member Functions & Data
- Keep # routines small
  - higher # associated with higher fault rates (Basili et al, 1996)
  - disallow implicitly generated member functions and operators you don’t want (make them private)
  - minimize # of routines called by a class
    - again more faults (Basili et al, 1996)
- Demeter principle: minimize indirect calls to other classes:
  - account.ContactPerson(), DaytimeContactInfo.PhoneNumber()
  - Demeter = Greek goddess of agriculture; grow software in small steps

### Constructors
- Initialize all member data in constructors, if possible
  - defensive programming
- Enforce singleton property by using a private constructor (if you want to use a singleton)
Singleton Java Example

```java
public class MaxId {
    // constructors and destructors
    private MaxId() { ... }
    // public routines
    public static MaxId GetInstance() {
        return m_instance;
    }
    // private members
    private static final MaxId m_instance = new MaxId();
    ...
}
```

When to Create a Class?
- model real world objects
- model abstract objects
- isolate complexity
- hide implementation details
- limit effect of changes
- hide global data
- streamline parameter passing
- make central points of control
- facilitate reusable code

How to Write good Routines
- reasons to create a routine
- routine design
- good routine names
- how long should it be
- how to use parameters
- special cases
- macros and inline routines
When to create a routine

- reduce complexity
  - "procedural abstraction"
- avoid duplication ("code factoring")
- support subclassing
  - easier for short, well-designed routines
- hide sequencing
- hide pointer operations
- improve portability
- simplify boolean tests
- improve performance
  - can recode with better algorithm

Design at Routine Level

- Functional cohesion
  - routine performs exactly one function
- Good routine names
  - describe everything it does
  - avoid meaningless names (e.g., dealWithInput())
  - to name a function, use description of return value
  - to name a procedure, use verb followed by an object, e.g., PrintDocument()
- Use precise opposites
  - add / remove, begin / end, open / close etc.
- Establish conventions for common names
  - not employee.id.Get() and dependent.GetId()
Routine Size

- No unequivocal evidence that large size increases errors or vice versa
- Rule of thumb
  - no more than 200 lines
  - if longer, maybe can factor out code

Routine Parameters

- Order them
  - eg. IN - IN/OUT - OUT
- Use same order across routines with similar parameters
  - bad example C: printf / fprintf (file argument first) but puts / fputs (last)
- Use all parameters
  - unused parameters correlated with increased errors (Card et al., 1986)
- Put status / error variables last
- Don't use parameters as work variables

Routine Parameters (2)

- Document interface assumptions about parameters
- Limit the # of parameters (7 is a rule thumb)
- Establish a naming convention for IN / IN-OUT / OUT parameters
  - e.g., input_inout_out_
- Make sure actuals match formals
Special Considerations

- Functions versus procedures
  - primary purpose is to return a value, then use a function, otherwise a procedure
- Setting the return value
  - don't return references or pointers to local data
  - make sure value is set on all paths

Macros & Inline Routines

- Fully parenthesize macros
  - #define Cube(a) a*a*a will break: Cube (x+1)
  - #define Cube(a) (a)* (a) *(a) still incorrect!
  - #define Cube(a) ((a)* (a) *(a))
- Use {} for multiple-line macros
  - use only sparingly, use procedures instead
- Use inline routines sparingly
  - violate encapsulation
  - turn on interprocedural optimization instead (i.e., use a good compiler)