COSC334 Data Structures

Dr. Matt Evett
Eastern Michigan University
Fall 1999
Principles of Programming

- Problem Solving and Software Engineering
- Achieving a Modular Design
- Key Issues in Programming
What is Problem Solving?

- Problem Statement.
- Understand Problem and Analyze.
- Develop solution process.
- Implement the Solution.
- Test and debug.
Software Life Cycle

- Specification
- Design
- Risk Analysis
- Verification
- Coding
- Testing
- Refining the solution
- Production
- Maintenance
Specification

Make the problem statement precise and detailed

- What is the input? What is the output? What is the relationship between input and output? Is the program interactive? User interface? Will the program be used often, or only a few times? How much error control is needed?

Prototype programs can clarify the problem.

- Simulation of the actual program to validate the interface, identify potential problems, etc.
Design

- Divide the program into manageable modules.
- Module can be a single fn or a collection of fns.
- Loosely coupled modules are independent.
- Each module performs one well-defined task.
- For each module, specify its purpose, assumptions, input & output.
- Also specify the data flow among modules.
Design

 kaz Module specification is like a contract between the module designer and the module user; both should obey the rules of the contract or the sky will fall!
 kaz Module specification should not say how it’s done
 kaz Module specification might include pre- and post-conditions for each function.
 kaz Precise documentation is essential.
 kaz Use existing software components in design.
Design Example

First Draft

Sort (A, N)
// Sorts an array
// Precondition: A is an array of N integers, N>0
// Postcondition: The integers in A are sorted.
Sort(A,N)
// Sorts an array into ascending order.
// Pre: A is an array of N integers &
//     1 <= N <= MAX
//     where MAX is the maximum size of A.
//       N is unchanged.
// output A[] is a permutation of input A[]
Risk Analysis

Some risks are predictable and others are not.

Risks can affect a project’s timetable or cost or the people.

Techniques exist to identify, assess, and manage risk.
Verification

♫ Formal methods help to prove that an algorithm is correct.
♫ Assertions can be made at various points in an algorithm.
♫ Loop invariant can be used with looping statements.
♫ Mathematical induction is also useful.
♫ Empirical methods or a hybrid can also be used.
Coding

❖ Minor phase of software life cycle.
❖ Translate the algorithms into programs.
❖ Use dummy functions (*stubs*).
❖ Top down implementation.
❖ Bottom up implementation
Testing

Goal: Remove logical errors.

Bottom-up testing.

- Test low level modules first.

Test with valid data first.

Test with extreme values.

- Example: N = 0 or N = MAX.

Test with invalid data.
Testing

- Try random test data. Try actual data.
- A test case covers only part of a program.
- Testing is an art. It is not an easy problem.
- It may be impossible to do exhaustive test.
  - Exponential number of test cases.
- Expect users to report bugs later.
Refining the solution

- First, make program work on simple input.
- Refine by allowing other inputs.
- Add more errors checks.
- Replace stubs with actual code.
- Use better algorithms for some modules.
Refining the solution

- Enhance the features to satisfy all the problem requirements.
- Modularity should help in adding these refinements.
- After the refinement, thoroughly test the program again even if the changes seem minor.
Production

- Package the software.
  - Simplifies installation by avoiding different implementations.
- Use installation wizards when possible.
  - Reduces risk of errors during installation
- Distribute to users.
- Install on user computers.
Maintenance

❖ Users will report bugs.
❖ Users will want additional features.
❖ Better data structures can be used.
❖ Better algorithms can be used.
❖ Make the software more efficient
Maintenance

- Better user interface.
- Better error handling.
- Make it more user friendly.
- Make it more intelligent.
- Update documentation.
- Charge more and make more money!!!
What is a good solution?

How much does it cost?

- Computer Resources (Memory, Time)
- Development time
- Cost of maintenance
- Readability (new employees need to catch up)
What is a good solution?

✍ Is it easy to upgrade?
  - Adding new features
  - More error checks
  - Interfacing with other programs

✍ Efficiency is only one factor in cost.

✍ Program must be well structured and documented.
Achieving Modular Design

- Abstraction and Information Hiding
- Object Oriented Design
- Top-Down Design
- General Design Guidelines.
Abstraction

- Separation of the purpose of a module from its implementation.
- Modularity breaks a solution into modules.
- Abstraction specifies each module clearly before implementation.
- Abstraction helps to focus on high level design of the problem.
- Helps to modify one module without significantly affecting others.
Functional Abstraction

- Separation of the purpose of a function from its implementation.
- Once a function specification is known, you can use it without knowing how it is implemented.
- Essential to team projects.

Sort(A,N) - Sorts array A.
Data Abstraction

❖ Collection of data with operations defined on them such as add, delete, search etc.
❖ Focus on what the operations do instead of how they do it.
❖ Other modules know supported operations and their purpose.
❖ Other modules do not know how data is organized or how operations are implemented.
Data Abstraction (contd)

ADT (Abstract Data Type) is a collection of data items with operations defined on them. ADT might impose some integrity constraints on the data item which are preserved by the operations. ADT + Algorithms = Programs. Client - ADT User. Server - ADT Designer.
Information Hiding

Abstraction, Function Abstraction, Data Abstraction.

Abstraction helps to identify details that should be hidden from the clients. Private.

Abstraction helps to identify details that are relevant for the client. Public.

Private details should be hidden from clients.
Information Hiding (contd)

Not only hide the details within a module, but also ensure that other modules (clients) cannot tamper with these hidden details.

The implementer of a module does not worry about its use.

The user of a module does not worry about the details and implementation.
Object-Oriented Design (OOD)

- Objects = Data + Operations
- OOD involved identification of objects and their relationship.
- Objects have behaviors.
- Objects support *encapsulation*.
- Functions encapsulate actions.
- Objects encapsulate data and actions.
OOD - An example

A Digital Clock

☞ Object - Clock
  ● Operations:
    Set time. Advance time. Display time.

☞ Object - Indicator (Hour & Minute)
  ● Operations:
    Set value. Advance value. Display value.
OOD Example (contd)

- Identify class of objects
  - Clock Class
  - Indicator Class
- Clock Object is an instance of Clock class.
- Hour and Minute indicators are instances of the indicator class.
- Clock object contains two indicator objects.
Inheritance

❖ OOD supports object class hierarchy.
❖ Sub-classes *inherit* data and operations from super classes.
❖ Example: Alarm Clock is a Clock.
❖ Sub-class can have new data & operations.
❖ Inherits supports *Class Reuse*. 

COSC334 Data Structures 32
Polymorphism

- Inheritance makes operation identification hard.
- Function and Operator overloading adds complexity too.
- The determination may depend on info that is available only at execution time.
- Polymorphism allows this determination to be done at execution time.
Top-Down Design

OOD identifies objects by nouns.

Top-down design identifies operations by verbs.

Address a task at successively lower levels of details.

Structure Chart shows the operation hierarchy.
Top-Down Design (Contd)

- Focus on “what” a module does.
- Do not focus on “how” the module does it.
- Identify sub-modules & build the hierarchy.

Diagram:

Find Median

Read Scores
- Prompt User
- Store in Array

Sort Scores

Get Middle
Modularity

- Easy to manage. Allows team programming
- Debugging is much easier
  - Isolate errors to individual modules.
- Modular programs are easy to read.
- Minor changes are isolated to few modules.
Modifiability

Functions makes modifications easier.

Named constants facilitates modifiability

- const int NUM_MAJORS = 202;
- int scores[NUM_MAJORS];
- for (i = 0; i < NUM_MAJORS; i++)

typedef also renders modifications easier.

- typedef float Weight;
- typedef double Weight;
Ease of Use

- Clear and unambiguous user prompts.
- Echo the input whenever possible.
- Label outputs clearly.
- Design a good user interface.
Issues in Programming Style

- Extensive use of functions.
- Avoid Globals (tightly-coupled functions)
- Proper use of function arguments
  - Pass by value (actual not changed)
  - Pass by reference (actual is changed)
  - const pass by reference (copying is expensive)
- Proper use of functions.
- Error Handling.
Review of C++

- Edit - Compile - Error - Edit - Compile
- Test - Debug - Edit - Compile
- Header Files (.h files)
- Source Files (.cc or .cpp files)
- Object Files (.o or .obj files)
- Executable Files. (.exe files)
- Library Files (.a or .lib files)
  - Collection of Object files. (Static or Dynamic)
C++ Fundamentals

Pre-processor statements

- `#include <filename>`
  - Always use `<>` notation.
  - Use compiler option `-I<dir>` to specify location.
- `#define <name> <value>`  -- Avoid in C++.
- `#if defined(<name>) <body> #endif`
  - Useful to guard against multiple inclusion.
  - Conditional compilation for debugging
  - For machine specific compilation.
Comments

✎ C++ allows two styles of comments

- // <comment>. Preferred when possible.
- /* <comment> */. Nesting is a problem.

✎ Each file (.h or .cpp) should have comment at start of the file. (Filename, Author, etc)

✎ Before each function.

✎ Before each logical segment of code
Fundamental Data Types

- `bool` -- true, false  `#include <bool.h>`
- `char`, `unsigned char`
- `short int`, `int`, `long int` (unsigned versions too)
- `float`, `double`, `long double`

```c
bool done; // useful for loop.
int id;    // student id
```
Reference Variables

These are variables with no storage on their own but serve as aliases for existing ones.

Useful for passing parameters, as return values etc.

```c
int n;
int &m = n;
n = 5;
m = 7; // changes n.
```
Variable Names

- Choose meaningful names (must)
- Long names are preferable to abbreviations.
- Start with lowercase (deviation from book)
- Subsequent words are capitalized.

```c
int      maxSoFar;
bool     isListEmpty;
int      count; // Count of # of items in list.
double   epsilon = 0.000001; // small value
```
Named Constants

Choose meaningful names even if long.
All uppercase with _ separating words.
Use const statement. Must initialize.
Avoid #define statement.
You can also use enum.
Put each enum constant on a new line.
  • Facilitates comments for each constant.
Named Constants (contd)

const int MAX_FILENAME_SIZE = 50;
enum Days
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};
Type and Class Names

✍ Choose meaningful names even if long.
✍ All words are capitalized.
✍ Avoid type definition for array.
  ● typedef int Scores[MAX_SCORES];
  ● Scores studentScores;
  ● Book does it. We should avoid.
✍ Class Names are like types too.
  ● Book uses names starting with lowercase.
Expressions

Use white space for clarity.

- $i \cdot j - k / 2$  vs  $i \times j - k / 2$

Arithmetic Operators

- $+ - \ast / \%$ (remainder)

Relational Operators

- $< \leq \equiv \geq > \neq$

Logical Operators

- $\| \&\& \!$ (unary operator)
Assignments and Expressions

❖ Short-circuit evaluation
  • As soon as value is known, rest skipped.
  • (5 == 4) && (A < B)
  • (5 == 5) || (A < B)

❖ Operators have precedence & associativity.

❖ Assignment Statement: <var> = <expr>

❖ If there are several assignment statements one after another, align them for better look.
Assignments (contd)

Avoid multiple assignments. $x = y = 0$;
maxSoFar = a[0];
count = 0;
done = false;

Versus
maxSoFar = a[0];
count = 0;
done = false;
Type Conversion

Sometimes it is necessary to convert one type of value to another (casting).

Implicit casting (compiler does it automatically)
- smaller type is promoted to larger one.

Explicit casting (programmer indicated)
syntax: type(expression)
double volume; int(volume)
Other Operators

C++ provides other convenient operators

\[ a += b; \] same as \[ a = a + b; \]
\[ -= *= /= %= ||= &&= \]
\[ |= \text{(bit or)} \quad \&= \text{(bit and)} \]

Pre-increment: \[ c = ++a; \]

Post-increment: \[ d = a++; \]

\[ c = --a; \quad d = a--; \]
Input and Output

```cpp
#include <iostream.h>

Input operator is `>>`. Use stream `cin`.
Output operator is `<<`. Use stream `cout`.
There is no need for format information.

```cpp
int n; double p;
cin >> n >> p;
cout << "n = " << n << endl;
```
Input and Output (contd)

- *cin* skips white space characters.

- To read the next character no matter what it is, you can use the function *get*.
  - `cin.get(nextChar);`
  - `nextChar = cin.get();`

- To output a character, use *put*.
  - `cout.put(nextChar);`

- Special characters in C++: `\n` `\t` `\0` etc.
For formatted input or output, use manipulators.

#include <iomanip.h>
cout.setw(10); // Use 10 columns.
cout << n;
cout << setw(10) << n;
setfill(f) -- sets fill char to f.
setprecision(n) - for float.
setf(ios::flag) where flag can be fixed, left, right, showpoint etc
Functions

type fnname(formal parameter list)
{
    body
}

*type is void* for fncts not returning a value.

Foo(a,b); // Call Foo. a & b actual args.

return <value>; // return value for a fnct.

return; // for fncts with return type void.
Formal Parameters

- **Pass by Value.**
  - Function needs the value as input.
  - Function will not change the actual.

- **Pass by Reference**
  - Fn may or may not need the value as input.
  - The function will change the actual.

- **Const Pass by Reference**
  - Function needs the value as input.
  - Function will not change the actual.
  - Copying of argument is expensive.
Formal Parameters (contd)

Example: Roots(m,n,p,r1,r2);

boolean Roots(double a, double b, double c, double &root1, double &root)
{
    double disc; // b^2 - 4ac
    :
}

Formal Parameters (contd)

- `int`, `double` etc can be passed by value.
- Always pass class variables by reference.
- Use `const` if the actual is not to be changed.
- If there is a strong reason to pass a class by value, you can still do that.
- Arrays are always passed by reference.
- If the formal is an array, use `[ ]` notation.

```c
void Sum(const int a[], const int n);
```
Function Design

- If a function is single valued, use return value of the function.
- If a function returns more than one value, make them all as output parameters.
- The return value for multi-valued function is usually void or indicates status of the operation (Success or Failure etc).
Function Design

Always use a decorated header before each function to explain the purpose and usage.

Never return reference to local variables as return values.

```c
int &Foo(int n)
{
  int m;
  :
  return m;
}
```
Function Prototype

✧ It is just the first line of function definition.
   ● Cut and Paste to keep it identical.

✧ Prototype *always* ends with a semicolon.

✧ For member functions of a class, the class definition has the prototype.

✧ For non-member functions, place the prototype either in the header file or at the start of `.cc (.cpp)` file.
Function Header

//---------------------------------------
// Function: ComputeMax
// Parameters:
//   items[] input array
//   count   # of items
// Returns:  Max item of array items.
// Precond:  count > 0.
// Postcond: return value is the maximum
//           of the items in the array.
// Comment:  items[] and count are const.
//---------------------------------------
int ComputeMax(const int items[],
               const int count)
If Statement

if (expression)
{
   /* 3 columns of indentation */
}
else if (expression)
{
  n = 0; /* example of indentation */
}

else
{
   /*
   */
}
If Statement (contd)

- Always use { } even if the body has only one statement.
- } should appear in the same column as if.
- body should be indented 3 columns inside.
- Use spaces for clarity.
  - if(n==0) vs if (n == 0)
- else if parts and else part are optional.
- You can nest if statement.
Switch Statement

switch (expression)
{
    case <value1>:
        // note indentation style.
        :
        break;
    case <value2>:
        :
        break;
    :
    default:
        // catch all case.
}
Switch Statement (contd)

- `<value1> <value2> etc must be constant expressions.
- Each case should normally have a break.
- A case missing a break will continue with the body of next case (no matching needed).
- If a break is missing, include a comment.
  - // Fall through
- Always use default case. If nothing to do, just use a ; and put a comment.
While Statement

```c
while (expression)
{
    // body starts indented inside.
    :
}
```

- Always use `{ }` even if just one line body.
- `break` can be used to get out of loop.
- `return` can be used to get of a function.
- `continue` can be used to start next iteration.
- Body executed 0 or more times.
For Statement

for ( initialize; test; update )
{
    // body is 3 columns inside.
    :
}

⚠️ Always use {} even if one line body.
⚠️ Use spaces liberally for clarity.
⚠️ If you use local variables, do not use it outside loop, particularly loop variable.
For Statement (contd)

- **break** can be used to get out of loop.
- **return** can be used to get out of a function.
- **continue** can be used to start next iteration.

If loop variable is needed outside the loop, declare it in the function, not in the loop.

```c
for (int i = 0; i < n; i++)
{
    :
}
if (i == n) // can't use i here.
```
Do Statement

do
{
    // body starts 3 columns inside.
    :
} while (expression);

break can be used to get out of loop.
return can be used to get of a function.
continue can be used to start next iteration.
body executed 1 or more times.
One-Dimensional Array

❖ Always use named constant for dimension.
  • int temperature[DAYS_PER_WEEK];

❖ Do not use typedef to create array type.
  • typedef int ArrType[MAX];
  • ArrType arrVar;
  • Do you know why?

❖ The indices for n element array: 0..n-1.

❖ Arrays are always passed by reference.
Strings

- C++ supports strings. #include <string>
- Prefer C++ strings to C style strings.
- Do not use cstring header file.
- C++ strings support lots of functions.
- C++ string is implemented using a class.
- String class has many member functions.
- For g++, include string header before vector
Strings Examples

```cpp
string name1("woof");   // Constructor.
string name2 = "acdef"; // Constructor.
name1 = "Debby";        // operator= call.
string name3 = name1 + name2; // Concatenation
string name4 = name1 + "Dallas";
name3 = name1.substr(2,3); // ebb
if (name1 < name2)       // Lexigraphic comparison
    name1[1] = 'c';    // Accessing individual chars
    cin >> name3;     // skips whitespace.
    cout << name1;
getline(cin, name2);
    n = name2.length();
if (name2.find("ab") == string::npos) //if not found
    //...look for "cd" in the string, starting at the 2nd character
    string::size_type pos = name2.find("cd",1)
```
File Input

#include <fstream.h>

ifstream inFile("ages.dat"); // open.

ifstream inFile; // create variable.
inFile.open("ages.dat"); // open now.

if (!inFile)
{
    // File was not opened. Error.
    cout << "Unable to open file.\n";
}

File Input

char ch;
inFile >> ch;    // skip white space
inFile.get(ch);  // read next char.
ch = inFile.get(); // same
ch = inFile.peek(); // Get, don’t consume.
inFile.ignore(2); // skip next 2 char.
while (inFile.peek() != EOF)
{
    cout.put(inFile.get());
}

File Input

#include <fstream.h>
// Sum integers in a file.
ifstream inFile("data");
int num, sum = 0;
while (inFile >> num)
{
    sum += num;
}
cout << "Sum is " << sum << endl;
File Output

#include <fstream.h>
ofstream outFile(“ages.out”);
outFile << ‘a’;
outFile.put(ch); // ch is char variable.
string name(“abc”);
outFile << name;
outFile.close(); // close file.
Header Files

//----------------------------------------
// File: queue.h
// Author: K. Ganesan
// Purpose: Interface file for queue.c.
//          Supports an ordinary queue.
//----------------------------------------

#ifndef _QUEUE_H
#define _QUEUE_H

#endif
Header Files

What goes into a header file?

- include other header files.
- names constants
- typedef and class definitions.
- non-member function prototypes.
- extern declarations for globals in corresponding c++ source file. extern int dbase;

Do not place code in header files unless it is a template file. (templates later in class).
C++ Source Files

What goes into a c++ source file?

- include header files.
- local typedef and class definitions.
- Any global variables (avoid if possible).
- code for all functions.
- Local functions should be declared static.

Never include a c++ source file.
Library Files

- Library file = collection of object files.
- You can include them in a project.

Say you have a directory lib with libutil.a and want to use this library and math library.

```g++ -L../lib f1.cc f2.cc -lm -lutil```

- In Unix, you can use `ar` command to create a library file from object files.
Conditional Macros

✈ Useful for controlling debug statements.
✈ Useful to selectively use different logic based on machine or operating system or project or whatever.

```c
#if defined(DEBUG)
cout << "sum is " << sum << endl;
#endif
#if defined(WINDOWS95)
#include <win.h>
#endif
```
g++

Never ever compile a header file.
Assume that you are using xterm window.
To create queue.o from queue.cc:
g++ -I../include -c queue.cc
-I option tells compiler where to look for our header files.
To create queue.exe:
g++ -o queue.exe queue.o queuetest.o
To run the program simply type the name: queue.exe