CSE 143

Modules and Specifications

[Chapter 1]

Modularization

- Basic idea: break apart large system into smaller units (modules)
- Group related functionality in one module
- Often continued in a hierarchy of modules
- Design modules to be general and reusable
  - Multiple times in same program
  - Different programs/programmers

Specification vs. Implementation

Two parts of each module

- Specification (*what*)
  - Also known as "interface"
  - Describes the services that the module provides to clients (users)
  - **Publicly** visible
- Implementation (*how*)
  - Parts of the module that actually do work
  - **Private**, hidden behind module interface

Specification as Contract

Module specification acts as a contract between client and implementor

- Client depends on specification not changing
- Doesn't need to know any details of how module works, just what it does
- Implementor can change anything not in the specification, (eg. to improve performance)
- Implementation is a "black box" (***encapsulation***), providing ***information hiding***

Locality

- Locality of design decisions from encapsulation
- Benefits of private data and algorithm locality:
  - Division of labor
  - Easier to understand
  - Implementation independence

Specification

- Supplies constants, data types, function prototypes
- Comments describing what each function does
  - **Preconditions**: What should be true before function call (e.g., relationships among arguments)
  - **Postconditions**: What is true after function return (e.g., relationship of return value to arguments)
  - **Invariants**: What must be true while function executes (e.g., relationships among local variables)

Sample Specification File

```
// Specification file for computational geometry
// functions
// POST: returns the area of a circle with given
//       radius
double circleArea (double radius);
// PRE:  area must be non-negative
// POST: returns the radius of a circle of given
//       area
double circleRadius(double area);
```

prototype

geometry.h

Sample Implementation File

```
// Implementation of geometry functions
#include "geometry.h"
#include <math.h>

const double PI = 3.1415;

double circleArea (double radius) {
  return PI * radius * radius;
}

double circleRadius (double area) {
 return sqrt(area/PI);
}
```

geometry.cpp

Sample Client File

```
#include <iostream.h>
#include "geometry.h"
int main(void)
{
  double value;
  cout << "Enter radius: ";
  cin >> value;
  cout << "Area of circle is " <<
    circleArea(value) << endl;
  return 0;
}
```

main.cpp

Separate Compilation

- One module usable by many clients
- Individual modules may be changed and recompiled without changing entire program
- Client's code can be changed and recompiled without recompiling modules
- Note: Interface (specification) changes mandate recompiling both implementation and client

Designing Modules

- Must think about implementor's and client's roles
- Implementor's goals:
  - Provide a reusable, robust abstractions
  - Make interface independent of client
  - Keep freedom to modify implementation
  - Protect module from abuse and misuse
- Client's goals:
  - Assemble a program from usable modules
  - Rely solely on specification

Standard Libraries

- C/C++ comes with some (C++ many) predefined modules (libraries)
  - `iostream.h`, `fstream.h` for stream I/O
  - `math.h` for `sin`, `cos`, `sqrt`, etc.
  - `string.h` for `strcmp`, `strlen`, etc.
- Compilers also include nonstandard libraries
  - Graphics, windowing, etc.
  - Please don't use them for CSE 143

Summary

- Specification (`.h`) and implementation (`.cpp`) files
- Specification as contract
- Information hiding, black box analogy, locality
- Imports and exports, public and private code
- Separate compilation

1/14/99 80