CSE 143 Summer 1998

Abstraction and Modules
[Chapters 1 & 2]

What is Abstraction?

- It's what you get when you…
  - Ignore the messy details
  - Focus on the essential qualities
- The canonical Black Box
- The Platonic ideal
- The foundation of Computer Science
  - Every item in your bag of tricks is an abstraction

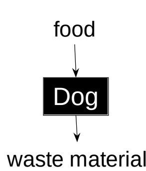
Abstraction in Everyday Life

- We intuitively make and use abstractions
- They provide us with mental models for the world around us
- They make us smarter
  - Better organization of information
  - Better ability to cope with complexity
  - Better capacity for problem-solving

Abstraction in Computer Science

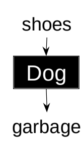
- Programming languages contain **abstraction mechanisms**
  - A tool for building a new abstraction
  - Examples: functions, classes, modules
- Two components: specification and implementation
  - Specification: what an abstraction promises to do
  - Implementation: how it keeps that promise
- Once implementation works, forget about it!

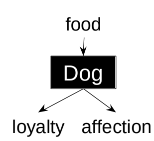
Example



Choosing the Right Abstraction

- Often, many abstractions are possible
- Trick is to choose the right one
- Correct abstraction depends on what's "essential"



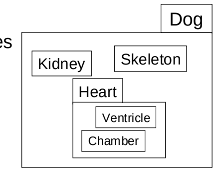


Levels of Abstraction

- A higher level of abstraction disregards more details
- A dog is…
  - A physical system (a collection of atoms)
  - A bag of organs
  - A furry thing that slobbers
- Must choose the correct level of abstraction
  - Too low: too many messy details still remain
  - Too high: difficult to understand/control behaviour

Layers of Abstraction

- An abstraction can be composed out of other abstractions
  - Black boxes within black boxes
- Layers don't know internals of lower layers
  - No crossing of the "Abstraction Barrier"



Example

- OSI networking model: seven layers

application
presentation
session
transport
network
link
physical

Program Decomposition

- When designing a program, the first step is to break it down into manageable chunks
  - And repeat the process for each chunk!
  - Some chunks are given to you, *e.g.* a library
- Implement the chunks, "glue together" for final program
- Important to find correct layers and levels
- Not all program design is top-down in this way

The Chunk Hierarchy

- There's no fixed set of names for layers of decomposition
- But many, many buzzwords
  - Many ideas about how this should be done
- function, class, module, component, library, toolkit, framework, subsystem, system, …
- In this course, we'll focus on the first three

Modules

- C and C++ do not require multiple files
- Guh.

What is a Module?

- A unit of decomposition
- A unit of reusability
- A collection of related items packaged together
- Example: a stereo system

Modularization

- Basic idea: break apart large system into smaller units (modules)
- Group related functionality in one module
- Design modules to be general and reusable
  - Multiple times in same program
  - Different programs/programmers
- Package modules into black boxes, communicate via interfaces

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)

Locality

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

Modules in C++

- Modules represented by a pair of files
  - specification (.h) file
  - implementation (.cpp, .cc, .c++, .C, etc) file
- Client's only interaction with module is through the interface defined in the .h file

Imports and Exports

- Specification (.h) file declares which items are exported
  - constants, function prototypes, and data types
- Client program must import features of a module to use them
  - Use the `#include` directive
  - Implementation (.cpp) file also uses `#include` to ensure it obeys the contract

Specification

- Supplies constants, data types, function prototypes
- Comments describing what each function does
  - Including preconditions, postconditions and invariants, as appropriate
- Client should be able to refer to specification as module's documentation

Sample Specification File

```
// geometry.h -- Specification file for
// computational geometry functions

#ifndef __GEOMETRY_H__
#define __GEOMETRY_H__

// circleArea: Returns the area of a circle with given radius
double circleArea( double radius );

// circleRadius: Returns the radius of a circle of given area
// PRE:  area must be non-negative
double circleRadius( double area );

#endif // __GEOMETRY_H__
```

prevent multiple inclusion!

geometry.h

Sample Implementation File

```
// geometry.cpp
// Implementation of geometry functions

#include <math.h>
#include "geometry.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

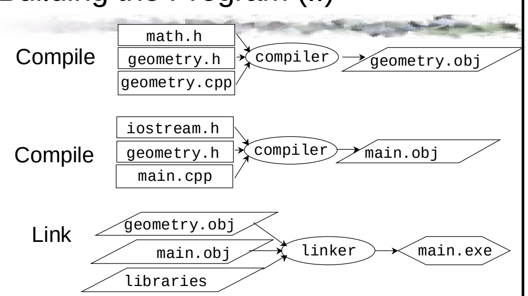
```
#include <iostream.h>
#include "geometry.h"

int main( void )
{
    double value;

    cout << "Enter radius: ";
    cin >> value;
    cout << "Area of circle is " << circleArea( value )
         << endl;

    return 0;
}
```

main.cpp

Building the Program (I)

- Three stages to go from source code to executable:
  - Preprocess
    - reads `#include`d files, expands `#define`s
  - Compile
    - Converts C++ code to object code the computer can understand
  - Link
    - Connects your object code with system libraries to make an executable program

Building the Program (II)



Separate Compilation

- Each module's .cpp source code is converted into object code separately
- Linker collects object code together to build executable
- Many environments hide this process from you
  - On MSVC, just press the "build all" button (or even just "run" …)
  - Must be done "manually" under UNIX

7/8/98 CSE 143 Summer 1998 88

Advantages of 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
- Can distribute object code to protect secrets
- **But**: Interface (specification) changes mandate recompiling both implementation and client

7/8/98 CSE 143 Summer 1998 89

Designing Modules

- Must think about implementor's and client's roles
- Implementor's goals:
  - Find right level of abstraction, build clean interface
  - Protect the implementation
- Client's goals:
  - Assemble a program from usable modules
  - Rely solely on specification

7/8/98 CSE 143 Summer 1998 90

Standard Libraries

- C/C++ comes with some 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

7/8/98 CSE 143 Summer 1998 91

Summary (I)

- Abstraction is ingrained in our minds
- Programming languages are tools for constructing abstractions
- Choosing the correct levels and layering of abstractions is crucial in programming
- Finding the right breakdown is difficult

7/8/98 CSE 143 Summer 1998 92

Summary (II)

- A C++ module has specification (.h) and implementation (.cpp) files
- Specification as contract
- Modules support reusability and decomposition through encapsulation
- Separate compilation is a big win when writing software

7/8/98 CSE 143 Summer 1998 93