CSCE 221

Data Structures & Algorithms

Dr. Teresa Leyk

Office: 317B Teague Bldg.

e-mail: teresa@cse.tamu.edu

Course web page:

http://courses.cs.tamu.edu/teresa/csce221/csce221-index.html

What Is a Data Structure?

- A data structure is a systematic way of organizing input data and specifying operations (algorithms) which can be performed on this data (e.g., add, delete, search).
- A data structure is a theoretical concept of representing data and operations on them which can be implemented in any programming language (C, C++, Java, etc.)
- Purpose:
  - Make a particular operation (algorithm) more efficient in terms of computational time and memory use.
  - Reuse a given data structure for many different applications.

Abstract Data Type (ADT)

- To understand the design of a data structure, we use an abstract model called *Abstract Data Type* (ADT) that specifies
  - the type of the data stored
  - the operations that support the data
- This approach is independent of any particular implementation. The main feature of ADT is
  - a clear description of the input to each operation
  - the action of each operation
  - its return type

  It allows a programmer to focus on an idealized model of the data and its operations.

Abstract Data Type (ADT)

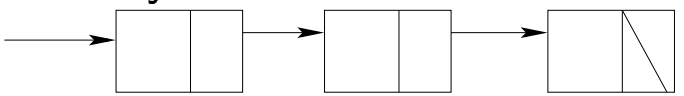
- Any ADT can have a few different implementations. We look for an implementation which is most efficient (in terms of running time and/or used memory) for a given problem.
- We will use ADT to describe any data structures in this course.

Examples

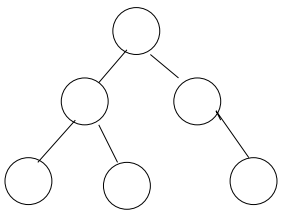
- linked lists
- queues
- stacks
- trees
- heaps
- graphs

Data Organization

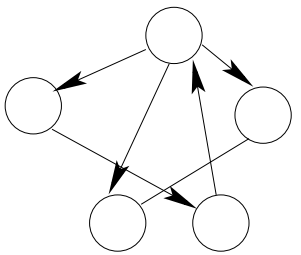
- The relation between particular elements of a data structure can be:
  - linear: list, array, vector, linked list, sequence, queue, stack, deque



  - hierarchical: tree, heap



  - arbitrary: graph, network



Data Structures in C++

- An ADT is realized in C++ as a class. The public member functions of the class correspond to the operations of the ADT. The private part of a class depends on a given implementation.
  - Typical operations on data structures: insert, delete, update, search, sort.
- C++ provides implementations of different data structures in the STL (Standard Template Library); they are called containers.

C++ Containers

Containers					
Simple	Sequences	Adapters	Associative		Others
			ordered	unordered (C++11)	
pair	vector	queue	set	unordered_set	bitset
	list	stack	multiset	unordered_multiset	array
	deque	priority queue	map	unordered_map	string
			multimap	unordered_multimap	

A Simple Vector ADT

- A fixed size vector ADT represented as a C++ class.

```
class My_vec {
  private:
    int size, capacity;
    char *ptr;
  public:
    My_vec(): size(0), capacity(10),
              ptr(new char[capacity]) {}
    ~My_vec() { delete ptr; }
    int get_size() const { return size; }
    bool is_empty() const { return size==0; }
    char& elem_at_rank(int r) const;
    void insert_at_rank(int r, const char& elem);
    void replace_at_rank(int r, const char& elem);
    void remove_at_rank(int r);
};
```

Application Programming Interface (API)

- API description allows other programmers to use the public interface for the class without having to view the implementation details.
- C++ does not provide an explicit mechanism for specifying APIs (= interfaces) as in Java where an interface is a collection of function declarations with no data and no bodies.
  - C++ abstract class (with pure virtual functions) is one way to define interfaces.
  - Another way is to use "informal interfaces" as is suggested in the textbook.

C++ Programs

- Any program usually performs three basic operations: it gets data, manipulates the data and outputs the results.
- A C++ program is a collection of functions and classes which obey their predefined syntax.
- Writing and editing a source code.
  - Divide a problem into classes, and decide how they should interact.
  - Write and edit a C++ source code using a text editor (do not use a word processor). For instance, Windows XP/7/8 editors are `notepad` or `edit`, and UNIX/Linux editors are: `emacs`, `vi`, or `pico`.
  - You can use Visual Studio on Windows.
  - Your code should be readable – add meaningful comments.

C++ Files Organization

A good idea to place each class in a separate file. There are two common type of files: source and header files.

- The source file contains executable statements, data and function definitions. Often the suffix "cpp" or "C" is added to its name.
- The header file contains declarations of classes, structures, enumeration, typedefs, functions, and definitions of inline functions. The suffix "h" or "H" is added to the name of this file.

The source files may be compiled separately by a compiler and later combined by a linker into one program.

Compiling a Source Code (Unix/Linux)

- An example of compiling the source code under Linux using g++ compilers for the 2003 and 2011 versions, respectively:

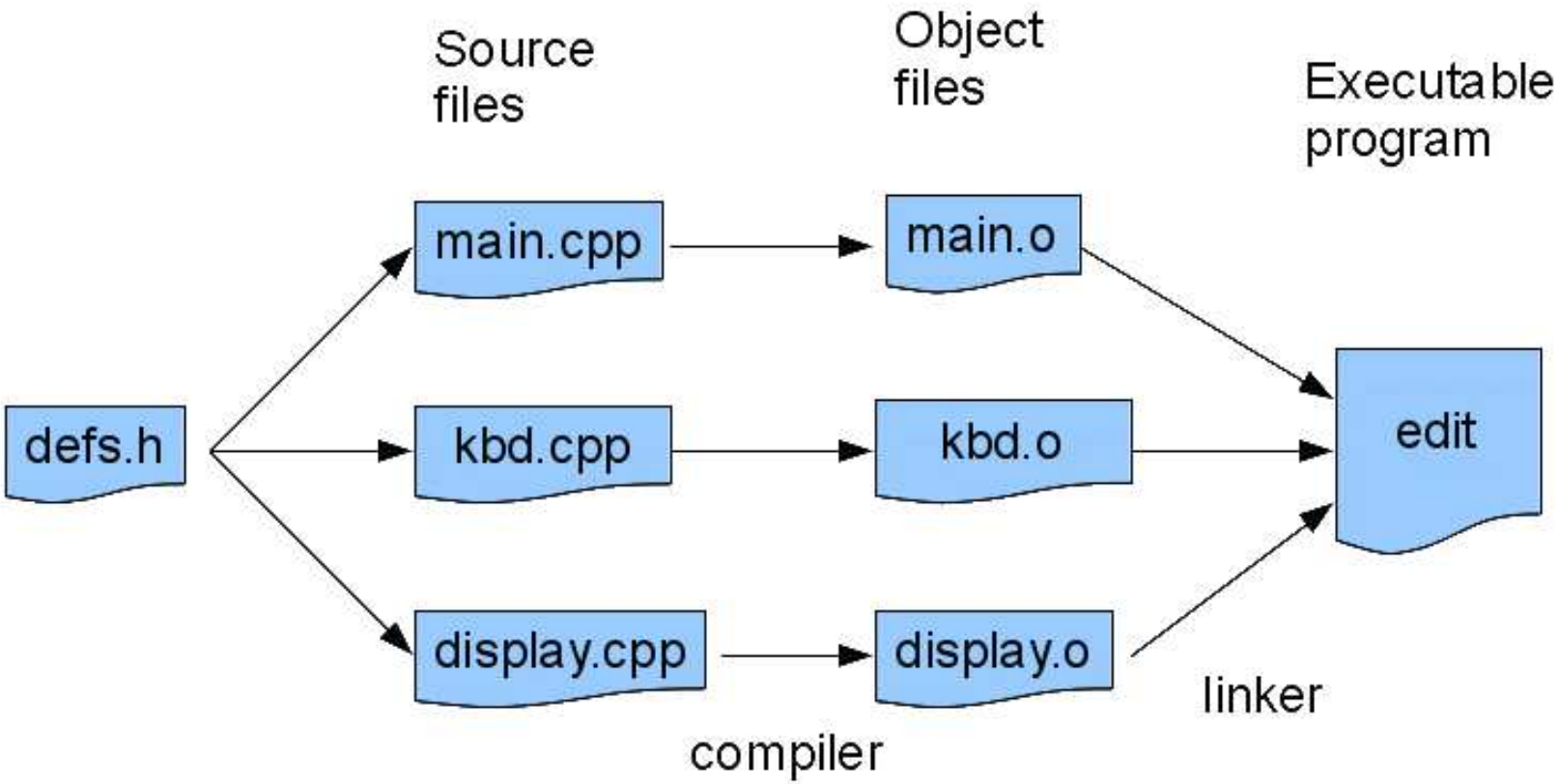
```
g++ -c myProgram.cpp
g++ -c -std=c++11 myProgram.cpp
```

- After compilation you get an object code – source code translated to the machine language. You need to correct all errors detected during the compilation.

Compiling with Make (Unix/Linux)

- The `make` program/command is useful for the automatic generation of files of any type including binary executable or object files.
- Before you can use the utility "`make`", you need to prepare a file called "`makefile`" or "`Makefile`" that describes the relationships among files in your program.
- Typically the executable file is created from object files (ending with `.o`), which, in turn, are created by compiling C++ source files (ending with `.cpp`). Invoking the command `make` is sufficient to perform all necessary compilations.

Example


Source files
Object files
Executable program
main.cpp
main.o
defs.h
kbd.cpp
kbd.o
edit
display.cpp
display.o
linker
compiler

Linking an Object Code

- An object file (containing an object code) must be combined with libraries and with other previously generated object files to create an executable file. The linking phase is often done by the compiler so you do not need to call a linker directly.

```
g++ -o myProgram myProgram.cpp
g++ -std=c++11 -o myProgram myProgram.cpp
```

- A successful compilation and linking will create the executable file called `myProgram`.

Running a Program

- Program execution begins in a function with the special name `main()`.
- To run a program you need to call the executable file, often by writing its name (such as `myProgram`).
- Some errors may appear during this stage (they are called *run-time errors*) which sometimes are not easy to find and fix. For example, you can get:
  - bus error (incorrect function arguments)
  - segmentation fault (uninitialized pointers or arrays, working beyond array limits).

Debugging a Program

- Using a debugger (for instance, `gdb`) may be helpful.
- Inserting output statements (using `cout`) is also useful for small programs.
- You always need to test your program – do not assume that it will work correctly after a successful compilation.
- Consider special cases of program input.

How to Approach Programming Assignments

The software development process consists of the following phases (sometimes called *software life cycle steps*):

1. Problem Analysis
2. Structured Design
3. Implementation
4. Testing and Debugging
5. Production and Maintenance

Problem Analysis

- Thoroughly understanding a problem, user's input and output requirements and their representation.
- Thinking about data organization and algorithms for their manipulations in a program
- Identifying library software useful for a problem solution
- Deciding about design and a paradigm
  - generic programming supported by templates – focusing on efficiency of algorithms
  - object oriented programming (OOP) with its principles: encapsulation, inheritance and polymorphism.

Structured Design

- Designing an algorithm to solve a problem
- Partitioning a problem into subproblems using the top-down design, stepwise refinement and modular programming
- Using interfaces as a communication mechanism between modules (classes)
- Specifying precondition and postcondition for operations

Implementation

- Translating an algorithm into a programming language
- Removing syntax errors

Testing and Debugging

- Removing all logical errors
- Black-box testing, using different input data and checking the correctness of output
- White-box testing, using internal structure and implementation of each function.

Production and Maintenance

- Installing the software on user machines (Unix/Windows)
- Modifying the software according to user requirements.

Software Design Features

- **Robustness** – producing correct solutions and being able to handle inputs that are not expected for this application.
- **Adaptability** – designing a software in a smart way that can be adopted easily to other applications across different platforms with reasonable maintain time or complemented by additional systems.
- **Reusabilty** – using part of a program as a separate component of different systems.