# EEE101: C Programming & Software Engineering I

## Lecture 10: Advanced Data Representations & Other Software Development Issues

Dr. Rui Lin/Dr. Mark Leach
Office: EE512/EE510
Email: rui.lin/mark.leach@xjtlu.edu.cn
Dept. of EEE XJTLU

# Outline of Today’s Lecture (10)

- Data structures:
  - Arrays and Linked Lists
  - Binary Trees
  - Hash Tables
- Abstract Data Types
- Additional Programming Issues:
  - Debugging your code
  - “Make” file
  - Version Control
  - Software Testing

## Data Structures

## Data Structures

- We have so far considered two basic data structures.
  - Arrays – sequentially ordered data elements of predefined size.
  - Linked List – dynamically created and distributed data elements linked by the data elements.
- We have also considered the advantages and disadvantages of the two, but let’s recap a little…

## Data Structures – Arrays

- The array is a basic data structure, consisting of a group of elements all with the same data type, stored in a sequential block of memory
- Advantages
  - Easy to access by index or pointer (incrementing or decrementing either moves through the elements)
  - Easy to perform a search
  - Fixed memory size (known at compile time)
- Disadvantages
  - Fixed memory size (too little or too much)
  - Sorting is time consuming (whole elements must be copied)
  - Inserting and deleting is also awkward

## Data Structures – Linked List

- The linked list is another basic data structure, constructed of structure variables, each dynamically created and linked by including a pointer to the next element in the list within the structure.
- Advantages
  - Length of the list is only limited by computer memory
  - Sorting is accomplished by re-ordering the pointers
  - Deletion/insertion accomplished the same way
- Disadvantages
  - Searching is not simple like the array
  - Searching is in one direction (unless doubly linked)

## Bubble Sort

- Let’s look at a popular sorting algorithm.

```
#include<stdio.h>
#define SIZE 6
main(){
int i, n=0, temp, array[SIZE]={6,3,4,8,1,3};
        while(n<SIZE-1){
                for(i=0; i<SIZE-n-1; i++){
                        if(array[i+1]<array[i]){
                                temp=array[i];
                                array[i]=array[i+1];
                                array[i+1]=temp;}
                        }
                n++;}
}
```

## Sorting

- Imagine that the array was hundreds, or even thousands of elements long and that each element was a **struct** containing thousands more elements.
- The sorting would take a long time. Each swap requires 3 element contents to be moved.
- The linked list requires only that pointer values are swapped.
- Similar issues can be highlighted for deletion and insertion of new elements.

## Searching

- I have a data set consisting of numbers, that is ordered in increasing value. (think about your guessing game in assignment 3)

  What is the best place to start the search?

- Most sensible is the array middle and then moving up and down by halving the remaining section.
- **Example: an array contains 1,3,4,5,7,8,9 and I know the values can be between 1-10**
- If I want to find where the 7 is, what is the fastest route?

## Binary Search

- **array contains 1,3,4,5,7,8,9 and I know the values can be between 1-10**
- **If I start at the beginning of the array I have to look in 5 elements to find the 7.**
- In general it is always best to choose the mid point
- **First test would be a element 4, which equals 5.**
- **7 is bigger than 5 so move to the mid point between array centre and end of array.**
- **Now looking at element 6, which equals 8.**
- **7 is smaller, so move down the array**

  …Hey presto, element 5 contains the value 7.

## Binary Search

This method of searching is called a binary search and represents the fastest method on average for searching a data structure for a value.

It is possible with an array, but in a linked list there is no way to know where the middle of the list is and we can’t move up and down the list (unless it is doubly linked)

## Array or Linked List?

- If a situation calls for a list that is continuously resized with frequent insertions/deletions but that isn't searched often, linked lists are a better choice.
- If a situation calls for a stable list with only occasional insertions/deletions but that are searched often, an array is the better choice.
- Of course there’s more…two other important data structures, namely the binary tree and the hash table, reduce the time for searching an element while retaining the advantages of using a linked list.

## Binary Search Tree

- A binary tree is a data structure that like a linked list is built on the structure data type i.e. new elements are created dynamically and the order of the elements is determined by pointers embedded in the structures.
- But unlike a linked list they are stored in a predetermined order, based on some quantity within the structure

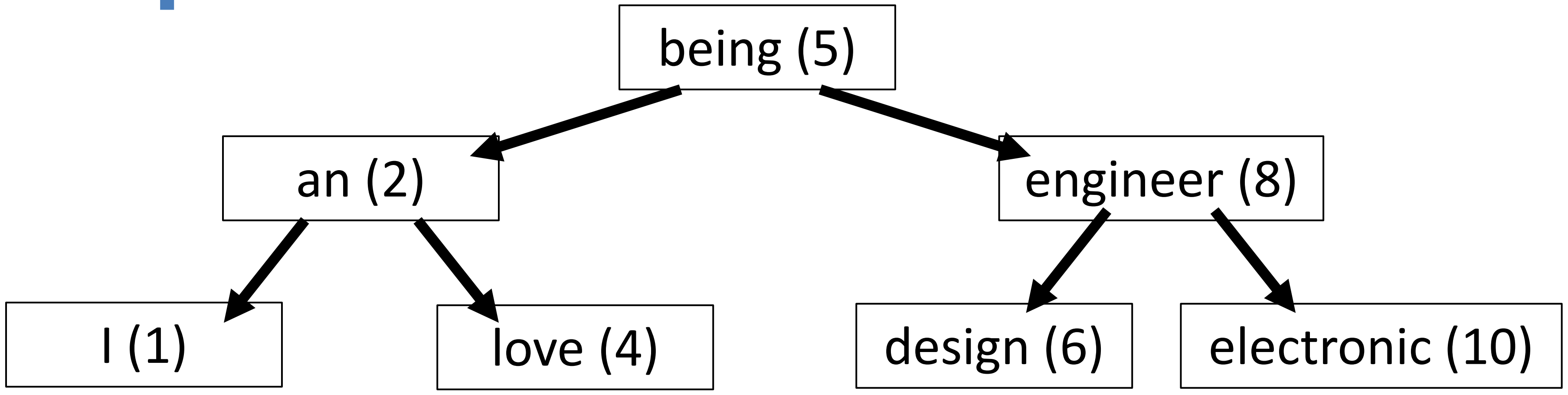
## Binary Search Tree - Example

Say we wanted to store all of the words in a book in a list ordered by the number of letters in each word. Let’s consider the passage “I love being an electronic design engineer”

| | | | |
|---|---|---|---|
| I | 1 | electronic | 10 |
| love | 4 | design | 6 |
| being | 5 | engineer | 8 |
| an | 2 | | |

Values are stored in a tree as nodes. Each node has a left and right branch (pointers), left <=, right >. The head node will be the centre value (5)

# Binary Search Tree - Example

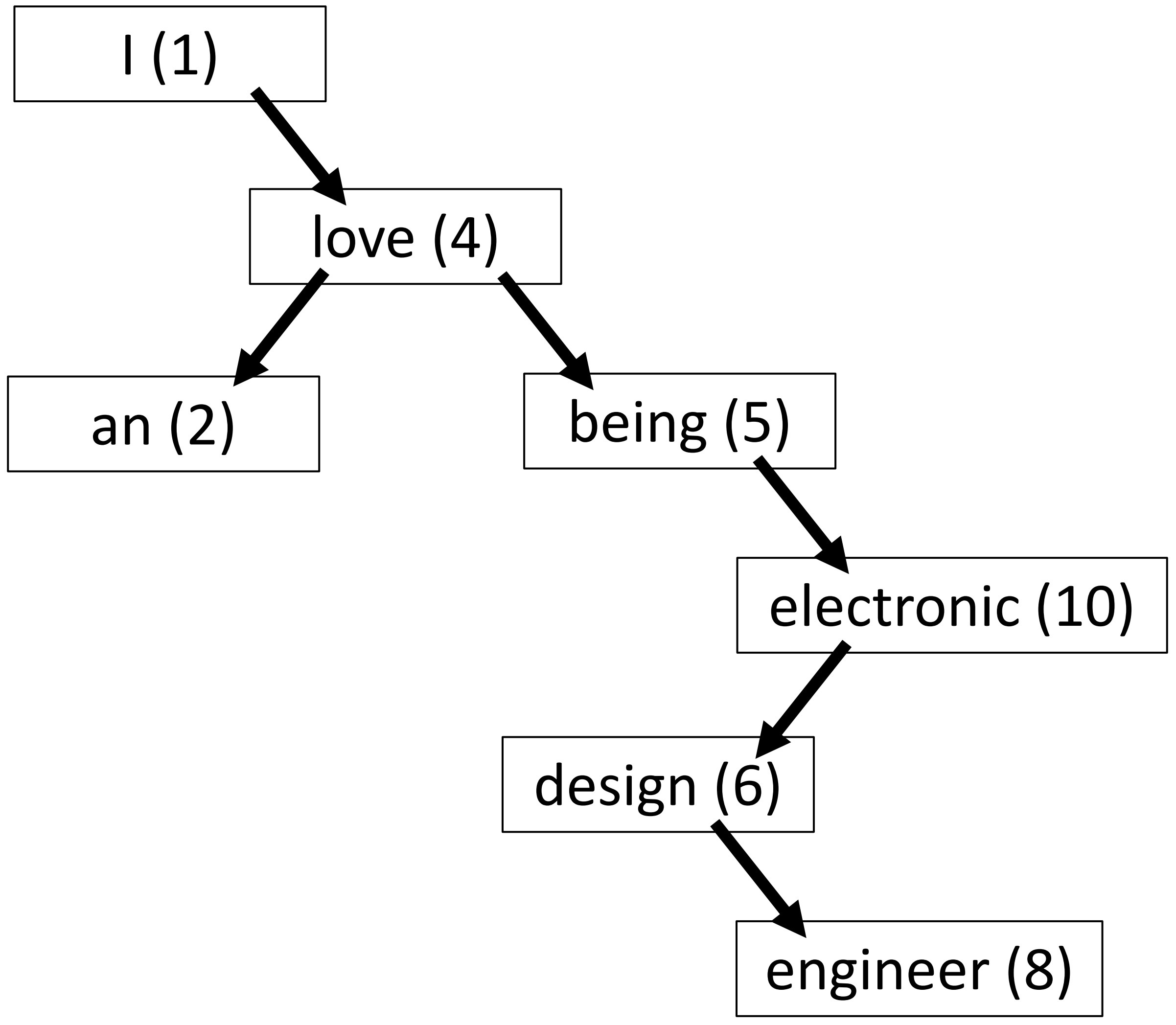


This is a balanced binary tree, designed for fast searching since it is sorted.

When forming a binary tree, usually the data is read into the program using the same rules, when input is finished the data is sorted.

```
struct node{    char *word;
                int count;
                struct node *left;
                struct node *right;};
```

# Unbalanced Binary Tree



# Binary Tree Example Program(1/2)

```
#include<stdio.h> #include<string.h>
#define MAXWORD 50                          /*maximum word length*/
struct node *addtree(struct node *, char *); /*Place node in tree*/
int getword(char *, int);                   /* function to get a word*/
struct node* talloc(int);        /*function to obtain memory for struct*/
                  /*All functions need to be written*/
int main(){
struct node *root;
char word[MAXWORD];
root=NULL;
while(getword(word, MAXWORD)!=EOF)
    if(isalpha(word[0]))                    /*Is word alphabetic*/
        root=addtree(root, word);
return 0;
}
```

## Binary Tree Example Program (2/2)

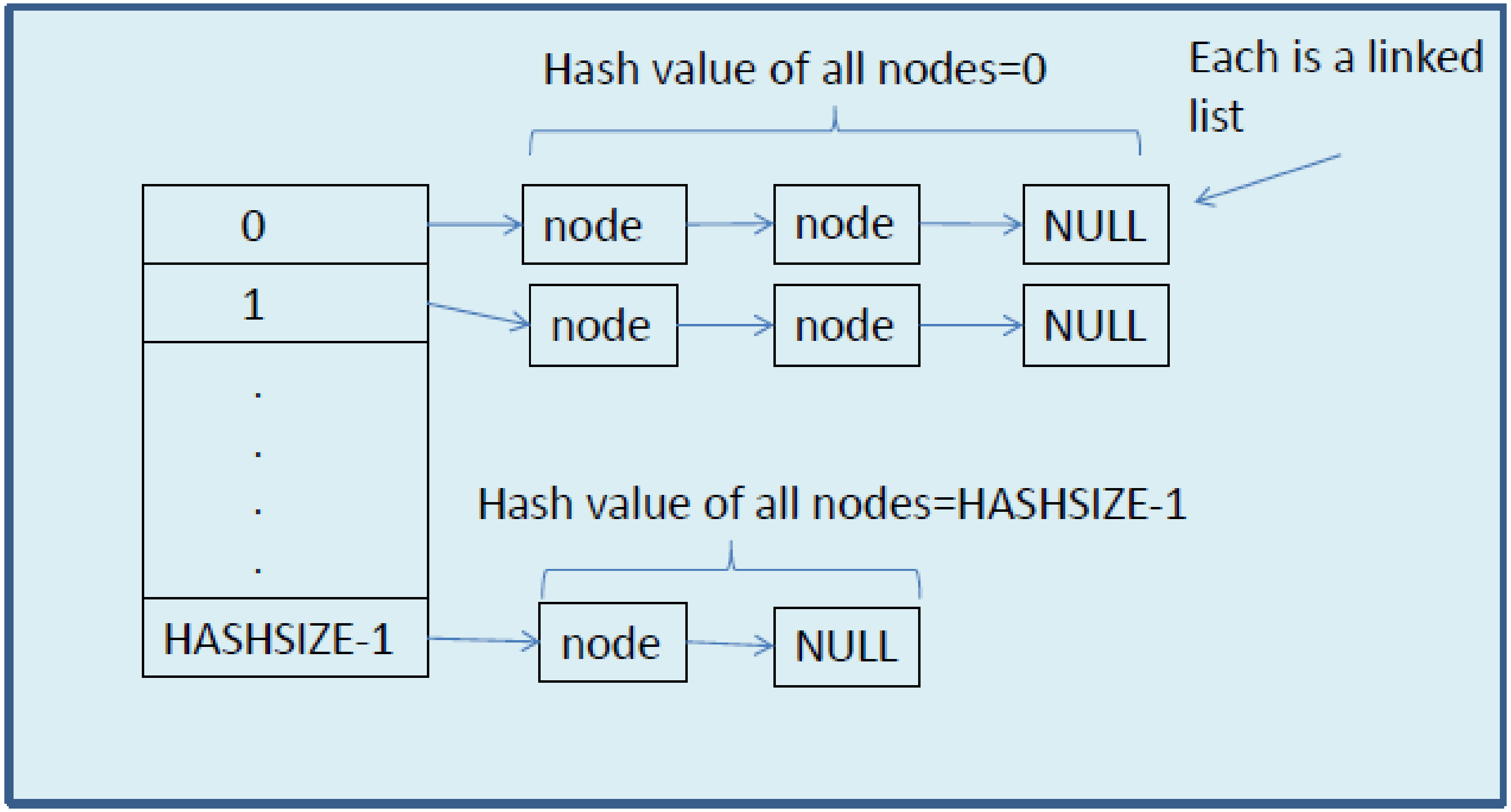
```
struct node * addtree(struct node *p, char *w){
        int cond;
        if(p==NULL){            /*a new word has been found*/
                p=talloc();     /*obtain memory for a new node*/
                strcpy(p->word,w);      /*copy word to new node*/
                p->count=1;             /*count word occurrence*/
                p->left=p->right=NULL;}         /*initialise pointers*/
        else if((cond=strcmp(w,p->word))==0)    /*does word exist*/
                p->count++;             /*count word occurrence*/
        else if(cond<0)         /*if w is less than p->word*/
                p->left=addree(p->left,w);      /*move left on tree*/
        else
                p->right=addtree(p->right,w); /*move right on tree*/
        return p;
        }               /*note recursion*/
```

## Lookup Table (Hash Table)

Another data structure is called a Hash Table, which does a similar job.

- A hash table is really an array of linked lists
- Appears to be a two dimensional array.
- The array is sorted in some pre-determined order defined as the hash value.
- Each element of the array is a pointer to the head of a linked list i.e. the array is an array of pointers.
- Choosing the hash value is important to try and ensure even distribution of the total number of entries.

# Lookup Table (Hash Table)



## The Hash Table - Example

```
#define HASHSIZE 200     /*This table has 200 rows*/
unsigned int hash(char *s){
        unsigned int hashval;
        for(hashval=0; *s != '\0'; s++)
                hashval = *s+31*hashval;
        return hashval % HASHSIZE;
        }                   /* so, 0<=return value< HASHSIZE*/
```

- The hashval is calculated to distribute words in the hash table by adding together the ASCII character values for each letter in a word multiplied by 31. The return value is scaled from 0-199 by the % operator

## Abstract Data Types

## Abstract Data Types (1/5)

In programming, you try to match data type to the needs of a problem, e.g. using an **int** or a **char**…but what does a data type define?

**Consider int** a; **- what is defined by this statement?**

**The memory requirements**

**The name of the variable…anything else?**

**All of the operations are also defined e.g. multiply, divide,** etc…These are all defined for you by the data type **int**

## Abstract Data Types (2/5)

So…to truly define a data type we need to:

- **provide a way to store the data e.g. a struct**
- **provide ways to manipulate the data**

A data type defined in this way with an abstract concept is called an abstract data type.

The abstract nature becomes concrete when you implement it using a programming language.

## Abstract Data Types (3/5)

How is such a thing defined and created?

It’s a 3 step process moving from abstract to concrete:

1. Provide an abstract description of the type's properties and of the operations you can perform on the type. Such a formal abstract description is called an abstract data type (ADT)
   Note: This description shouldn't be tied to any particular implementation. It shouldn't even be tied to a particular programming language.

# Abstract Data Types (4/5)

2. Develop a programming interface that implements the ADT. That is, indicate how to store the data and describe a set of functions that perform the desired operations.
   Note: In C, for example, you might supply a structure definition along with prototypes for functions to manipulate this structure. These functions play the same role for the user-defined type as C's built-in operators play for the fundamental C types. Someone who wants to use the new type will use this interface for her or his programming.

## Abstract Data Types (5/5)

3. Write code to implement the interface. This step is essential, of course, but the programmer using the new type need not be aware of the details of the implementation.

# ADT - An Example (1/4)

Consider the previous linked list of movies. How to make this an ADT?

1. Remember to stay abstract

- Define the data content

  (some useful operations)
- Initialise a list to empty
- Add an item to the end of the list
- Determine if the list is empty
- Determine the number of items in the list
- Display the content of an item in the list.
- Remove an item from the list
- ….

For the purposes of the example let's simplify…

## ADT - An Example (2/4)

Simplified ADT:

| | |
|---|---|
| • Type name: | Simple_list |
| • Type property: | Hold a sequence of items |
| • Type operations: | Initialise the list |
| | Add an item to the end of the list |
| | Display an item in the list |

2. Define the interface for the ADT

   Data definition

```
typedef struct movie {char title[SIZE]; int rating;}Item;
typedef struct node {Item item; struct node * next;}Node;
typedef struct list {Node *head; int size;}List;
```

## ADT - An Example (3/4)

Supply function prototypes such as:

```
void initialiselist(List *plist);
        /*operation: initialise a list*/
        /*preconditions: plist points to a list*/
        /*postconditions: the list is emptied*/
        function call might be:
        initialiselist(&movies);

void traverse(List l, void(*pfun)(Item item)),
        /*operation: apply a function to a list item*/
        /*preconditions: l is an initialised list*/
        /*postconditions: the function pointed to by
        pfun is exectued once for each item in the list*/
        function call might be:
        traverse(list_2014,(*display_item)(film_1)
```

## ADT - An Example (4/4)

Implement the code:

```
void initialiselist(List *plist){
        *plist = NULL;}

bool listisempty(List l){
        if(l==NULL)
                return true;
        else
                return false;}
```

## ADT – Notes (1/2)

When implementing an interface, you may wish to include additional functions that don’t relate directly to the ADT, but help to facilitate the interface.

The interface should be defined as an abstract list of operations. It should not be defined for a particular data representation or algorithm. This makes it easy to change the implementation without re-writing the final program.

## ADT – Notes (2/2)

The benefits this approach provides for the program development process are:

- If something is not working right, you probably can localize the problem to a single function.
- If you think of a better way to do one of the tasks, such as adding an item, you just have to rewrite that one function.
- If you need a new feature, you can think in terms of adding a new function to the package.
- If you think that an array or double-linked list would be better, you can rewrite the implementation without having to modify the programs that use the implementation.
- In short, as long as the interface is clearly defined and no changes have been made, you can do whatever you like for your part without having any impact on other people’s work. So, great for team work.

## Additional Programming Issues

# Debugging your code

**Firstly compilation errors:** these must be located and corrected before you can run your program.

**Incorrect results/outputs or exceptions:** these mean there is a problem with your algorithm design and so the debugging begins:

**The old way**

- Use a pencil and paper, execute (calculate the result of each instruction, line by line.
- Slow…but effective (a good test of your programming ability)

**The new way**

- Use a software tool (e.g. Visual Studio) to step through the code ling by line until the error is seen or the exception happens

**Keep a pencil and paper handy either way ☺**

## Debugging Techniques (1/2)

**Commenting out code**

**Sometimes you know there is a bug, but it’s hard to find.**

Suppose your program fails with an exception and aborts.

**Commenting-out** single lines of code (changing them to comments) can sometimes help locate bugs

```
int main(){
        drawBlock1();
/*      drawBlock2(); */
        drawBlock3();
        }
```

**Change each line into a comment one by one and run**

**If the error goes away, the fault probably lies in that function.**

## Debugging Techniques (2/2)

**Insert print statements controlled by a variable**

```
int main(){
        int debug = 0;
        if(debug) printf(“Now at step a”);
        /*lots of code*/
        if(debug) printf(“Now at step b”);
        /*lots of code*/
        if(debug) printf(“Now at step c”);
        /*lots of code*/
        }
```

**Changing the variable debug to 1 will print al of the statements.**

**Can determine where a programme crashes, or print values at specific points.**

## Software Debugging Tools

**Most compilers provide common debugging tools including:**

**Single stepping**

Allows your program to be evaluated line by line. Coupled with an output or watch window, you can choose to see various variable values as statements are executed

**Breakpoints**

Allow a program to be executed to a particular point and then single stepped from that point

## makefile, Version Control & Testing

**The rest of todays topics are introduced from an industrial perspective. The purpose of delivering this information is to make you aware that these are issues and so are of interest.**

## “Make” Your Files (1/2)

- **What are “make” and “makefile”?**

**“make”**, initially developed on Unix but available everywhere in some form, is a program which manages all individual project files following instructions in a text file called **makefile**.

- **Why do we need make?**

To compile a project consisting of multiple files, each time you modify your code you need to type all of the file names each time. **“make”** is a tool which handles this automatically.

- **How does make work?**

After editing one or many files in a project, execute **make**. The **make** program follows instructions in the **makefile**, comparing dates on the source code files to the dates on the corresponding target files. The compiler is invoked if a source file date is more recent than that of the target file. The **makefile** contains all the commands to put your project together. Learning to use **make** saves you a lot of time and frustration.

## "Make" Your Files (1/2)

- **A simple example of using make and makefile**

```
#a comment
hello.exe: hello.c
	mycompiler hello.c
```

- **What does this say?**
  This says that hello.exe (the target) depends on hello.c.
  When hello.c has a newer date than hello.exe, make executes the "rule" mycompiler hello.c.
  There may be multiple dependencies and multiple rules. Many make programs require that all the rules begin with a tab.

# Version Control (1/3)

Imagine you are working in a software development team of more than 100 engineers for a new Operating System. With each engineer working on specific modules that must work together. The concept of ADT's is being applied but all modules will impact on all other modules when integrated.

It is possible while you are working that a problem you need to solve may disappear or be different because of other peoples work on other modules.

Version control software helps to monitor changes. Common functionality of version control include:

- File locking
- Version merging
- Baselines, labels and tags

## Version Control (2/3)

- **How does a centralized version control software package work? All software is stored at a server called the software repository and all programmers are clients on the server. Each client computer copies the files from the repository (working copy) and after completing coding, their work is deposited back to repository. This process is called “commit”. Some of version control systems use lock-modify-unlock (obsoleted) mode and some systems, for example SVN, use copy-modify-merge mode to keep the entire project compilable at any moment.**

- **What is the relationship between version control and make? Normally, make is integrated into version control system.**

- **Distributed version control Distributed revision control systems (DRCS) take a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository. Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer**

# Version Control (3/3)

**The following are two examples about what you do and see when you are using SVN (centralized) as your version control software.**

- Have a local working copy using command **checkout**

```
$svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
A my-working-copy/README A my-working-copy/INSTALL
A my-working-copy/src/main.c
A my-working-copy/src/header.h
…
Checked out revision 8810. $
```

- Update your local working copy using command **update**

```
$svn update
Updating '.':
U foo.c
U bar.c
Updated to revision 2.
$
```

## Software Testing (1/4)

Testing is a very important part of any product development process to assure the quality of the product. Testing, however, cannot identify all defects within the product (software in our case). Instead, it compares the state and behaviour of the product against **oracles**—principles or mechanisms by which someone might recognize a problem. Note that not only can functional requirements but also non-functional requirements such as testability, scalability, maintainability, usability, performance, and security often be the source of defects.

The scope of software testing often includes examination of code and execution of that code in various environments and conditions: does it do what it's supposed to do and do what it needs to do.

It's a dynamic process in the sense that it requires a product can be executed with given inputs and produce observable outputs to reveal the existence of the faults in the product under testing. Once a fault is detected, debugging activities take place and the fault fixed, the testing process then resumes. Testing often continues after product release on the market.

## Software Testing (2/4)

Testing methods are roughly divided into

- Static testing: reviews, walkthroughs, or inspections.
- Dynamic testing: executing programmed code with a given set of test cases.

Testing traditionally takes a box approach:

White box testing: tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. An internal view of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determines appropriate outputs. White box testing checks whether or not the software does what it is supposed to do.

Black box testing: treats the software as a "black box", examining functionality without any knowledge of internal implementation. black box testing checks whether or not the software does what it needs to do.

Grey box testing: involves having knowledge of internal data structures and algorithms for purposes of designing tests, for execution of those tests is at the user, or black-box level. This as a mixture of white and black box testing.

# Software Testing (3/4)

**Some of the techniques involved are:**

- **White box**
  - API (application programming interface) testing of the application using public and private APIs
  - Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
  - Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
  - Mutation testing methods
- **Black box**
  - equivalence partitioning
  - boundary value analysis
  - all-pairs testing
  - state transition tables
  - decision table testing,
  - fuzz testing
  - model-based testing
  - use case testing
  - exploratory testing
  - specification-based testing.

## Software Testing (4/4)

- **Testing levels:**

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. These levels are

  - unit testing,
  - integration testing,
  - system testing,
  - acceptance testing.

- **Testing process**
  - Traditional waterfall development model
  - Agile or Extreme development model
  - Top-down and bottom-up
- **A typical testing cycle**
  - Requirements analysis-> Test planning-> Test development-> Test execution -> Test reporting-> Test result analysis-> Defect Retesting-> Regression testing -> Test Closure
- **Automated testing and Manual testing**

**Questions?**

**The End…**

**but the labs are still going ☺**