CS 327

version control software and git

Suppose you have complex system with lots of 7iles stuff

project1.v1.java

project2.v2.java

- poules you working directory
- error prone
- How to keep

Suppose you fix a problem in one snapshot - and you want to put the fit in other versions (snapshots)

— manually

use a real version control system.

what is a version control system.

- Some sort of database that effectively stores all past snapshots of your code, usually called "repository"
- Utilities to create and manage the snapshots in the repository

How they typically work

- Repository database
- You have a local, working copy of the repository

  This is usually a directory in your system.
- Every so often you can take snapshot
  - check in all (or some) files under version control.

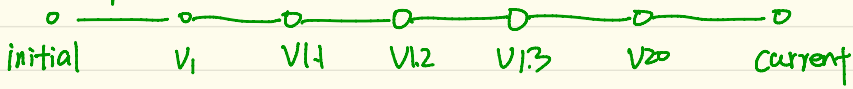
    (snapshot are cheap)
- You can recover previous snapshots whenever you want

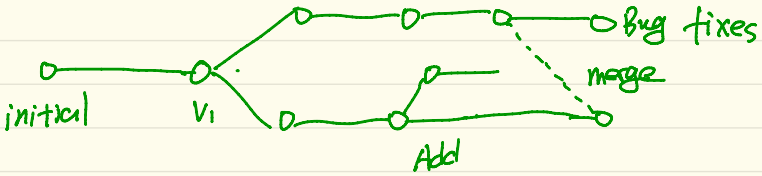
What should go into a repository

- All source files, Makefiles, Documentation
- What should definitely not go into the repository.
  Nothing that is automatically built from the files in the repository

Branching

Development need not be linear.

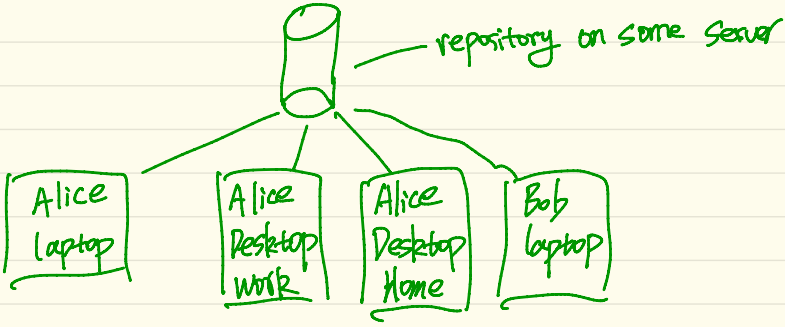




Remote Repository

You can have multiple working copies

- on different machines
- with different developers



These could be conflicts

Alice edits stack.c on laptop

Bob edits stack.c on laptop

If edits are far apart (not the same line) the system can usually merge changes

If edits are the same line then someone has to resolve conflict manually

popular (open source) version control systems

subversion replaced CVS replaced RCS

RCS snapshots per file

CVS snapshots per directory - but subdirectories

subversion snapshots of directory structure - (files and Directories)

Git · Develope by Linus Torvalds for use in c development

what we will be using in class

philosphy of git Disk is cheap

Network is expensive

So when you "check out" a repository from a server,

cheat sheet for git commands

git clone

git pull fetches change from the server and merge them with your local copy

git push

git add

git commit

git diff look differences between version

git status

ls

wol

coms327p1a jil hdpo zeg

emacs Maintainer

ssg @ cs.iastate.edu
coms-ssg @ iastate.edu

What is shell

usually written in C

usually runs interactively

(1) get prompt

(2) type command, press enter

(3) command executes

(4) get new prompt

Commands look like

Cmd arg1 arg2 arg3 ... argn

Cmd will either

built into the shell

Terminal Windows

- GUI front end for a shell
  - shell runs inside a window
- Windows has a shell "Command prompt" — Thin
  "power shell"
- Mac OS "Terminal"
- Linux lots of graphical terminal

Most important Unix Command

man -k keyword

$\boxed{\text{man}}$ — display manual pages

Arguments — which command to show man pages for

man man - gives documentation on the man program/cmd

Sections

The manual has sections and are numbered

Important for this class

- section 1 general commands (like man)
- section 3 c library functions (like printf)

Directiones

Shell concept "working directory" cwd

The directory the shell is currently "in" current working directory

$\boxed{\text{pwd}}$ print working directory

The directory the shell is currently in cwd

typically - when shell starts Your cwd is your home directory

e.g. /home/url

Shell Concept

Absolut file name or path

Absolute name:

- start with /
- give the complete name/path to the file

/home/staff/jil/mydir/readme.txt

Relative names

- Do not start/
- name/path

$\boxed{\text{cd}}$ change directory/change current working directory

cd no argument → change cwd to home directory

ls List files/directory entries

#ls — shows what's in cwd

#ls <file> — shows entry for file —

#ls <dir> — shows all entries in dir

ls -l long listing

man ls for more info

specied Direct (short hand)

relative path

- . means current working directory
- .. means parent directory
- ~ means home directory

eg. cd ..

ls .

ls ../foo

cp copy files

usage cp src dest

makes a copy of the file src named dest

eg. cp readme.txt readme2.txt

cp r1.txt r2.txt r3.txt mydir ← dest directory

mv move (rename) files

usage mv src dest

renames files or dir src to dest

eg. mv readme1.txt readm2.txt

mv r1.txt r2.txt r3.txt mydir

rm remove file

usage rm file1 file2 ...

cat catenate files

usage cat file1 file2 ...

- concatenates the file and displaces the content
- use on text file if displaying the file on screen

eg. cat readme.txt

less View a file like cat but you can "scroll"
more usage less <file>
more <file>

view Any editor can be used to view a file
ssh Secure remote shell

Processes

In Unix (Also in windows) a process is a name for a running program.
much more details about OS in the OS class, but this class we need to know

- The OS will set aside memory space for the process
  A process cannot read/write/execute outside of its memory space
  (if it tries to in Unix you get "bus error" "segmentation fault"
- each process has its own list of open files
  In Unix (and Windows) this includes special files
  (0) standart input — Default place to read user input from
  (1) standart output — Default place to display output to

Redirection

For any command or program
You can send its output (standard out) to a file

In C no compiler error
  no exception thrown

One of three things will happen when run this
(1) The program runs file.
(2) Program does something strange
(3) Segmentation fault

Some Java Features that you should forget in C
- Type safety C has type checking
- Bounds checking
- Garbage collection
- Objects

```
(0)  /* comment */
(1)  #include <stdio.h>
(2)  main()
(3)  {
(4)  printf( "Hello World");
(5)  }
```

hello.c

(0) Comment // in later version of C
(1) There are no built-in function into the compiler.
  Everything comes from something somewhere
(2) when you compile C program, the function main() is
  executed when you run program.
(4) formatted print
  man 3 printf

```
# gcc hello.c
# ./a.out
  Hello world
#
```

shell prompt

hello2.c

```
/* hello V.2 */
#include <stdio.h>
main()
{
  char name[20];
  printf("what is your name? \n");
  gets(name);
  printf("Hello %s! \n", name);
}
```

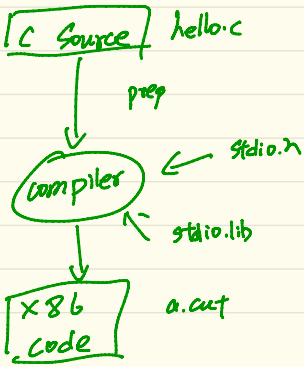
```
#gcc hello2.c
#./a.out
what is your name?
jim
Hello jim!
#
```

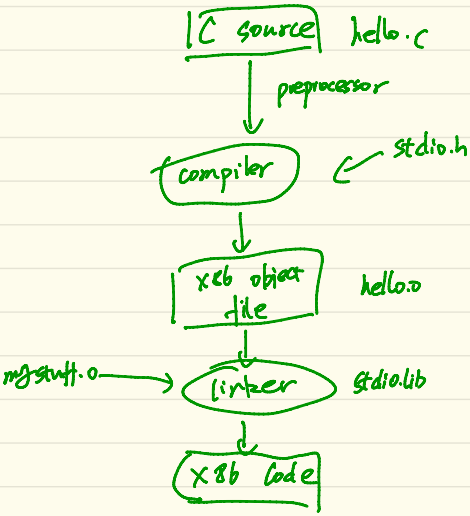
%s what is going on with printf?

- First parameter to printf is "format string"
- Remaining parameters to printf are printed according to the format specification. (% things) in the future string

%s : string
%d integer in decimal
%x integer in hexadecimal
%c character
%% %

Two way to compile





Another example

Convertic

```
#include <stdio.h>
#define KILOS-PRE_POUND 0.45359 ;
```

don't do the semicolon

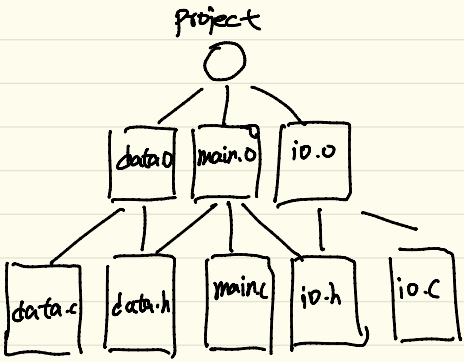
Sliping your C program

be sure no 2 files functions with same name in C

Similarly for global variables

Use .h file to make accessible to open files

Dependency and Dependency graph



How make utilize this sample makefile

```
project: data.o main.o io.o
	gcc data.o main.o io.o

data.o: data.c data.h
	gcc -c data.c

main.o: data.h main.c io.h
	gcc -c main.c

io.o: io.h io.c
	gcc -c io.c
```

tab character

makefile

shell prompt

```
#make
```

Make and multiple targets

```
all: program1 program2
	program1: --- ---

	program2: --- -
```

First target is used as the top dependency

```
#make
#make program1
```

```
clean:
	-> rm -f *.o program1 program2
```

C language summary

Types in C

void	– no type (for functions that return nothing, but also void pointers)	
char		8 bits
short ≡ short int	integer type	16 bits
int		32 bits
long or long int		64 bits
float		32 bits
double		64 bits

Integer types can be preceeded by unsigned or

User defineed type – structs

used to collect data together

- everything is public
- no method ⇒ no constructors

eg.

```
struct date
{
  short year;
  char month;
  char day;
};
```

← important

In this example, date is called the structure tag

Then we could have:

```
⋮
int x
struct date today;
```

structs can be nested

You can define alliases for types using "typedef"

- can give more descriptive names.
- can shorten type names.

usage:

```
typedef <type> <synocnym>
```

example

```
typedef int length-in-inches;
typedef int length-in-cm;
typedef struct date date-t;
  :
length-in-inches height;
length-in-cm x;
date.t today;
```

Pointers as a type

- In C we can have pointer variables (and types)
- If This helps, these are "address in memory where something is stored"

eg.

pointer to int -> address in memory where an int is stored.

- Type for a pointer

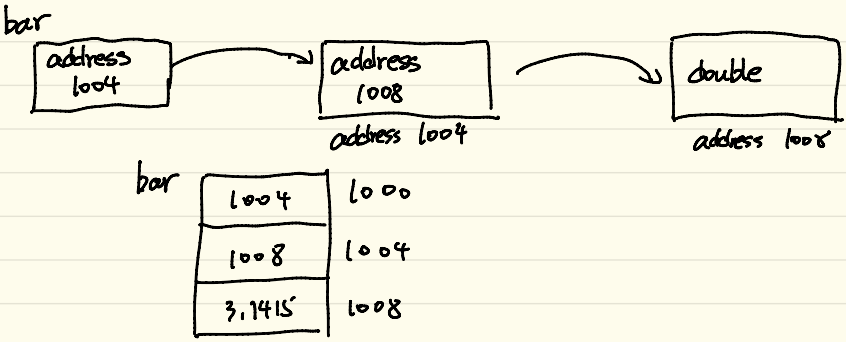
take the target type and add a "*" on the end

So

```
char *c      // x is a pointer to a char
long *foo
struct date *dptr
double **bar     // bar is a pointer to a pointer to a double
```

Integer

double ** bar



Arrays

- can declare (static sized) array as
  int A[10];
  A is an array of 10 ints
- can have array of pointers
  int * A[20]
  A is an array of 20 pointers to ints

Bitwise operators

- Applied to any integer type
- operate on the binary rep. of the integers

~:	one's complement	we have unsigned char (8 bit)
&:	bitwise and	unsigned char x = 0x96 (10010110)
|:	bitwise or	y = 0x28 (00101000)
^:	bitwise xor	
<<:	left shift	
>>:	right shift	

~x = 01101001
x&y = 00100000
x|y = 10101110
x^y = 10001110
x<<3 = 00110000

Assignment operators

$x = y + 1$

$x = 0$

$\underbrace{x = y}_{\text{first}} = 0$

Local and global variables

- Local variable are declared inside functions (or inside blocks inside functions) and can only be used inside that function block

```
#include <stdio.h>
void testvar()
{
  int a;
  printf("%d\n", a);
  a = 55;
}
void writevar()
{
  int b;
  b = 42;
}
main()
{
testvar();
testvar();
testvar();
testvar();
testvar();
writevar();
testvar();
}
```

Output

1	0
2	55
3	55
4	55
5	55
6	42
7	

Local static variables-

in C, static variables have nothing to do with the static modifier in Java

A variable that is declared static inside a function, is still scoped local to that function, but it maintains its value between function calls.

Global variables

are variables that are declared outside of any function and can be used at any time by any function.

- It is common to declare global variables at the beginning of the file
- Global variables should be avoided unless it cannot be avoided
- Use "extern" keyword to make variables declared in one file available to another.

```
# include <stdio.h>
void print_converted(int pounds){
    int stones = pounds/14;
    int ukllbs = pounds%14;
    float kilos_per_Pounds = 0.45359;
    float kilos = pounds * kilos_per_Pounds;
    Printf( ____ ); }
  main(){
    int tablesize, pounds, index;
    printf("Enter table size:");
    scanf("%d", &tablesize);
    for(pounds=10, index=0; index<tablesize; index++; pounds += 10)){
        print_converted (Pounds);
    }
    }
```

address of

```
main(){
    :
    :
}
print_converted( int pounds){
    :
}
```

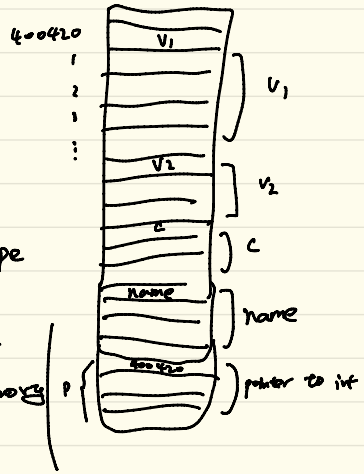
```
int swap(int a, int b){
    temp=b;
    b=a;
    a=tem;
}
```

does not work because a &b are location
we need pointer.

Program.c

```
#include <stdio.h>
main() {
    int v1;
    int v2;
    char c;
    char name[5];
    ⋮
```

In memory what does this look like.



Another type in C is the pointer type

for example int *P;

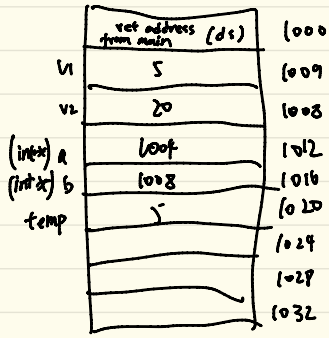
A pointer variable contains an address (of a variable) in memory

— Not the actual value.

Pointers and functions.

int * is a type

```
void swap (int *a, int *b){
    int temp = *a;
        *a = *b
        *b = temp;
}
int main(){
    int v1=5 ;
        v2=20 ;
    printf("%d %d\n", v1, v2);
    swap(&v1, &v2);
    printf("%d %d\n", v1, v2);
}
```



Prototypes and Header files

Because we may need prototypes for many functions, and those functions may be called from other C files, it makes sense to extract the prototypes out and make them part of a header file.

file.c has code for

```
int func1a (int int);
void func1b (int);
calls func2a
#include "file.h"
#include "file2.h"
func2b
func2c
```

file2.c has code for

```
void func2a();
int func2b (int);
int func2c (char);
call func1a
```

file.h

```
int func1a (int, int);
void func1b (int);
```

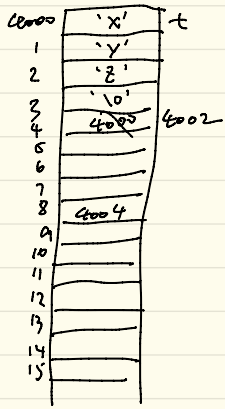
file2.h

```
void func2a();
int func2b(int);
int func2c (char);
```

Array and pointers

```
#include <stdio.h>
Int main() {
    char v[20];
    v[0] = 'a';
    v[1] = 'b';
    v[2] = 'c';
    printf ("%s\n", v);
    return 0;
```

```
char  t[4]
char  *p
char  **q;
t[0] = 'x';
t[1] = 'y';
t[2] = 'z'
t[3] = '\0'
 p = &t[0]
printf("%s\n",p);  //xyz
printf("%s\n",t);  //xyz
```



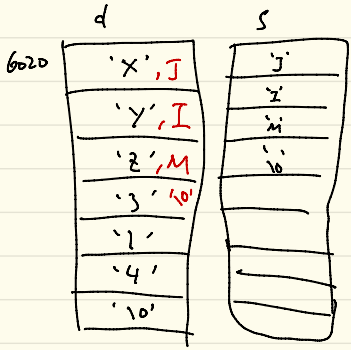
t, &t, &t[0] are all 4000

```
*p++ = t[0];
```
P after this statement is ~~4001~~

```
*p++ = '\0';
```
P after this statement is 4002

```
printf("%s\n",p);   // z
q=&p;
printf("%s\n",*q++);
}
```

```
void strcpy (char *s, char *d){
    while (*d++ = *s++);
```



Suppose d declared char d[3];

void strncpy (char *s, char *d, int n)

void strcpy (char s[], char d[])

'0' = 42

char *s ≡ char s[]

Standard I/O again

We will use fprintf and fscanf from stdio.h

fprint (File pointer, format string, ____); ✗

Standard file pointers (File descriptions)

stdout

printf is the same as fprintf (stdout, ··) fprint (stdout, ···);

stderr

fprintf (stderr, ___, ____). fprintf (stder, ···)

This is Useful as ~~this~~ since it does not get redirected to output file for > and >>

We can use fscanf, fsprintf to read and write file.

eg. use command line arguments to specify output/input file names

#include <stdio.h> — Note that this defines "NULL" to be 0 e.e #define NULL 0

```
int main( int argc, char *argv[]) {
    FILE *outfile;
    // Associate pointer with a FILE description
    outfile = fopen ("Zebra.txt", "w");
```

- open for writing - will overwrite file
- "a" used to append to end of file

filename in file system

eg. /home/jil/textdata.txt

→ returns a pointer to a structure that "describes" the file.

It will return NULL pointer if for some reason the file could not be opened

So, we want an if statement to checks for error

```
If (outfile == NULL){
  fprintf (stderr, "Could not open Zebra file\n");
  exit (1);
}
fprintf (outfile, "square Table \n");
for (int i=0; i<10; i++){
  fprintf (outfile, "%d %d\n", i, i*i);
}
 close (outfile);
}
```

I/O is buffered

You can flush the buffer for output with flush() flush(FILE *)

Note: We can write code like this.

```
    ⋮
    FILE *f = fopen("out.txt", "w");
    stdout = f;
    printf ("Hello, World \n");
```

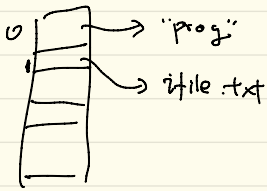
How to read from a file example:

```
#include <stdio.h>
main (int argc, char *argv[] ){
    FILE *readfile;
    if (argc > 1){
        readfile = fopen(argv[1], "r");
    }else{
        readfile = stdin;
    }
    float input1, input2
    fscanf (read, "%f %f ", &input1, &input2);
}
```

⇒ produce executable prog

run with prog ifile.txt

show do error checking



Quicksort example

want in C qsort (array, length)

Java

returns <0 if a<b

=0 if a=b

>0 if a>b

```
#C
input comp(int a, int b) {
    return a-b;
}
```

As strange as it may seem, you can have a pointer type variable that points to a function (or even pointer to pointer to functions)

eval.c

```
float plus (float a, float b) {return a+b;}
.   minus (      .          .
.   multiply (   .          .
.   divide (     .          .
float evaluate(float a, float b, char op){
```

```
float result;
switch (op){
    case "+"
        result = plus(a, b);
        break;
    case "-"
        :
```

float evaluate (float a, float b, float (* opfunc)(float, float))

main() {

func = +
evaluate (1.0, 3.0, *func);

If you had to write prototype for evaluate

float evaluate (float a, float b, float (*)(float, float));

Now back to quicksort and a generic computer

Now back to quicksort and our generic computer

void qsort (void *v[], int left, int right, □)

int (*comp) (void *, void *)

How would you compare location (index)
"c" in array with index "left" in an array

int c = comp (v[i], v[left])

Describe those

char **argv — a pointer to an array of characters (char type)
a pointer to a pointer to a char type

int (*data)[13] — a pointer to an array of ints
int * data [13] — An array of 13 int pointers

Structures and pointers to structures

Suppose we want to represent the point (in 2 space) using x, y coordinates

In C we use structures E.g.

```
struct point {
    int x;
    int y;
};
```

```
struct point {
    int x;
    int y;
} p1, p2, p3
```

```
struct {
    int x;
    int y;
} p1;
```

Use tag to define structure variable

```
struct point p1;
```

Unlike Java – structs are not reference

```
p1 = p2
```

```
struct point maxpt = {320, 200};
```

Use the "." (selector) to access components of the associate structure

eg. `printf("%d %d\n", maxpt.x, maxpt.y);`

You can also nest structures

```
struct rectangle {
    struct point p1;
    struct point p2;
};
struct rectangle screen;
screen.p1.x
screen.p1 = maxpt;
```

Add two points together

```
struct point
void addPoint(struct point p1, struct point p2)
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
{
```

addPoint(x, y) ← does not change x

```
struct point addpoint(struct point p1, struct point p2)
    struct point temp;
    temp.x = p1.x + p2.x;
    temp.y = p1.y + p2.y;
    return temp;
}
    a = addpoint(a, b);
```

Use pointers to structure to fix this

```
struct point * ppoint;
```

says that ppoint is a point variable that points to a structure of tag type point.

*ppoint dereferences ppoint to a struct

(*ppoint).x → component of structure.

```
struct point origin = {0, 0};
struct point *ppoint;
ppoint = &origin;
```

← is required

```
printf("origin is %d %d\n", (*ppoint).x, (*ppoint).y);
struct point q = *ppoint;
```

Because pointers to structures are used often we have struct to use

e.g. ppoint→x ≡ (*ppoint).x;

① The only legal operators on structure are

ⓐ copying (assignment) on structures (including parameter passing and return statements)

ⓑ taking the address of (&)

ⓒ Accessing its members with the (.)

② Arrays of structures work just like you think.

eg. struct histbin { 32 bits 4 bytes
    char *label;
    int count; 32 bits 4 bytes
};
struct histbin histagram[100];

sizeof(int) 4
sizeof(struct histbin) 8
sizeof(struct hisbin *) 4

```
void ExactChange (int userTotal, int conVals[]) {
    conVals[0] = userTotal / 100;
    userTotal = userTotal - 100 * conVals[0];
    conVals[1] = userTotal / 25;
    userTotal = userTotal - 25 * conVals[1];
            :
}
main() {
    int change[5]
    ExactChange (232, change)   & change[0]
    printf("Address returned = %d\n", change[0]
```

```
struct node {
    int data;
    struct node *next;
}
void push (struct **s, int d ) {
    struct node *toadd =malloc (sizeof(struct node ));
    (*toadd ).data =d;    (toadd -> data=d;)
    toadd -> next = *s ;
    *s = toAdd;
```

```
int pop (struct node **s) {
    int toReturn;
    if (*s == NULL) {
        fprintf (stderr, "STACK UNDERFLOW \n")
        return 0;
    }
    toReturn = (*s) -> data
    struct node *temp;
    *s = (*s) -> next
    free(temp);
    return toReturn;
}
```

dll.h

```
#ifndef DLL_H_
#define DLL_H_
#include NULL
#define NULL 0
#endif

typedef struct node_t NODE;
typedef struct node_t *NODEPTR;
struct node_t {
  NODEPTR prev,next;
  void *data;
}
typedef struct list_t LIST;
typedef struct list_t *LISTPTR;
struct list_t {
    NODEPTR head // Doublylinked list with Dummy node
                 and is a linked list.
    int numItems;
}
LISTPTR createList();
int add (LISTPTR lst, void *item);
void *get (LISTPTR lst, int index)
void *rem (LISTPTR lst, int dex)
int emplyList (LISTPTR lst);
void deletlist (LISTPTR lst );
#endif
```

Sep 29.

binarytree.h

```
#include binarytree_h_
#define binarytree_h_

#ifdef NULL
#define NULL 0
#endif

typedef struct node_t NODE;
typedef struct node_t *NODEPTR;
struct node_t{
   void *data;
   NODEPTR left, right;
};

typedef struct tree_t TREE;
typedef struct tree_t *TREEPTR;
struct tree_t {
   NODEPTR root;
   int numItems;
};
TREEPTR CreateTree();
int addTreeToItem
```

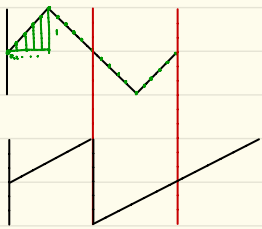
binarytree.C

```
#include (stdio.h)
#include <stdlib.h>
#include "binarytree.h"

TREEPTR createTree( ){
   TREEPTR t = malloc (sizeof (Tree));
   t->numItems = 0;
   t->root = NULL;
   return t;
}

int addTreeItem(TREEPTR t, void *item, int (*comp)(void*, void*)){
   t->root = addrec (t->root, item comp )
   t->numItems += 1;
   return 1;
}
NODEPTR addrec (NODEPTR r, void * itemtoadd, int (*comp)(void*, void*)){
   if (r==NULL){
      NODEPTR addnode = malloc (sizeof (NODE));
      addnode -> data = itemtoadd;
      addnode -> left = NULL;
      addnote -> right = NULL;
      return addnote;
   }else{
      int c = comp (itemtoadd, r->data);
      if (c==0); return r;
      else if (c<0){
         r->left = addrec (r->right, itemtoadd, comp);
      } else { r->right = addrec (r->right, itemtoadd, comp); }
      return r;
```

```
int a,b;
int *p1, *p2
a=100, b=10;
  p1 = &a;
  p2 = &b;

printf("%d, %d\n", *p1, *p2)      // 100, 10
```

s[0]	1	
s[1]	2	
s[2]	3	3

20/1=5



20 $\frac{1}{5}$

```
mix (sound *s[], float w[], int c){

for (i=0; i<3; i++)

while( cur < num ){
sum += s[0].sample[cur] * w[0]
        s[1].sample[cur] * w[1]
        s[2].sample[cur] * w[2]
```

Sound *filter(Sound *s, float fir[], int c){

samples: 0.1 0.3 0.4 0.1 0.2

fir: 1 0 1

c = 3

=> output: 0.1*1 + 0 + 0 = 0.1

0.3*1 + 0.1*0 = 0.3

0.4*1 + 0.3*0 + 0.1*0 = 0.4

0.1*1 + 0.4*0 + 0.3*0 = 0.1

0.2

0.3

0.2

0.7

```
while(Cur < length){
    for(int i=0; i<c; i++){
        int n=Cur;
    sum += fir[i] * p.sample[n];
        n--;

    i=1
```

10/6/2020 C++

10/8/2020

C++ additions (The little things)

bool type for boolean value

along with that you get constants true and false

bool is still an "integer" type and you can assign bool types to integers

int i = false;

still have single-line comments

still can declare variables inside blocks (for, while loops etc)

casting ~~can~~ do it in the C way

When you declare a struct, you can use the name of the struct to declare variables or parameters

e.g. struct date {

short year;

char month, day;

C long days_between (struct data start, struct data end);

C++ date today

Struct type checking in general enumerated type for example.

e.g. enum permission {

read = 4;

write = 2;

execute = 1;

};

permission group = wrk;

permission group = 2;

permission group = permission(2);

Assignment to void pointer still need to cast.

All other pointers require cast.

C++ Big differences

- I/O

  You can still use stdio.h to do input/output

  C++ uses streams (object)

  - streams have methods
  - can get/put items on/off the stream

Const

References

We can define reference varibles/parameters

sort of like pointers except

- must be initiallized when they are declared
- cannot be NULL
- cannot be reassigned

Classes

Defined like a struct except with keyword "class"

can have member methods inside

Function overloading

In C++ we can have two or more functions with same name

The name is related to a specific function by the number and type of parameters

Allowed

```
int square (int x);
double square (double x);
complex square (complex x);
int sum (int a, int b);
int sum (int a, int b, int c);
```

Not Allowed

```
int log (int a)
double log (int a)
```

Parameters can have default values

```
int increment (int a, int delta = 1 ){
        return a+delta;
}
x= increment( b, 4 )
b= increment (b)    ≡    b=b+1
```

In C++ you can overload

- functions methods
- member functions/methods
- Constructors
- operators

Things like + += + -- ->

g++ compiler

e.g. g++ -g -c -o myfile.o myfile.cpp

standard File extensions

- .h header files
- .cpp C++ source file

Hello World in Java

```
public ...
  public ...
    System.out.println ("Hello World");
```

class system — static variable out which is an object

Hello World in C++

```
#include <iostream>
using namespace std;
int main(){
  cout << "Hello World\n";
  return 0;
}
```

```
cout << "something"
cout << 120 ;
cout << x ;
```

— cout << "something" << 120 << x;

To add lines

1) '\n'
2) endl

ex — cout << "This is" << "my test"

output

```
This is
My test
```

no more C strings

```
#include <string>
#include <iostream>
Using namespace std;
int main(){
  string mystring;
  cout << "What is your name?";
  getline(cin, mystring);
```

lower case S

mystring.length
mystring.append
mystring.

Reference part 1

```
int i=7;
int &j=i;
j++;
j+=3;
cout << i;
```

& Initialize and for ints
i & j

&& logical and if (a<1 && b>4)

& address of
e.g. int *p=&i;

& reference

objects

10/13

Java syntax and semantics requires a single class per file and class names must match file names

C++ doesn't care

There are some conventions

Two files for each class

- .h class declaration, method prototypes, macros, inlined functions, member variables
- .cpp implementation of method

A very simple example: (no header file)

```
class Box{
  private:
    ?
  public:
    ?
```

A closer look of Box

```
class Box{
    private:
        int val;
    public:
        int myVeryBadIdea;
        Box (int v){
            val = v;
        }
        int value(){
            return val;
        }
};
```

keyword named "inline"
request to compiler to not make function call
but rather fill n code directly where it is being used.

class example with header file.

counter.h

```
# ifdef COUNTER_H
# define COUNTER_H
class Counter {
    private:
        int value;
        int maxValue;
    public:
        void SetValue(int);
        Counter();
        Counter(int);
        void increment();
        void decrement();
        int getValue();
        inline int getMaxValue() const {
            return maxValue;
        }
};
#endif
```

This method cannot make changes to the class. i.e. member variables

because of the ;

cat test.cpp

```
# include <iostream>
# include "counter.h"
using namespace std;
int main(){
    counter c1;
    counter c2(10);
    c1.
```

counter.cpp

```
# include <iostream>
# include <limits>
# include "counter.h"

Counter::Counter(){
    value=0;
    maxValue=INT_MAX;  // come from limits
}
Couter::Counter(int mv){
    value=0;
    maxvalue=mv;
}
Void Counter::Counter(int val){
    if ((val >=0) && (val<=maxvalue)){
        value=val;
    }else{
        cerr << "set value is out of range" << endl;


Void Counter::increment(){
    if (value < maxvalue){
        value++;
    }else{
        cerr << "Counter ... " << endl;
```

makefile

```
cntrtest: counter.o cntrtest.o
-> g++ -g -o cntrtest couter.o cntrtest.o
counter.o: counter.cpp counter.h
-> g++ -g -c couter.cpp
cntrtest.o: cntrtest.cpp counter.h
```

Const (The real story)

Simple stuff

```
const int C = 42;
```

Why this over #define #define C 42

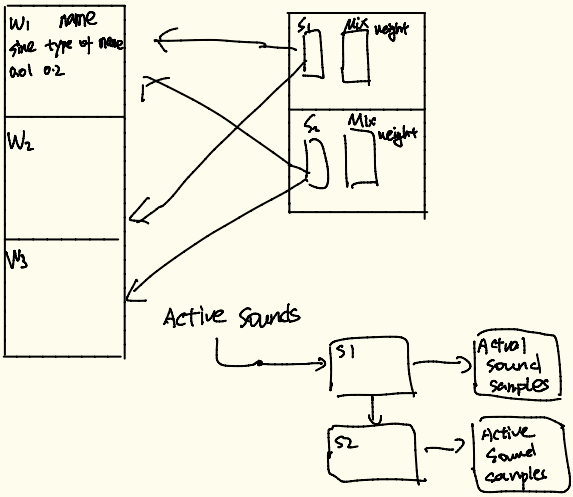
used by compiler for optimizations
type checking.

Something to think about

```
const int *C1;      a variable pointer to a constant integer.
int const * C2;     a variable pointer to a constant integer
int * const C3;     constant pointer to a variable integer
int const * const C4;
```

same

Rule of thumb
const applies to whether is on the left
unless there is nothing in which case use
what is on the right.



sample rate
0.9 16000

sound name	note #	start time	duration
S1	38	0	0.25
S1	39	0.5	0.25
S1	40	1.0	0.25

Sound	note	time	duration
S1	38	0	0.5
S2	40	0	0.20
S1	36	0.25	0.5
S1	48	0.25	0.25

current time in terms of absolute sample#
current time = 0

Reference types again

Recall example

```
int i=7;
int &j=i;
j++;
i=3;
```

i = 7 (11) j = (11)

Rules of Reference

- References can never be NULL. Rey must always refer to a legitimate variable/object
- Once established, a reference can never changed to make it reference something else,

  A reference does not require any mechanism to dereference.

Not only can you have variables that are reference, you can also pass parameters to functions/methods by reference.

A reference may be used as the target of an assignment statement. This seems odd and normal until you consider that a function may return a reference.

Ex:

```
int val[100];
int & index (int i){
    return values [i+2]
}
  :
main(){
    index(27)=12;
}
```

What does this do?

String class example from C++

```
class string{
  :
  private:
    char * buffer;
    int length;
  public:
    char & operator[] (unsigned int index){
      return buffer[index];
    }

main(){
    String test = Readline(file);
    test[0] = 'P';
```

3 special method

destructor

copy Constructor

operator =

Triple.h :

```
class Triple{
    private:
        int v1;
        int v2;
        int v3;
    public:
        ~Triple();
        Triple(int, int, int)
        Triple(const Triple&);

        const Triple & Triple::operator
                =(const triple & rhs);
```

Tripple.cpp

```
Triple::~Triple()
{
}
Triple::Triple(int p1, int p2, int p3):
        v1(p1), v2(p2), v3(p3)
{
}

Tripple::Tripple(const Triple & rhs):
        v1(rhs.v1), v2(rhs.v2), v3(rhs.v3)
{
}

const Triple & Tripple::operator=(const Tripple & rhs)
{
    v1 = rhs.v1;
    v2 = rhs.v2;
    v3 = rhs.v3;
}
```

IntBox.h

```
#ifdef INTBOX_H
#define INTBOX_H

class IntBox {
    private:
        int * storeIvalue;
    public:
        IntBox(int);
        IntBox(const IntBox &);
        ~IntBox();
        const IntBox & operator=(const IntBox&);
        int value();
};
#endif
```

IntBox.CPP

```
IntBox::IntBox(int intvalue=0){
    storedvalue = new int(intvalue);
}

IntBox::IntBox(const IntBox & rhs){
storedvalue = new int(*(rhs.storedvalue));
}

IntBox::~IntBox(){
    delete(storedvalue);
}

const IntBox&IntBox::operator=(const IntBox& rhs){
    if (this != &rhs){
        *storedvalue = *rhs.storedvalue;
    }
    return *this
}

int IntBox::value(){
    return *storedvalue;
```

10/27

3 special methods
- destructor
- Copy Constructor
- Operator=

Triple.h

```
class Triple{
private:
    int v1;
    int v2;
    int v3;
public:
    ~Triple()
    Triple(int,int,int);
    Triple(const Triple&);
```

Tripple.cpp

```
Tripple::~Triple(){}
Tripple::Triple(int p1, int p2, int p3): v1(p1),v2(p2),v3(p3){}
Tripple::Tripple(const Tripple & rhs): v1(rhs.v1), v2(rhs.v2),v3(rhs.v3)
{
}
const Tripple & Tripple::operator=(const Tripple & rhs){
    v1= rhs.v1;
    v2= rhs.v2;
    v3= rhs.v3;
}
```

```
const Tripple & Tripple::operator=(const triple & rhs);
```

Consider a linked list class

```
struct Node{
    int data;
    Node *next;
};
class SLL{
private:
    Node *head
public:
    SLL & operator=(const SLL & rhs){
        //delete linked list
        // copy linked list from RHS
    }
```

```
SLL a;
SLL b;
  :
a=b
```

a=a ⇒ Problem ⇒ Fix ⇒

```
SLL & operator=(const SLL& rhs){
    if (this != & rhs){
        // ...
        // ...
    }
    return *this;
}
```

11/03

wave * = new sinwave("__");
Triangle
Square
SawTooth

put 4 different wave subclasses?
put them in wave.h file?

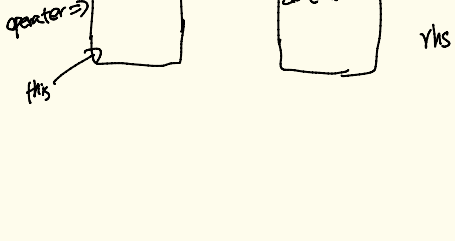
Another example:

```
class A {
    public:
        A(): dataOne (2){}
        virtual void whoami(){
            cout << "class A" << endl;
    private:
        int dataOne;
};

class B : public A {
  public:
    B(): dataTwo(9){}
    virtual void whoami(){
        cout << "class B" << endl;
    }
  private:
    int dataTwo;
};
```

```
main {
    A objectA();
    B objectB();
    object A = object B
    object A = whoami ();
}
```

what is output?

assignment = operate



A * objectA = new A();
B * objectB = new B();
object A = object B
object A → whoami();

No slicing - but you can only call stuff defined in the A type.

slicing does not occur with pointers and references

A & refA = B;
refA.whoami()
⇒ output "class B"

B* ptrB = new B();
A* ptrA = ptrB;
ptrA → whoami(); output "class B"

slicing can be non obvious

```
class Ick {
    A foo(B *x) {
        ⋮
        return (something that is an A)
};

Ick *c = new Ick();
B *b = new B();
A a = c->foo(b);
A *a = &c->foo(b)
```

use pointer or reference

calling constructors in a super class

Java:

```
public class Box {
    private int val;
    public Box(int v){
        val = v;
    }
}
public class BigBox extends Box{
    public double dval;
    public BigBox(double d, int v){
        super(v);
        dval = d;
    }
}
```

C++:

```
class Box{
    private:
        int val;
    public:
        Box(int v) : val(v){}
};

class BigBox : public Box{
    private:
        double dval;
    public:
```

2:46 pm

Combing Constructors

Java

```
public class A
    public int A (int i){
        :
    }
    public A(int i, int j){
        this (i);
        :
```

C++:

```
class A{
    public :
        A(int i){
            :
        }
        A (int i, int j){
            A(i)
```

What can be called is defined in the class
where it gets called is defined by the object type

Visibility modifiers
- public
- private
- protected

Java – package

C++ allows the subclasses to change to visibility of the super class (in some sense)

eg.

```
class Parent{
    public
      virtual void test()
        cout << "In Parent" << endl;
    };
class Child : public Parent
  private:
    //void test(){
    // cout << "In Child" << endl ;
    //}
  };
```

```
parent *p = new Child();
p -> test();
```

syntax error or not
if not – what is output?

– No error in syntax and compiles.
outputs "In Child"

```
Child *c = new Child();
c -> test();
```

syntax error.

What happens if I comment out test in Child

class B: X A

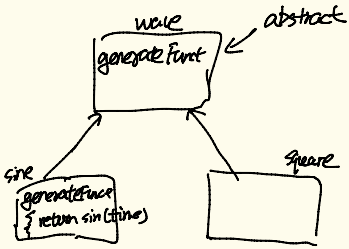
Public
Private
Protected

Type of inheritance

Base type membership access	Public	protected	private
Public	public in the derived class	protected in the derived class	
protected			
private	Hidden from derived class		

generateFunction

sin(time)
square (



sine extends wave

wave a = new sine();

```
cout << fixed << setprecision(2);
cout << ---- << endl;
```

```
int a = 5;
int b = 8;

int& ref = a;
ref = b;

// a=8, b=8
```

```
int* ref = &a;
*ref = 2;
ref = &b;
*ref = 1;
```

Exam Topic

source file, object files, executable files, preprocessor, compiler, linker. (Relationships)

Makefile
- format
- dependence
- creating targets
- multiple executables
- other target clean
- concept shift

git (not syntax)

C main functions
- C common stuff with Java
- C preprocessor #define
  #ifdef #endif

pointers!

stack memory - how it works with function calls

I/O stdio, stderr, stdout, scanf - File stuff fscanf fprintf

function pointers

heap memory - malloc & free

structures & unions

C++

I/O

operator overloading

Allocation new and delete

The trio – copy constructor, dest.. , operator =

references versus pointers

defining class .h versus .cpp – how does this work

const and

immutable classes

reference counting

Inline – what it does what it means

inheritance / poly morphism types of inheritance

pointers versus references versus declared object

nested classes

friend def & why/how to use

Static variables/methods

visibility modifiers

Garbage collection stuff

sample rate = num of samples in one second

TotalSamples SR x duration