int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}

\[
\begin{bmatrix}
\cos 90^\circ & \sin 90^\circ \\
\sin 90^\circ & \cos 90^\circ
\end{bmatrix}
\begin{bmatrix}
a_1 \\
a_2
\end{bmatrix} = \begin{array}{c}
\end{array}
\]
Pass By Value Reminder in C

```c
int f( int );
int g( int, int );

int main( int argc, char **argv )
{
    int a = f( 5 );
    int b = g( a, b );
    return a + b;
}

int f( int a )
{
    int b = a - 1;
    int c = b / 2;
    a = a + c;
    return a;
}

int g( int a, int b )
{
    return f( a ) +
           f( b );
}
```
Main()

- Return from main? That rings a bell, isn't that what functions do? So does that make main() a function?
  ```c
  int main( int argc, char **argv )
  {
      /* variables and program stuff... */
      return 0;
  }
  ```
- Yes! So you've been writing functions all along. Hah!
- That makes “int” here the “return type” and that “int argc, char **argv” stuff the variable list.
- But what “called” main? And how do we retrieve the return value?... A discussion for another week...
C Jane, C Jane Comment

- Yes, comments.
- The top of *all source code files* should have big comments stating the purpose of the program, author, date, etc.
- Any sufficiently large or complex chunk of code should have an explanation above it.
#include <stdio.h>

int main( int argc, char **argv )
{
    int a, b, c=0;

    for(a=0; a<10; a=a+1) {
        for(b=0; b<10; b=b+1) {
            while(c==0) {
                fprintf(stdout, "%d %d\n", a, b);
            }
        }
    }
    return 0;
}
```c
#include <stdio.h>

/*
 * Sean Cusack
 * CS102 Fall 20xx
 * Project 99
 * aa-bb-20cc
 */

int main( int argc, char **argv )
{
    int a, b, c=0;
    /*
     * This bit of code is very complex, and if I didn't document it
     * here, I'd never remember what it did after a night before.
     */
    for(a=0;a<10;a=a+1) {
        for(b=0;b<10;b=b+1) {
            while(c==0) {
                fprintf(stdout,"%d %d\n",a,b);
            }
        }
    }
    return 0;
}
```
Comments from the Peanut Gallery

• All projects henceforth must have a header comment like the one just given.

• All projects must also properly document any portion of code that is complex and not immediately obvious. On this point, some leeway will be given early in the semester, but continue to practice and build your own commenting style throughout the year.
One such form of clarity is called “proper indenting”.

There are numerous styles of indenting, any one is fine as long as it's consistent throughout a program:

- if(a==b) {
   
   ...

   }

- if(a==b) {
   
   ...

   }

- if(a==b) {
   
   ...

   }

- if(a==b) {
   
   ...

   }
int function( int a, int b, float d )
{
    int c;
    c=a+b;  d=c/a;
    while     (c!=6)
    {
        a=a+1
        ;
        b=b+c;
        if(b==a)
        {
            d=d/2.0;
            if(c==a)
            {
                d=d+1.0;
            } else {
                fprintf(stdout,"This is spot's dump, er, lump,
of code, generously shared with the lawn today.\n");}}}
}return c;
Play Dead, Spot!

int function( int a, int b, float d )
{
    int c;
    c=a+b;  d=c/a;
    while(c!=6){
        a=a+1;  b=b+c;
        if(b==a){
            d=d/2.0;
            if(c==a){
                d=d+1.0;
            }else{
                fprintf(stdout,"Here, spot played dead and sat on the left boundary of the page to take a nap.\n");
            }
        }
    }
    return c;
}
int function( int a, int b, float d ) {
    int c;
    c=a+b; d=c/a;
    while(c!=6) {
        a=a+1; b=b+c;
        if(b==a) {
            d=d/2.0;
            if(c==a){
                d=d+1.0;
            } else {
                fprintf(stdout,"Here, spot played nice.\n");
            }
        }
    }
    return c;
}
Roll Over!

```c
int function( int a, int b, float d )
{
    int c;
    c=a+b; d=c/a;
    while(c!=6)
    {
        a=a+1; b=b+c;
        if(b==a)
        {
            d=d/2.0;
            if(c==a)
            {
                d=d+1.0;
            } else
            {
                fprintf(stdout,"Here, spot played nice.\n");
            }
        } else
        {
            fprintf(stdout,"Here, spot played nice.\n");
        }
    }
    return c;
}
```
int function( int a, int b, float d )
{
    int c;
    c=a+b; d=c/a;
    while(c!=6)
    {
        a=a+1; b=b+c;
        if(b==a)
        {
            d=d/2.0;
            if(c==a)
            {
                d=d+1.0;
            }
            else
            {
                fprintf(stdout,"Here, spot played nice.\n");
            }
        }
    }
    return c;
}
#include <stdio.h>
int main( int argc, char **argv )
{
    int a, b, c=0;
    for(a=0;a<10;a=a+1) {
        for(b=0;b<10;b=b+1) {
            while(c==0) {
                if(a>b) {
                    if(b>0) {
                        fprintf(stdout,"%d %d\n",a,b);
                        c=c+1;
                    }
                } else {
                    c=c-1;
                }
            }
        }
    }
    return 0;
}
#include <stdio.h>
int main( int argc, char **argv )
{
    int a, b, c=0;
    for(a=0; a<10; a=a+1) {
        for(b=0; b<10; b=b+1) {
            while(c==0) {
                if(a>b) {
                    if(b>0) {
                        fprintf(stdout,"%d %d\n", a, b);
                        c=c+1;
                    } else { 
                        c=c-1;
                    }
                } else {
                    c=c-1;
                }
            }
        }
    }
    return 0;
}
Don't Do That!

```
#include <stdio.h>
int main( int argc, char **argv )
{
    int a, b, c=0;
    for(a=0;a<10;a=a+1) {
        for(b=0;b<10;b=b+1) {
            while(c==0) {
                if(a>b) {
                    if(b>0) {
                        fprintf(stdout,"%d %d\n",a,b);
                        c=c+1;
                    } else {
                        c=c-1;
                    }
                } else {
                    c=c-1;
                }
            }
        }
    }
    return 0;
}```
Optical Illusions
Stay, Spot

• Also henceforth for all upcoming projects, all C code must be properly indented using one of the three major conventions

• This is in addition to other rules... reminder:
  – Programs must compile and run properly on students
  – All files must have name, class, date, assignment#
  – Files will continue to be divided, i.e. main.c, functions.c, functions.h
  – Cite all assistance [see next slides]
Honor Policy

- Specifically for “coding”, i.e. programming
- It is imperative that you start from a “blank file” all by yourself
- You cannot copy anyone else's work and “start from there”
- You can help each other with concepts, but not “how to do it”
  - No typing for someone else
  - No reading someone else's work while typing
Honor Policy [cont'd]

• What you can do is copy your own work
• These labs and projects have been designed to be useful to each next project
• If you're stuck, make a copy of a previous lab that seems similar, and ask yourself how it is different, then merely make a few changes
• You may copy my examples from lectures, they are meant to be used
Honor Policy [cont'd 2]

- What *is* necessary is citations
- In the comments at the top of the program, include who helped, when, and with what part of the project (be specific)
- This protects you from claims of plagiarism and as future engineers, you need the practice
Signature & Citations Example

#include <stdio.h>

/* Sean Cusack - 20131009 - Homework X */
/* Citations: main.c - Bob Hopkins helped me with function foo() by moving a loop from main into the function with me. */

int main( int argc, char **argv )
{
    foo();
}

The great Void

- In case you were wondering, you *can* make functions that don't return int's, but rather return "nothing":
  ```
  void function( ... );
  ```
- And likewise functions that take no arguments should look like:
  ```
  ... function( void );
  ```
Team Building

- Work is often broken down by function: "hey, I'll work on \( y( \text{int } x ) \) and \text{main}()\), and you work on \( y\prime( \text{int } x ) \) at the same time, and we'll put our stuff together later"

- This is good in theory, until person two realizes his mistake and changes his function's prototype to:
  
  \[
  \text{int } y\prime( \text{int } x, \text{int } h )
  \]

  without telling his partner...

- This is like telling your engineering design partner to build a piece that fits into a 2" hole because that's what your half will have ready, then only drilling a 1" hole
Share Nice, Kiddies

- The best way to share nice is to have ONE and only one copy of the prototypes (think shared blueprints)

- This is best accomplished by putting prototypes in a separate file, that must end in .h

- For example, $y'(x)$ is defined in $y.c$:
  ```c
  int yprime( int x ) { return 2 * x; }
  ```
  and the prototype is in $y.h$:
  ```c
  int yprime( int x );
  ```
Being Inclusive

- However, both person 1 and person 2 need that prototype in their code... cut-and-pasting won't work because that'll lead to the first problem again, so instead, we have the compiler do the work:
  
  ```c
  #include "y.h"
  (notice the "quotes" not the <angle brackets>)
  ```

- If this is in each file, it is as though "the full contents of file y.h" are in place of the #include

- If y.h ever changes, the next time the file that includes it is recompiled, it'll pick up the change
Compiling a Laundry List

- So if person 1 and 2 both share a y.h file in some directory, and
- Person 1 writes main.c that contains main() and y(x), and
- Person 2 writes yprime.c that contains yprime(x,h)
- Then to make one program out of it, just put all the C files on one gcc line (not the .h file-- that gets pulled into each automatically):
  
  \texttt{gcc main.c yprime.c -o program}

- Of course, this works just as well if you're the only one making the files...
Plucking out Headers

• If we want to further pick it apart, not just into functions, but separate files, break it up:
  – Copy the function definition and prototype to func.c
  – Now, we don't want the prototype copied in two places, we want one place shared, so first copy the prototype to func.h
  – Then replace the prototype in both main.c and func.c with:
    #include "func.h"
/* main.c */
#include <stdio.h>
int func( int max );
int main( int argc, char** argv ) {
    int x = func(10);
}

/* func.c */
#include <stdio.h>
int func( int max );
int func( int max ) {
    int i = 0;
    while( i < max ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
    return 42;
}
/* main.c */
#include <stdio.h>
#include "func.h"
int main( int argc, char** argv ) {
    int x = func(10);
}

/* func.c */
#include <stdio.h>
#include "func.h"
int func( int max ) {
    int i = 0;
    while( i < max ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}

/* func.h */
int func( int max );
Live Example

 cd cs102e/lab-6
 cp common/main.c .
 cp common/func.c .
 gcc main.c func.c -o main
 ./main

 0
 1
 2
 ...
 9
Be evil

• Now you get to play the evil teammate... edit func.c so that func (and its prototype) now takes an int max, and uses it in the loop:

```c
int func( int max );
int func( int max ) {
    int i = 0;
    while( i < max )
```

• Now you're the good teammate, go try to recompile and run:

```sh
#> gcc main.c func.c
#> ./main
```
Be good

• Now we'll straighten things out... replace the prototype of func in both files with:
  \#include "common/func.h"

• And recompile:

  \$> gcc main.c func.c -o main
  func.c:3: error: conflicting types for 'func'
  common/func.h:1: error: previous declaration of 'func' was here
  func.c:3: error: conflicting types for 'func'
  common/func.h:1: error: previous declaration of 'func' was here
Be prepared

• Why did the evil teammate's one blow up?
  
  ```
  #> cat common/func.h
  int func( void );
  ```

• Because they are now using the shared blueprint, and it's not what the evil one designed to match

• Now I'll change the blueprint... (wait for me, as I change common/func.h)
Be knowledgeable

...And recompile:

```bash
$ gcc main.c func.c -o main
main.c: In function `main':
main.c:3: error: too few arguments to function `func'
```

Why?

```bash
$ cat common/func.h
int func( void );
```

So the good teammate is still broken, but at least knows why – the blueprint changed. So rather than having to debug something strange, it's immediately clear.
Random Integers

For use in your project, this is how to produce a random integer (from -2.1 billion to +2.1 billion):

```c
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char **argv )
{
    int random;
    srand48(getpid()); /* call this once only, in main */

    random = mrand48(); /* either put in a variable first */
    fprintf( stdout, "random integer = %d\n", random );

    /* or just use it like this */
    fprintf( stdout, "new random integer = %d\n", mrand48() );
    return 0;
}
```
Random Example 2

main.c:
#include <stdio.h>
#include <stdlib.h>
#include "func.h"

int main( int argc, char **argv )
{
    int x = 0;
    srand48(getpid());
    x = foo();
    fprintf(stdout, "%d", x);
}

func.c:
#include <stdio.h>
#include <stdlib.h>
#include "func.h"

int foo()
{
    int x = mrand48();
    fprintf(stdout, "%d\n", x);
    return x;

    /* note that we don't need to call srand48 again, just once at the top of main */
Random reminders

- The line:
  ```c
  #include <stdlib.h>
  ```
- Goes at the top of any .c file that contains functions that call either srand48() or mrand48()
- The line:
  ```c
  srand48(getpid());
  ```
- Goes *once* in main() after all variables have been declared
To the n'th degree

• Now for something completely different

• You can create an “array of int's” and simply access the “n'th” one:
  ```c
  int n = 2;
  int i[10];
  i[0] = 1;
  i[1] = 2;
  i[n] = 3;
  ```
Array

```c
int array[5];
array[3] = 42;
```
One-shot Array Setup

- Just like you can set an int at declaration time:
  ```
  int a = 5;
  ```

- You can also set an array immediately, too:
  ```
  int a[4] = { 1, 2, 3, 4 };
  ```

- Or using a loop...
Array

```c
int array[5];
array[0] = 1;
array[1] = 1;
array[2] = 1;
array[3] = 1;
array[4] = 1;
```

But that's the "repetitive" way, how about "simpler" using a loop"
2-D

• This will be necessary for your project... you can also have more than one dimension of array:
  
  ```c
  int matrix[10][10];
  int i = 0;
  int j = 9;
  matrix[i][j] = 999;
  ```

• How to set each value? Loops? Hmm...
int matrix[3][2];
matrix[1][1] = 42;

(orientation doesn't matter)
Loops of Stuff 1

• If you do 60 things in a loop:
  
  ```
  for( m=0; m<60; m=m+1 )
  {
    fprintf(stdout,"minute=%d\n",m);
  }
  ```

• And we want to do these 60 things... 24 times each...
Loops of Stuff 2

• Just do all the 60 things in a loop... a loop of loops:
  for( h=0; h<24; h=h+1 )
  {
      fprintf(stdout,"hour=%d\n",h);
      /*
       * the bunch of 60 stuff
       * you want to repeat 24 times
       */
  }
Loops of Stuff 3

• Just do all the 60 things in a loop... a loop of loops:
  for( h=0; h<24; h=h+1 )
  {
    fprintf(stdout,"hour=%d\n",h);
    for( m=0; m<60; m=m+1 )
    {
      fprintf(stdout,"minute=%d\n",m);
    }
  }
How many times?

/* 1 time */
for( h=0; h<24; h=h+1 )
{
    /* 24 times */
    fprintf(stdout,"hour=%d\n",h);
    for( m=0; m<60; m=m+1 )
    {
        /* 24 x 60 times */
        fprintf(stdout,"minute=%d\n",m);
    }
    /* 24 times */
}
/* 1 time */
Plucking out Functions

- Sometimes it makes sense to write code, and then pull it out into a function. Here's a simple example:

```c
#include <stdio.h>

int main( int argc, char** argv ) {
  int i = 0;
  while( i < 10 ) {
    fprintf( stdout, "\%d\n", i );
    i = i + 1;
  }
}
```
Plucking out Functions 2

#include <stdio.h>

int main(int argc, char** argv) {
    int i = 0;
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}

void func( void ) {
    int i = 0;
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}

Plucking out Functions 2

```c
#include <stdio.h>

int main(int argc, char** argv) {
    void func(void) {
        int i = 0;
        while(i < 10) {
            fprintf(stdout, "%d\n", i);
            i = i + 1;
        }
    }
    func();
}
```
```c
#include <stdio.h>

void func( void );

int main( int argc, char** argv ) {
    func();
}

void func( void ) {
    int i = 0;
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}
```
#include <stdio.h>

int main( int argc, char** argv ) {
    int i = 0;
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}

void func( int i ) {
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}
```c
#include <stdio.h>

int main( int argc, char** argv ) {
    func(10);
}

void func( int i ) {
    while( i < 10 ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}
```
#include <stdio.h>

void func( int times );

int main( int argc, char** argv ) {
    func( 10 );
}

void func( int times ) {
    int i = 0;
    while( i < times ) {
        fprintf( stdout, "%d\n", i );
        i = i + 1;
    }
}
Entering the Matrix

• To pass an array or matrix to a function, name it without any brackets:

```c
int array[5];
func(array);
...

int func( int array[5] )
{
    array[0] = 1; /*etc*/
}
```
Caveat

• Be aware: for reasons not totally explained yet, if you alter a matrix element in a function that has been passed in as a parameter, it changes it back in the function (possibly main) where it came from.

• That being said, you most certainly can do:
  ```c
  int foo( int array[5] );
  int main( ... ) {
      int x[5];
      foo( x );
  }
  ```
# Travelling arrays

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int array[5] = {1, 2, 3, 4, 5};
    void func(int x[5]) {
        x[2] = 42;
    }
    func(array);
    x[2] = 3 at this point
    x[2] = 42 now
}
```
From outside func

array

array-of-5-ints shaped hole

void (no output hole)
From outside func

array-of-5-ints shaped hole

void (no output hole)
From inside func

array

giving int

void (no output hole)
From inside func

void (no output hole)
From outside func

array of 5 ints shaped hole

void (no output hole)
From outside func array

array-of-5-ints shaped hole

void (no output hole)
Lab 0

- A reminder about turning repetitive things into loops:
- lab0.c:

```c
#include <stdio.h>
int main( int argc, char **argv )
{
    fprintf(stdout, "%d\n", 0);
    fprintf(stdout, "%d\n", 1);
    fprintf(stdout, "%d\n", 2);
    fprintf(stdout, "%d\n", 3);
    fprintf(stdout, "%d\n", 4);
    return 0;
}
```
Lab 0 - part 2

- A reminder about turning repetitive things into loops:

- lab0.c:

```c
#include <stdio.h>
int main( int argc, char **argv )
{
    int i = 0;
    fprintf(stdout,"%d\n",i); i=i+1;
    fprintf(stdout,"%d\n",i); i=i+1;
    fprintf(stdout,"%d\n",i); i=i+1;
    fprintf(stdout,"%d\n",i); i=i+1;
    fprintf(stdout,"%d\n",i); i=i+1;
    return 0;
}
```
Lab 0 - part 3

- A reminder about turning repetitive things into loops:

- lab0.c:

  ```c
  #include <stdio.h>
  int main( int argc, char **argv )
  {
      int i = 0;
      while( i < 5 )
      {
          fprintf(stdout,"%d\n",i);
          i=i+1;
      }
      return 0;
  }
  ```
Lab 1a

- Open up lab-6
- `git init` (and remember to add your files as you go)
- Write a program that makes an array of 10 integers, and fill them with the number 5
- Call it “lab1.c” (`git add lab1a.c`)
- Fill those integers with random values, one at a time, in a loop... (No static initializer “int x[10] = { ... };”)
- *After* filling all integers, print them all out, in a loop (so, two loops, one right after the other)
- `git commit -a -m 'lab1a milestone - done or whatever'`
- [Approx 6-8 lines]
Lab 1a Spreadsheet

array[0] = 5
array[1] = 5
array[2] = 5
array[3] = 5
array[4] = 5

print array[0]
print array[1]
print array[2]
print array[3]
print array[4]
Lab 1b

• Copy it to “lab1b.c” (git add lab1b.c)
• Write a program that makes an array of 10 integers, and fill them with random numbers in the range 0-100
• git commit -a -m 'lab1a milestone - done or whatever'
• [Approx 3 new lines]
Lab 1b Spreadsheet

array[0] = random 0
array[1] = random 1
array[2] = random 2
array[3] = random 3
array[4] = random 4

print array[0] 0
print array[1] 1
print array[2] 2
print array[3] 3
print array[4] 4
Lab 2

- Copy lab1.c to lab2.c (git add lab2.c)
- "Pluck out" a fill and print function - each should do the filling or printing loop inside the function
  - Pass the array to a function that does the filling
  - Then pass the array to a function that does the printing
- The prototypes for the functions go above “int main” but below “stdio.h”
- The declarations go way at the bottom
- git commit -a -m 'lab2 milestone - done or whatever'
- [Approx 2 lines of prototypes and 4-8 more lines of boilerplate just to move stuff into functions.]
Lab 2 Main

main.c:

```c
#include <stdio.h>
void fill( int array[10] );
void print(int array[10] );

int main( int argc, char **argv ) {
    int array[10];
    fill(array);
    print(array);
    return 0;
}

... function definitions ...
```
Lab 3

- Copy lab2.c to lab3.c (git add lab3.c)
- "Pluck out" the headers into separate files:
  - Move the prototypes to a lab3func.h file
  - Move the declarations to a lab3func.c file
- git commit -a -m 'lab3 milestone - done or whatever'
Lab 3 Main

main.c:

```c
#include <stdio.h>
...
move prototypes to functions.h ...
#include "functions.h"

int main( int argc, char **argv ) {
    int array[10];
    fill(array);
    print(array);
    return 0;
}
```

... move function definitions to functions.c ...
Lab 4

- Copy lab3.c/lab3func.c/lab3func.h → lab4.c/etc (git add …)
  - Change the array to a 5x5 2D array
  - The prototypes and declarations will have to change
  - In the print function, print them all out in a nice, orderly format, lined up in columns

*Hint 1: you don't need to print a newline with every fprintf*

*Hint 2:*

```c
    fprintf( stdout, "%5d", i );
    will always use 5 at spaces when printing an integer
```

* [ Approx 4-8 lines difference ]*
array[0][0] = random 0 0
array[0][1] = random 0 1
array[0][2] = random 0 2
array[0][3] = random 0 3
array[0][4] = random 0 4
array[1][0] = random 1 0
array[1][1] = random 1 1
array[1][2] = random 1 2
array[1][3] = random 1 3
array[1][4] = random 1 4
array[2][0] = random 2 0
Lab 5

- You may just want to edit lab4, rather than make 3 new files – this is pretty simple... just commit your previous changes first...
  - Make *two* arrays, fill both, print both
  - Make sure to reuse your existing functions - i.e. don't make any new functions

- [ 3 lines ]
Lab 6

- Copy lab5.c/etc to lab6.c/etc (and git add)
  - Make a third matrix
  - After filling in all the random values in array A and B, add every element, one by one, and put the result in array C
  - Do this in a function called add(a,b,c)
  - Print c afterwards (back in main, using print())
  - i.e. Cx,y = Ax,y+Bx,y for every x,y

- [ 3 lines ]
Lab 6 Spreadsheet

\[
\begin{align*}
  c[0][0] &= a[0][0] + b[0][0] & 0 & 0 \\
  c[0][1] &= a[0][1] + b[0][1] & 0 & 1 \\
  c[0][2] &= a[0][2] + b[0][2] & 0 & 2 \\
  c[0][3] &= a[0][3] + b[0][3] & 0 & 3 \\
  c[0][4] &= a[0][4] + b[0][4] & 0 & 4 \\
  c[1][0] &= a[1][0] + b[1][0] & 1 & 0 \\
  c[1][1] &= a[1][1] + b[1][1] & 1 & 1 \\
  c[1][2] &= a[1][2] + b[1][2] & 1 & 2 \\
  c[1][3] &= a[1][3] + b[1][3] & 1 & 3 \\
  c[2][0] &= a[2][0] + b[2][0] & 2 & 0
\end{align*}
\]
Homework 6 Spreadsheet

\[
c[0][0] = b[0][0] \times a[0][0] + b[0][1] \times a[1][0] + b[0][2] \times a[2][0] + b[0][3] \times a[3][0] + b[0][4] \times a[4][0]
\]

\[
c[0][1] = b[1][0] \times a[0][0] + b[1][1] \times a[1][0] + b[1][2] \times a[2][0] + b[1][3] \times a[3][0] + b[1][4] \times a[4][0]
\]

\[
c[0][2] = b[2][0] \times a[0][0] + b[2][1] \times a[1][0] + b[2][2] \times a[2][0] + b[2][3] \times a[3][0] + b[2][4] \times a[4][0]
\]

\[
\vdots = \vdots \times \vdots
\]

\[
c[1][0] = b[0][0] \times a[0][1] + b[1][0] \times a[1][1] + b[2][0] \times a[2][1] + b[3][0] \times a[3][1] + b[4][0] \times a[4][1]
\]

\[
c[i][j] = b[j][0] \times a[0][i] + b[j][1] \times a[1][i] + b[j][2] \times a[2][i] + b[j][3] \times a[3][i] + b[j][4] \times a[4][i]
\]

\[
\vdots = \vdots \times \vdots
\]

\[
c[i][j] = b[j][k] \times a[k][i]
\]