NO: 1 DATE:

Programming Fundamentals

Plan code first before coding. One example is to determine if 7 is prime.

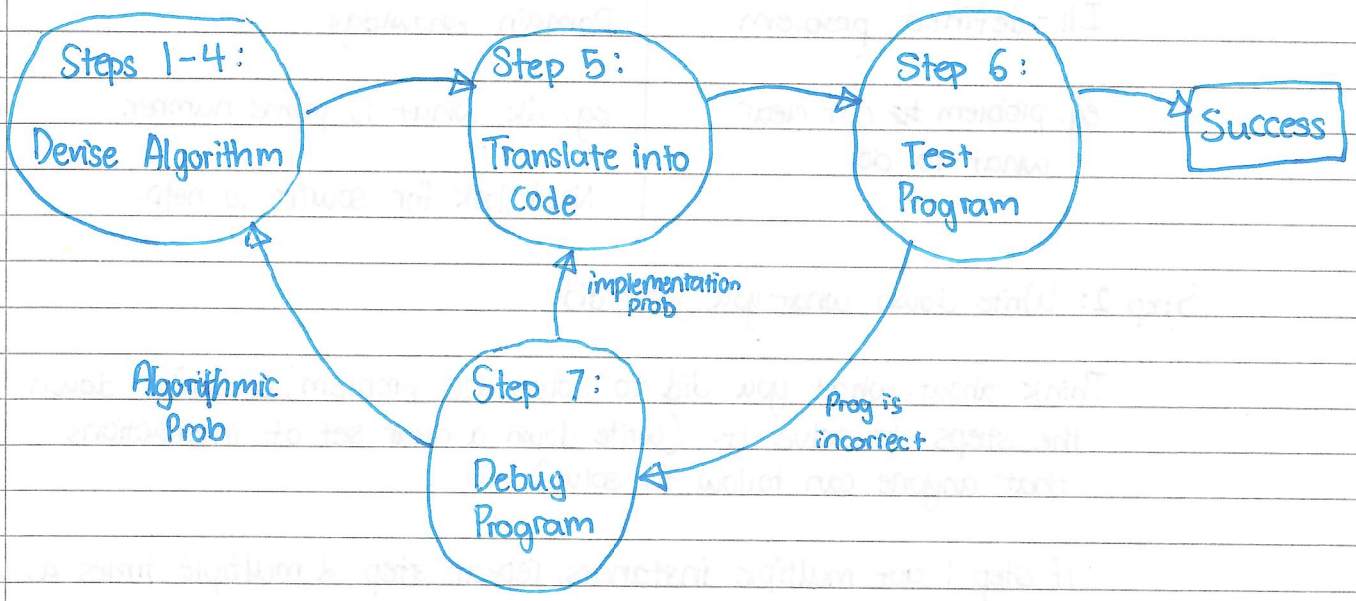
↓

Instead of programming to solve 7 is prime,
~~de~~ plan something to solve N is prime.
(solve general class of problems)

↓

Account all legal values of N

7 steps.



Algorithms.

An algo is a clear set of steps to solve any prob in a particular class.
Can have parameters or ~~a fix~~ none.

Eg. Given a non-negative integer N:
Make a variable called X, set it equal to (N+2)
Count from 0 to N (include both ends), & for each number (call it "i")
that you count:
Write down the value of (X * i)
Update x to be equal to (X + i * N)
When you finish counting, write down the value of X

N	#2
X	2+2=4,
i	#4*0=0,

First 4 Steps.

Step 1: Work an example yourself.

Try to design an algorithm to work at least 1 instance of the prob (pick specific values by hand)

for problems with yes or no ans, work out examples that give both ans

If you get stuck at this step, there are 2 reasons.

1.	2.
Ill-defined problem	Domain knowledge
eg. problem ~~is~~ not clear what to do	eg. dk what is prime number.
	Need look for sources to help.

Step 2: Write down what you just did.

Think about what you did to solve the problem, & write down the steps to solve it. (write down a clear set of instructions that anyone can follow to solve)

If step 1 got multiple instances, repeat step 2 multiple times as well.

think about what exactly you did to accomplish the problem. Implicit assumptions abt what to do / relying on common sense lead to imprecise or omitted steps.

Think thru all the details.
Eg. Computing X to the Y, might write steps for X=3 & Y=4

Multiply 3 by 3
You get 9
Multiply 3 by 9
You get 27
Multiply 3 by 27
You get 81

NO: 3 DATE:

Step 3: Generalise your steps

Generalise steps took from step 2 into algorithms.
Find a pattern that solves a whole class

Made up of 2 activities.

1. Take particular values (that were used) & replace them with mathamatical expressions of the parameters
   eg. from prev step 2 eg.

multiply x by 3
you get 9
multiply x by 9
you get 27
multiply x by 27
you get 81
81 is your ans.

→ computed 3^4, always multiply by 3 each step
generalise 3 with X. (Multiply x by 3)

2. Find repetition (repeated steps)
   Generalise how many times to do the steps. Sometimes steps repetitive, may need to adjust step to make them ~~extremety~~ exactly repetitive.
   eg. From prev 3^4 eg.

multiplication steps almost repetitive, multiply X by something but something changes.

Start with n = 3
n = multiply x by n
n = multiply x by n
n = multiply x by n
n is your ans

contemplate num of time steps repeat, don't jump to conclusion that it repeats X times (in this case just a coincidence.) In this case is Y-1 times.

Start with n = 3 or X (generalise)
Count up from 1 to y-1 (inclusive)
  n = multiply x by n
n is your ans.

Pseudo-code, is working to design an algorithm with no target language.

Step 4: Test your algorithm

After Step 3, we have an algorithm that we think is right. Test our algorithm with diff values from what we use to design. Compare if it is right, this is to ensure our steps are right bfr we proceed. (more test cases, more confident)

- 1 common mistake is misgeneralizing in Step 3. If that happend, will be detected in Step 4 when testing by hand.

↓

go back & rexamine generalizations in Step 3

↓

return to Step 1 & 2 to whatever exposed the problem (give you a concreate set of steps to regeneralise)

- Presence of cases that were not considered
  eg. X^Y example.

nvr consider when y=0 → when execute by hand, when x=2, y=0 → should 2^0=1 however get 2

when start with n=x=2, count up from 1 to 0-1=-1, gives back 2

→ To fix it, revisit Step 1 & 2 for failed case.

Better way

realise that if count no times, need ans of 1
should start @ n at 1 instead of X.
we need to count 1 more time (to y)
- to multiply by X 1 more time.

Start with n=1
count up from 1 to Y (inclusive), for each num count,
n = multiply x by n
n is your ans.

1 way

if: y=0 then
1 is ans
else:
start with n=x
:
:
:

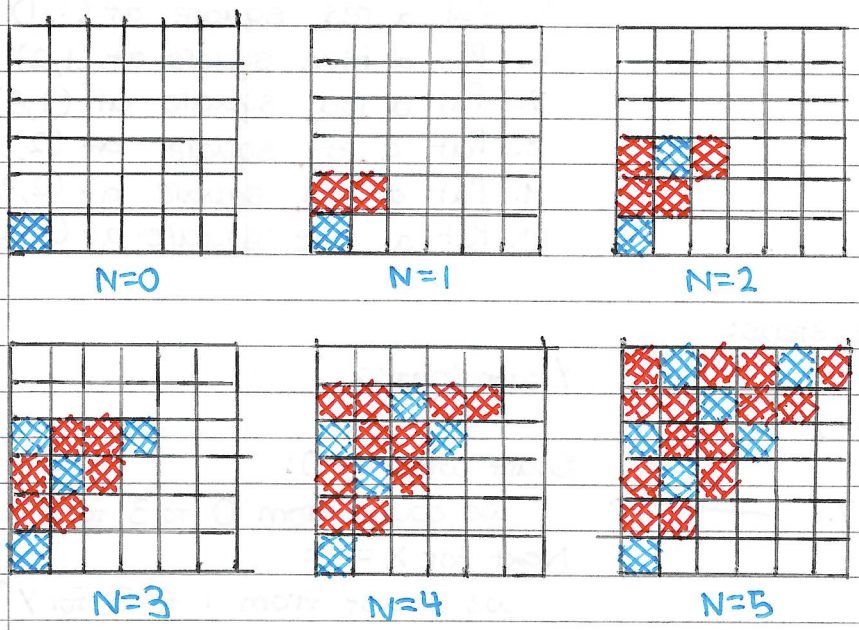
* Retest with all test cases all test cases alrdy used & new ones.
Test cases with diff ans, test corner-cases (cases that differ from general)

Steps 1-4 practice quiz.

If on step 3 & can't see a pattern, repeat steps 1 & 2 on diff examples.

A Pattern Of Squares

As 2nd example, look @ pattern of squares on a grid.



To devise this algorithm, work thru steps 1-4 (many correct algo, just how you approach)

There is always more than 1 correct ans. to any proggraming problem. can look completely diff from other solutions.

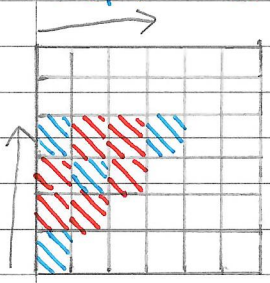
top down, left to right

```
Count down from N to 0 (inclusive), call each number you count "y" &
    Count down from 0 to y (inclusive), call each num you count "x" &
        if (x + y is a multiple of 3)
            then place a blue square at (x,y)
        otherwise place a red square at (x,y)
```

NO: 6 DATE:

Developing an algorithm of squares.

Step 1: Do an instance of the problem



N = 3

Step 2: Write down what you did

1. Put a blue square at (0,0)
2. Put a red square at (0,1)
3. Put a red square at (0,2)
4. Put a blue square at (0,3)
5. Put a red square at (1,1)
6. Put a blue square at (1,2)
7. Put a red square at (1,3)
8. Put a red square at (2,2)
9. Put a red square at (2,3)
10. Put a blue square at (3,3)

Step 3: Generalise the steps:

X coordinates

X = 0 → 1., 2., 3., 4.

X = 1 → 5., 6., 7.

X = 2 → 8., 9.

X = 3 → 10.

X: 0 to 3

Y coordinates

Start with X = 0:
we count from 0 to 3 for Y → 1-4

Next for X = 1:
we count from 1 to 3 for Y → 5-7

Next for X = 2:
we count from 2 to 3 for Y → 8-9

It looks like we count from X to 3

Ignoring colour:

Count from (0) to 3 (call it Y)
put a (color) square at ((0),Y) → 1-4

Count from (1) to 3 (call it Y)
put a (color) square at ((1),Y) → 5-7

Count from (2) to 3 (call it Y)
put a (color) square at ((2),Y) → 8-9

Count from (3) to 3 (call it y)
put a (color) square at ((3),Y) → 10

Count from 0 to 3 (call it x)
Count from x to 3 (call it y)
put a (color) square at (x,y)

generalise for a class

For any N
Count from 0 to N (call it X)
Count from x to N (call it y)
Put a (??) square at (X,Y)
* (do for N = 1 & generalise with N = 3)

↓

Still need to figure out colors,
go back to N=3.

extract blue:

* if can't spot pattern, try diff parameter, bigger in size.

Put a blue square at (0,0) 1. Total: 0

Put a blue square at (0,3) 4. Total: 3

Put a blue square at (~~3~~1,2) 6. Total: 3

Put a blue square at (3,3) 10. Total: 6

↓

An algorithm for any N:

Count from 0 to N (call it x)
  Count from x to N (call it y)
    if (x+y) is a multiple of 3,
      Put a blue square at (x,y)
    Otherwise,
      Put a red square at (x,y)

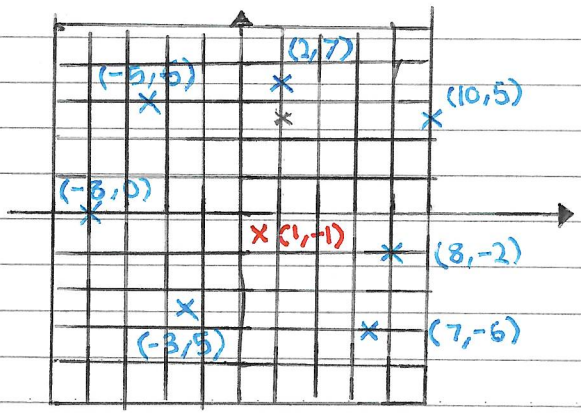
Step 4: Test the algorithm with a diff parameter that was used.
Eg. N=2 or N=6

Example algorithms.

Draw a rectangle: Count from y to (y+height) (exclusive), call each num i
Draw a blue horizontal line of length width start (x,i)

Closest Point

distance = $\sqrt{\Delta x^2 + \Delta y^2}$



NO: 9 DATE:

Algorithm Practice (Practice Quiz)

Q1. N = 2 output: 6, 8, 10, 12, 14, 16 (6 num) 2+4 = 6 = 2+2+2=6

6 = 2+4 = 2+2+2 (+2) = 2+4+2(0) = 2+(2x2)
8 = 2+6 = 2+2+2+2 = 2+4+2(1) = 2+(2x2)+2(1)
10 = 2+8 = 2+2+2+2+2 = 2+4+2(2) = 2+(2x2)+2(2)
12 = 2+10 = 2+2+2+2+2+2 = 2+4+2(3) = 2+(2x2)+2(3)
14 = 2+12 = 2+2+2+2+2+2+2 = 2+4+2(4) = 2+(2x2)+2(4)
16 = 2+12 = 2+2+2+2+2+2+2+2 = 2+4+2(5) = 2+(2x2)+2(5)

N=4 output: 12, 14, 16, 18, 20, 22, 24, 26, 28, 30 (10 num)
4+6=10 = 4+4+2=10

12 = 4+8 = 4+(2x4)+2(0)
14 = 4+8+2 = 4+(2x4)+2(1)
16 = 4+8+2+2 = 4+(2x4)+2(2)
18 = 4+8+2+2+2 = 4+(2x4)+2(3)
20 = 4+8+2+2+2+2 = 4+(2x4)+2(4)
22 = 4+8+2+2+2+2+2 = 4+(2x4)+2(5)
24 = 4+8+2+2+2+2+2+2 = 4+(2x4)+2(6)
26 = 4+8+2+2+2+2+2+2+2 = 4+(2x4)+2(7)
28 = 4+8+2+2+2+2+2+2+2+2 = 4+(2x4)+2(8)
30 = 4+8+2+2+2+2+2+2+2+2+2 = 4+(2x4)+2(9)

Deciding num of num: [2+2+2=6] & [4+4+2=10]
= [N+N+2]
= 2N+2

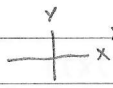
Deciding num in sequence: ~~2+(2x2)+2(i)~~ = ~~4+(2x4)+2(i)~~
= N+(2xN)+2i
= 3N + 2i

Algo: For 0 to 2N+2 (exclude), for each num count "i",
output 3N+2i

```
#include <iostream>
using namespace std;

int main ()
{   int N;
    cout << "Type in the number of N: ";
    cin >> N;
    for (int i=0 ; i< ((2*N)+2); i++)
        Cout <<(3*N)+(2*i)<<endl;
}
```

NO: 10 DATE:

Q2. N=3 Output



x=0: Plot red block (0,0); Plot red block (0,1) — y=1
x=1: plot red block (1,1); plot red block (1,2) — y=2
x=2: plot red block (2,2); plot red block (2,3) — y=3
x=3: plot red block (3,3); plot red block (3,4) — y=4
x=5: plot blue block (5,3); plot blue block (5,5)

count from 0 to 3 (including), count is "i"

for x=0,
count from 0 to 1 for y
for x=1,
count from 1 to 2 for y
for x=2,
count from 2 to 3 for y
for x=3,
count from 3 to 4 for y
for x=5,
count from 3 to 5 for y

N=2 Output

x=0: plot red block (0,0); plot red block (0,1)
x=1: plot red block(1,1); plot red block(1,2)
x=2: plot red block(2,2); plot red block(2,3)
x=3: plot green block(3,3)

For any N
count from 0 to N, (call it x)
count from N to N+1, (call it y)
plot a red block (x,y)
if x is even,
plot a green block (x+1, x+1)
else,
plot a blue block (x+2, x)
plot a blue block (x+2, x+2)

NO: 11 DATE:

Q3. N=2 output: 0, 2, 2, 3 (4 num) = 2×2 = N×2

N=4 output: 0, 2, 4, 6, 4, 5, 6, 7 (8 num) = 4×2 = N×2

N=5 output: (10 num) = 5×2

count from 0 to 4×2 (excluding)
if i is less than 4
output 2×i
else,
output i

↓

count from 0 to 2N (excluding)
if i is less than N
output 2×i
else,
output i

N=4	N=5
0. 0 = 2×0	0. 0 = 2×0
1. 2 = 2×1	1. 2 = 2×1
2. 4 = 2×2	2. 4 = 2×2
3. 6 = 2×3	3. 6 = 2×3
4. 4	4. 8 = 2×4
5. 5	5. 5
6. 6	6. 6
7. 7	7. 7
	8. 8
	9. 9

0, 2, 4, 6, 8, 10, 6, 7, 8, 9, 10, 11 → N=6

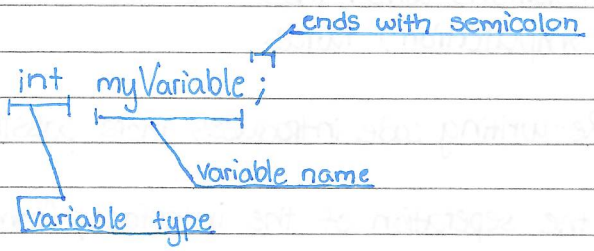
```
#include <iostream>
using namespace std;

int main()
{ int N;
  cout<<"Type in the number of N:";
  cin >> N;
  for (int i=0 ; i<(2*N); i++)
    if (i<N){
      cout<< 2*i <<endl;
    }
    else { cout << i <<endl;
    }
}
```

NO: 12 DATE:

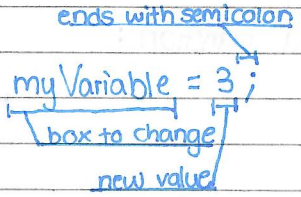
NO: 15 DATE:

Week 2

Variable syntax: `int myVariable;`



Assigning a variable: lvalue = rvalue



Declaring & Assigning a Variable:

```
int x;
int y;
```

x	?
y	?

uninitialised represented as ?

↓

bad as can lead to bugs, hard to fix

```
int x;      → initialisation
x = 4;      → assignment
int y = 6;  → initialisation + assignment
```

Expression with normal operators: +, -, *, /, % has mathematical rules of precedence * & / bfr others

% (modulus) evaluates to the remainder when ÷

eg. 19 % 5 = 4

↳ because 19/5 = 15/5 remainder 4

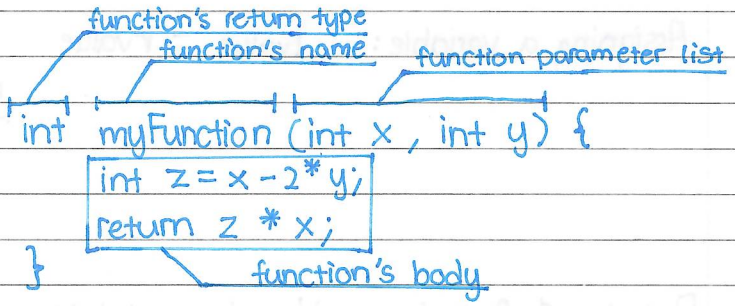
floor division uses integer (int) as it can only hold whole numbers. → eg. 5/2 = 2

Functions: - Situations when there's a need to execute same code (or computation) twice.

- Re-writing code introduces more possibilities for errors (bugs)

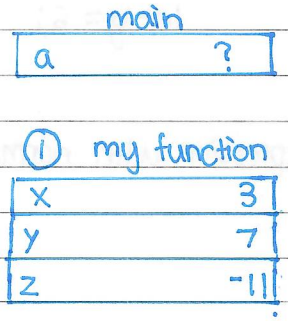
Abstraction is the seperation of the interface, from its implementations.

Anatomy of a Function:



Evaluating a Function: when starting to read code with func calls, group tog the variables belonging to one particular func into a larger box, labeled with the func name (called a frame or stack frame)

```
int myFuction (int x, int y) {
  int z = x - 2 * y;
  return z * x;
}
int main (void) {
  int a;
  a = myFuction (3, 7); ①
  return 0;
}
```



- main is a special function, the prog starts at main
- functions must be evaluated to determine return result.

1. Draw frame for func being called.
2. Initialise the parameters by evaluating corresponding expressions in func call, copy results into parameter's box in func frame.
3. Mark location of func call & note that at func frame.
4. Move execution arrow bfr first line of code in func
5. Evaluate lines of code inside func
6. When reach a return statement, evaluate its arg to value & note down.
7. Use the value prev as the value of func call in exp it appears.

Shorthand

Shorthand	Meaning
x += y;	x = x + y;
x -= y;	x = x - y;
x *= y;	x = x * y;
x /= y;	x = x / y;
x++;	x = x + 1;
++x;	x = x + 1;
x--;	x = x - 1;
--x;	x = x - 1;

While Loops

syntax:

```
while (x<n) {
    y = y * x;
    x++;
}
```

Conditional Expression

loop body.

quiz (while loop)

x	2, 3, 4
y	6, 5, 4
output	4, 2

Do / While Loops

syntax:

```
do {
    y = y * x;
    x++;
} while (x<n);
```

loop body

Conditional Expression

For Loops

syntax:

```
for (int i = 0; i<n; i++) {
    y = y * i;
}
```

Initialisation Statement

conditional expression

Increment Statement

loop body

vs

Syntax (equivalent while loop)

```
{ int i = 0;
  while (i < n) {
    y = y * i;
    i++;
  }}
```

Break, (exit loop completely)

syntax: break;

- jumps out of the innermost enclosing loop.
- if break statement is inside multiple nested loops, it then only exits only the most immediately enclosing one.
- if break occurs & not inside a loop/switch, its a error

Continue, (jump back to the top of the loop.)

syntax: continue;

- the continue statement jumps to the top of the innermost enclosing loop
- in for loop, (if have continue) "increment statement" is written bfr (}) close curly brace & also bfr any continue statement
- error if not in loop.
- in for loop, the "increment" in the loop is executed immediately before the jump.

Loops Quiz.

Q1.

n : 5

ans	0	-1	1	4	8
i	1	2	3	4	5

Q2.

x: 1 1 0

n: 3

i: 0 1

Scope: The scope of a variable is the region of code in which its visable

Within a variable's scope, can refer directly
Outside it, cannot refer to it directly.

Most variables used will be local variables (ones declared in func) & func parameters.

In C, the scope of a local variable begins with declaration and ends at the closing curly-brace (}).

```
int myfunction (int x, int y) {
    int z = x + y;
    int n = 0;
    int i = 0;
    while (i < z) {
        int q = i * x - y;
        printf ("%d\n", q);
        n += q;
        i++;
    }
    return n;
}
```

scope of x & y

scope of var n

scope of var q

Printing

printf function; takes a string specifying what to print,
↳ requires format specifiers, which start with (%) sign.

printf syntax:
```
int x = 3;
int y = 4;
printf("x + y = %d ", x+y);
```

Conditional Statements

```
expr1 == expr 2
expr 1 != expr 2
expr 1 <= expr 2
expr 1  > expr 2
expr 1 >= expr 2   (NOT)
```

```
!expr
expr1 && expr 2   (AND)
expr1 || expr 2   (or)
```

If / Else

Conditional Expression

```
syntax:  if (x == 3) {
           y = z + 1;        "then" clause
         }
         else {
           z = x - 2;
           x = x + 1;        "else" clause
         }
```

Switch Case

execution continues as normal until encounters the keyword break. reaching another case label does not end another case. It's called "falling through" into the next case.

Selection expression

```
syntax: switch (x-y) {
          case 0:
            y= 7;          case labels
            break;
          case 1:
            y= 9;          fall through
          case 2:
            z = 42;
            break;
          default :        default case
            n=3;
            break; }
```

NO: 13 DATE:

Algorithms Quiz

Q3. N = 3 Outputs:

X	Y	C
0	3	red
0	4	green
1	2	red
1	3	blue
2	1	red
2	2	green
3	0	red

X, Y — ~~for~~ — (N+1)-X

plot red square @ (0,3) } count from 3 to 4
plot green square @ (0,4) }

plot red square @ (1,2) } count from 2 to 3
plot blue square @ (1,3) }

plot red square @ (2,1) } count from 1 to 2
plot green square @ (2,2) }

plot red square @ (3,0) } count from 0 to 1
plot blue square @ (3,1) }

plot green square @ (4,0)

count from 0 to 4, call each count "i" (Y)

count from (N+1)-X to (N+1)-X
= count from N-X to N-X+1

count from 0 to N+1, call each iteration "i"

count from 0 to N+1 (excluding), call it "i" (X)
count from N-i to N-i+1, call it (Y)

X	Y	C	
0	3	red	(one odd / one even)
0	4	green	(both even)
1	2	red	(one odd / one even)
1	3	blue	(both odd)
2	1	red	(one odd / one even)
2	2	green	(both even)
3	0	red	(one odd / one even)
3	1	blue	(both odd)
4	0	green	(both even)

```
if (X or Y is odd )and (x or y is even)
    plot red
if (x and Y is odd or 0)
    plot blue
else
   plot green
```

NO: 14 DATE:

count from 0 to N+1 (excluding), call it X
  count from N-X to N-X+1, call it Y
    if (X or Y is odd) and (X or Y is even or 0)
      plot red at (X,Y)
    if (X and Y is odd)
      plot blue at (X,Y)
    else
      plot green at (X,Y)
  plot green at (N+1, 0)

Q3. N = 3 output -9, -8, -5, 0, 7, 16, 27, 40, 55 (N×3)

+1 +3 +5 +7 +9 +11 +13 +15

+2 +2 +2 +2 +2 +2 +2

0 -9 = -(3×3)
1 -8 = -(3×3)+1 = -(3×3)+
2 -5 = -(3×3)+4 = -(3×3)+(2×2)
3 0 = -(3×3)+9 = -(3×3)+(3×3)
4 7 = -(3×3)+16 = -(3×3)+(4×4)
5 16 = -(3×3)+25
6 27 = -(3×3)+36

for 0 to 3N-1, count i as iteration
output (-(N²))+(i²)

Week 2 Graded Quiz

Q7. a: 3 4 5
b: 6 5 4
~~i~~ i: 0 1
output: a is 3 , b is 5 , a * i + b = 5

Q8. x: 2
a: 27

someFunction (x 2, y 3)
a: 5 7 9
x: 2
y: 3
i: 0 1 2

output: In the loop with i = 0 , a = 5
In the loop with i = 1 , a = 7
a = 27
x = 2

Week 3

Data types: Char : Store characters & letters (1 byte) only 2^[8] = 256 values

Int : whole numbers 2^[32] = 0 - 4,294,967,295 or half if split

Float: stores fractional numbers, single precision (floating point num)

Double: stores fractional numbers, double precision

Printing Redux

Format specifiers allows us to print variables of a variety of types. (via the function printf) syntax: printf (" .... (format specifier)/n"

format specifier	will be printed as...
%c	single character
%d	decimal integer
%u	unsigned decimal num
%o	octal number
%x	unsigned hexadecimal num
%f	decimal floating point
%e	scientific notation
%g	picks the shorter of %e or %f
%s	string prints chars until '\0'
%%	prints the % character

decimal formatting	will be printed as
%f	default : shows 6 decimal places
%.nf	shows n decimal places
%mf	prints with minimum width m
%m.nf	n decimal places & minimum with m

escape sequence	will be printed as
\n	newline
\t	tab
\\	prints a backslash.

Expressions Have Types

Determined by the types of the sub-expressions that make them up.

Type of function is declared in its return type.

eg. int f (int x, int y)
int g (double d , char c)

then the expression has type int.
Can see this from f(3,4) has type int as does g(42.6 , 'a'). As we dissussed, adding two int results in an it.

What happens when you have a binary operator & its operands have diff types?

↳ two operands must be converted to same type. (Most operations ~~must~~ can only be performed on the same type)

eg. int a
double b
(type) c = a+b

} compiler chooses which operand to convert based on what gives best answer → in this situation, int will be converted to double. ↓ to avoid losing fractional part of answer.

Some ocasions, you or the compiler ~~is~~ have to temporary treat the variable as if were another type. When a programmer does it, it is called casting & when the compiler does it, it is called type conversion or type promotion.

Casting (only for C not C++)

explicitly request conversion from one type to another

eg.
```
int main (void) {
    int nHrs = 40;
    int nDays = 7;

    float avg = nHrs / (float) nDays;
    printf ("%d hours in %d days\n", nHrs, nDays);
    printf (" work %.1f hours per day! \n", avg);

}
```

Once one is converted to a real type, the compiler is forced to automatically convert the other.

Prefer a / (double)b over (double)a / b as not required to remember operator precedence.

Overflow & Underflow

type	size (typical)	intepretation	example
char	1 byte (8 bits)	one ASCII character	'f'
int	4 byte (32 bits)	binary integer	42
float	4 byte (32 bits)	floating point num	3.141592
double	8 byte (64 bits)	floating point num	3.141592...

each set size creates a limit on the smallest & largest possible num that can be stored, in a type.

eg. short is typically 16 bits = 2^16 possible values. if split bet positive & negative numbers, largest possible is 32767 (0111 1111 1111 1111).

if add 1 to this num, becomes 1000 0000 000 000 which is -327678

Overflow

Expression Have Types (Practice Quiz)

Q1. Casting a value should be done rarely & the original value is left unmodified.

Q3. char c = 250; // 1111 1010 in binary
c += 8; // 8 is 1000 in binary, 258 in binary is 1000 00010

10 overflows so = 2

Structs (form of abstraction)

Allows to bundle multiple variables into a single entity. There are multiple ways to declare, define & use structs.

Options 1-4	Creating new tags (1-3) & types (2-4)	Instating the variable

1. Define a tag (rect_t) Tag can be used with the word struct as a prefix.	struct rect_t {
	int left;
	int bottom;
	int right;
	int top;
	};
		int main ()
		{
		struct react_t myRect;
		myRect.left = 1;
		...

2. Define a tag (rect_tag) & then define its type alias (rect_t).
 	
Struct declaration and typedef can occur in either order. Tag can be used on its own with struct prefix.
	struct rect_tag {
	int left;
	int bottom;
	int right;
	int top;
	};
	typedef struct rect_tag rect_t;
		int main ()
		{
		rect_t myRect;
		myRect.left = 1;
		...

3. Abbreviation from 2. Declaration & defination occur in the same statement.	typedef struct rect_tag {
	int left;
	int bottom;
	int right;
	int top;
	} rect_t;
		int main ()
		{
		rect_t myRect;
		myRect.left = 1;
		...

4. Type defination with no tag declaration. Downside: struct cannot refer to itself.	typedef struct {
	int left;
	int bottom;
	int right;
	int top;
	} rect_t;
		int main ()
		{
		rect_t myRect;
		myRect.left = 1;
		...

type struct rect_t

type rect_t

Typedef (creates a new data type that is ~~es~~ explicitly of the type struct.)

To declare, pass or use the new struct.

Typedefs can help you make your code more readable, by naming a type by meaning, use.

Continuation of Typedef

```
typedef struct rect_tag rect_t;
```

typedef : new name for existing type.

tells that we have → new name for existing type

new name for existing type.

```
typedef unsigned int rgb_t
```
→ another use of typedef

Enumerated Types (Makes it easier to read, write & modify code)

named constant that can increase reliability & correctness of code.
Most useful when you have a type of data with a set of values that you would like to label by their conceptual name. example used

```
enum threat_level_t {      // C creates a correspondence between words &
  LOW,                     // 0          integer value
  GUARDED,                 // 1
  ELEVATED,                // 2
  HIGH,                    // 3
  SEVERE                   // 4
};

void printThreat(enum threat_level_t threat) {/* omitted */}
void printShoes(enum threat_level_t myThreat) {/* omitted */}

int main(void) {
  enum threat_level_t myThreat = HIGH;

  printf("Current threat level is: \n");
  printThreat(myThreat);
  printShoes(myThreat);
  return 0;
}
```

```
void printThreat(enum threat_level_t
                 threat) {
  switch (threat) {
    case LOW:
      printf("Green/Low.\n");
      break;
    case GUARDED:
      printf("Blue/Guarded.\n");
      break;
    case ELEVATED:
      printf("Yellow/Elevated.\n");
      break;
    case HIGH:
```

Complex, Custom Data Types Quiz

Q2. What does typedef do? : Make another name for an existing type.

Q3. For the following struct defination; which of the following correctly declares a struct?

```
struct point_tag {
    double x;
    double y;
};
typedef struct point_tag point_t;
```

Ans: point_t myPoint;
struct point_tag myPoint;

Quiz Types.

Writing a Sorting Algorithm

Q1. Write an algorithm that sorts the data from least to greatest.

N=3 inputs: 0.98, 1.92, 0.53
index: 0, 1, 2
compare 0.98 & 1.92, 0.98 > 1.92? ✗
stay the same
compare 1.92 & 0.53, 1.92 > 0.53? ✓
switch the positions of 1.92 & 0.53

N=6 inputs: 3.18, 1.36, -0.12, 0.53, 1.92, 0.98
index: 0, 1, 2, 3, 4, 5

compare val of index 0 & 1
3.18 > 1.36 ✓
move 3.18 to index 1
move 1.36 to index 0
compare val of index 1 & 2
3.18 > -0.12 ✓
move 3.18 to index 2
move -0.12 to index 1
compare val of index 2 & 3
3.18 > 0.53 ✓
move 3.18 to index 3
move 0.53 to index 2
compare val of index 0 & 1
1.36 > -0.12 ✓
move 1.36 to index 1
move -0.12 to index 0
compare val of index 1 & 2
1.36 > 0.53 ✓
move 1.36 to index 2
move 0.53 to index 1
compare val of index 0 & 1
-0.12 > 0.53 ✗
don't move.

count from 0 to 2, (call it a)
count from 1 to 3, (call it b)
if (index[a] > index[b])
tempVal = index[b]
index[b] = index[a]
index[a] = tempVal
else
continue.

simplified

Domain Knowledge

count from 0 to N-2, (call it a)
count from a+1 to N-1, (call it b)
~~if index[a] is more than index[b]~~
~~remember value of index[b]~~
if value at a is bigger than value at b,
switch places.
else.
continue.