Introduction to Programming

Chapter 4

Arrays

Basic concepts

Your first data structure

A data structure is an arrangement of data that enables efficient processing by a program.

An array is an *indexed* sequence of values of the same type.

Examples.

- *52 playing cards in a deck.*
- 100 thousand students in an online class.
- 1 billion pixels in a digital image.
- 4 billion nucleotides in a DNA strand.
- 73 billion Google queries per year.
- 86 billion neurons in the brain.
- 50 trillion cells in the human body.
- 6.02 × 1023 particles in a mole.

Index	value
0	2♥
1	6♠
2	A♦
3	A♥
…	
49	3♣
50	K♣
51	4♠

Main purpose. Facilitate storage and manipulation of data.

Processing many values of the same type

10 values, without arrays

```
double a0 = 0.0;
double a1 = 0.0;
double a2 = 0.0;
double a3 = 0.0;
double a4 = 0.0;
double a5 = 0.0;
double a6 = 0.0;
double a7 = 0.0;
double a8 = 0.0;
double a9 = 0.0;
...
a4 = 3.0;
...
a8 = 8.0;
...
double x = a4 + a8;
```

tedious and error-prone code

10 values, with an array

```
double a[10] = {};

...
a[4] = 3.0;
...
a[8] = 8.0;
...
double x = a[4] + a[8];
```

an easy alternative

1 million values*, with an array

```
double a[1’000’000] = {};

...
a[234567] = 3.0;
...
a[876543] = 8.0;
...
double x = a[234567] + a[876543];
```

scales to handle huge amounts of data

*May need to increase “stack” size

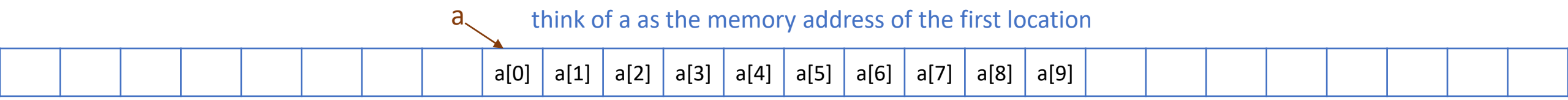
```
g++ -Wl,--stack,10000000 main.cpp
```

Memory representation of an array

An array is an indexed sequence of values of the same type.

A computer's memory is *also* an indexed sequence of memory locations.

- Each primitive type value occupies a fixed number of locations.
- *Array values are stored in contiguous locations.*



Critical concepts

- Indices start at 0.
- Given `i`, the operation of accessing the value `a[i]` is extremely efficient.

C++ language support for arrays

Basic Support

operation	typical code
Declare an array	`int a[1000];`
Declare and initialize an array	`int a[4] = {10, 20, 30, 40};`
Refer to an array entry by index	`a[i] = b[j] + c[k];`
Iterate over all entries one-by-one (range-based for loop)	`for(int x : a)` `cout << x;`

Static Arrays

Size of array must be known at compile-time

```
int n; cin >> n;
int a[n]; // error
```

Don’t be surprised: some compilers allow this

```
const int n = 1000;
int a[n]; // fine, n is constant
```

C++ arrays: Initialization options

operation	typical code
Only declaration, no initialization	`int a[1000];`
Declare and initialize an array	`int a[4] {10, 20, 30, 40};`
Size can be omitted when given initialization list	`int a[] {10, 20, 30, 40};`
Initialize to default value (0 for numeric data types)	`int a[1000] {};`
Remaining entries initialized to default value	`int a[1000] {10, 20, 30, 40};`

Programming with arrays: typical examples

Print array values, space separated

```
for(int i=0; i<N; i++)
    cout << a[i] << " ";
```

N is length/size of arrays in all this code.

Copy to another array

```
int b[N];
for(int i=0; i<N; i++)
    b[i] = a[i];
```

Check if two arrays of same size are equal

```
bool eq = true;
for(int i=0; i<N; i++)
    if(a[i]!=b[i]) eq = false;
```

Programming with arrays: more examples

Create an array with N random values

```
int a[N];
for(int i=0; i<N; i++)
    a[i] = rand();
```

N is length/size of arrays in all this code.

Compute the average of array values

```
double sum = 0;
for(int i=0; i<N; i++)
    sum += a[i];
double avg = sum / N;
```

Find the maximum of array values

```
int max = a[0];
for(int i=1; i<N; i++)
    if(a[i] > max) max = a[i];
```

Programming with arrays: typical bugs

Array index out of bounds

```
int a[10];
for(int i=1; i<=10; i++)
    a[i] = rand();
```

No `a[10]` (and `a[0]` unused)

entries need to be assigned or compared individually

Copy array variable

```
int a[5] = {10, 20, 30, 40. 50};
int b[5];    // Syntax error
b = a;
```

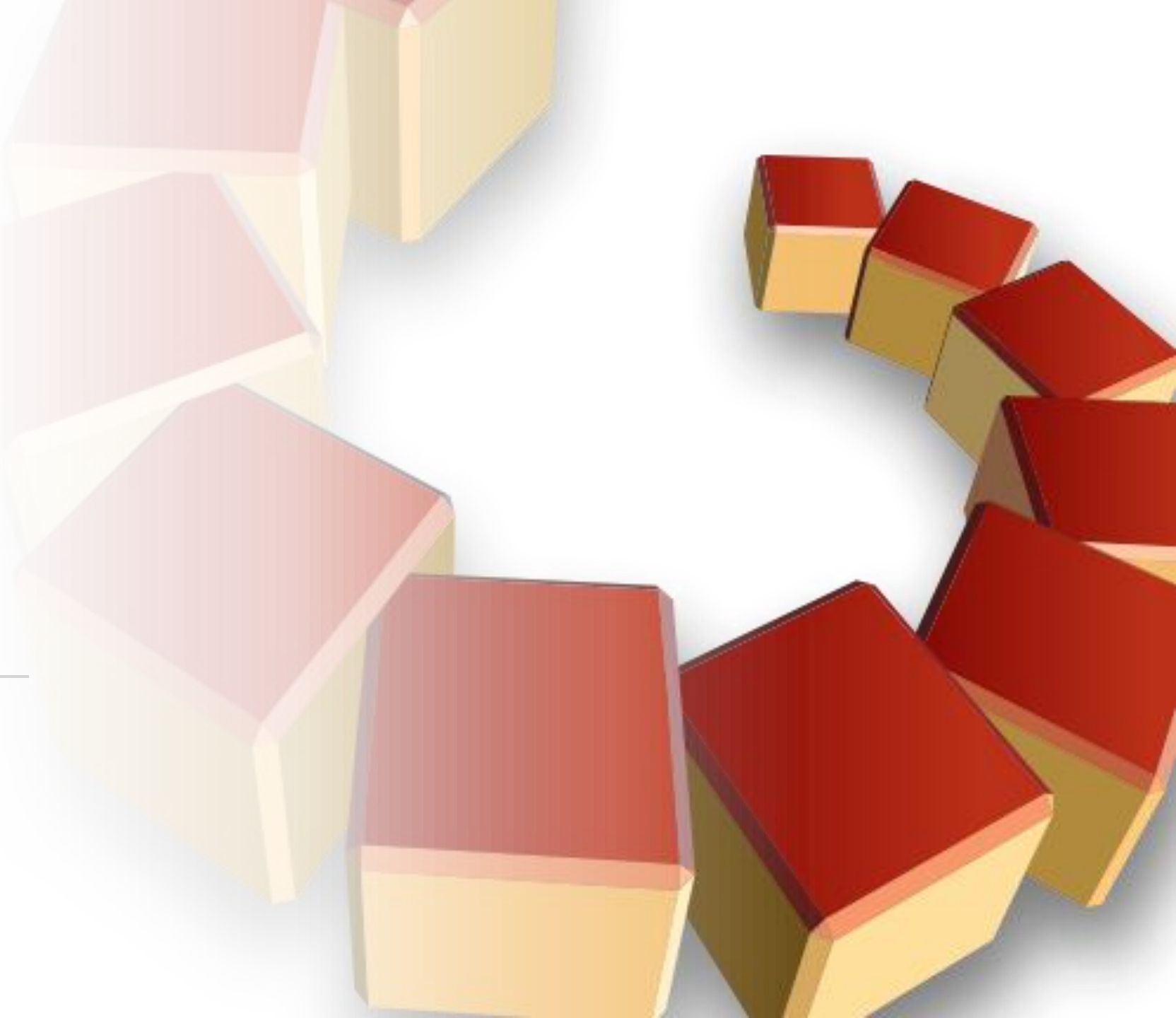
copy entries instead

Compare array variables

```
int a[5] = {10, 20, 30, 40, 50};
int b[5] = {10, 20, 30, 40, 50};
if(a==b) cout "Equal";    //No error!
else     cout "Not Equal";
```

need to compare corresponding elements

Examples of array-processing code



Example of array use: create a deck of cards

Define three arrays
- Ranks.
- Suits.
- Full deck.

```
string rank[] = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
```

```
string suit[] = {"C", "D", "S", "H"}; //Club, Diamond, Spade, Heart
```

```
string deck[52];
```

Use nested for loops to put all the cards in the deck.

```
for (int j = 0; j < 4; j++)
    for (int i = 0; i < 13; i++)
        deck[i + 13*j] = rank[i] + suit[j];
```

better style to use `std::size(rank)` and `std::size(suit)`
clearer in lecture to use 4 and 13

j	0	1	2	3
suit	C	D	S	H

i	0	1	2	3	4	5	6	7	8	9	10	11	12
rank	2	3	4	5	6	7	8	9	10	J	Q	K	A

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	…
deck	2C	3C	4C	5C	6C	7C	8C	9C	10C	JC	QC	KC	AC	2D	3D	4D	5D	6D	7D	8D	9D	…

Example of array use: create a deck of cards

```
#include<iostream>
#include<string>

int main() {
    std::string rank[] {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
    std::string suit[] {"C", "D", "S", "H"}; // Club, Diamond, Spade, Heart
    std::string deck[52];

    for (int j = 0; j < 4; j++)
        for (int i = 0; i < 13; i++)
            deck[i + 13*j] = rank[i] + suit[j];

    for(int i=0; i<52; i++)
        std::cout << deck[i] << " ";
    std::cout << std::endl;
}
```

```
➢ a
2C 3C 4C 5C 6C 7C 8C 9C 10C JC QC KC AC
2D 3D 4D 5D 6D 7D 8D 9D 10D JD QD KD AD
2S 3S 4S 5S 6S 7S 8S 9S 10S JS QS KS AS
2H 3H 4H 5H 6H 7H 8H 9H 10H JH QH KH AH
```

Pop quiz 1 on arrays

Q. What happens if the order of the for loops in Deck is switched?

```
for (int j = 0; j < 4; j++)
    for (int i = 0; i < 13; i++)
        deck[i + 13*j] = rank[i] + suit[j];
```

```
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 4; j++)
        deck[i + 13*j] = rank[i] + suit[j];
```

Pop quiz 1 on arrays

Q. What happens if the order of the for loops in Deck is switched?

```
for (int j = 0; j < 4; j++)
    for (int i = 0; i < 13; i++)
        deck[i + 13*j] = rank[i] + suit[j];
```

```
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 4; j++)
        deck[i + 13*j] = rank[i] + suit[j];
```

A. The array is filled in a different order, but the output is the same.

j	0	1	2	3
suit	C	D	S	H

i	0	1	2	3	4	5	6	7	8	9	10	11	12
rank	2	3	4	5	6	7	8	9	10	J	Q	K	A

	0	1	2	…	12	13	14	15	…	25	26	27	28	…	38	39	40	41	…	49	50	51
deck	2C	3C	4C	…	AC	2D	3D	4D	…	AD	2S	3S	4S	…	AS	2H	3H	4H	…	QH	KH	AH

Pop quiz 2 on arrays

Q. Change Deck to put the cards in rank order in the array.

- a

```
2C 2D 2S 2H 3C 3D 3S 3H 4C 4D 4S 4H 5C 5D 5S 5H 6C 6D 6S 6H 7C 7D 7S 7H 8C 8D 8S
8H 9C 9D 9S 9H 10C 10D 10S 10H JC JD JS JH QC QD QS QH KC KD KS KH AC AD AS AH
```

Pop quiz 2 on arrays

Q. Change Deck to put the cards in rank order in the array.

```
➢ a
2C 2D 2S 2H 3C 3D 3S 3H 4C 4D 4S 4H 5C 5D 5S 5H 6C 6D 6S 6H 7C 7D 7S 7H 8C 8D 8S
8H 9C 9D 9S 9H 10C 10D 10S 10H JC JD JS JH QC QD QS QH KC KD KS KH AC AD AS AH
```

A.

```
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 4; j++)
        deck[i*4 + j] = rank[i] + suit[j];
```

j

	0	1	2	3
suit	C	D	S	H

i

	0	1	2	3	4	5	6	7	8	9	10	11	12
rank	2	3	4	5	6	7	8	9	10	J	Q	K	A

	0	1	2	3	4	5	6	7	8	9	10	11	12	…
deck	2C	2D	2S	2H	3C	3D	3S	3H	4C	4D	4S	4H	5C	…


Take a card!
Any card!
That's my
credit card.
Abra kadabra.

Array application: take a card, any card

Problem: Print a random sequence of N cards.

Algorithm

Take N from input and do the following N times

- Calculate a random index r between 0 and 51.
- Print deck[r].

Implementation: Add this code instead of printing deck

```
for (int i = 0; i < N; i++)
     std::cout << deck[rand()%52] << " ";
```

another way to get equal probability of each card →

```
for(int i=0; i<N; i++) {
    double u = rand()/((double)RAND_MAX+1);
    // u in [0,1)
    std::cout << deck[(int)(u*52)] << " ";
}
```

Note: Same method is effective for printing a random sequence from any data collection.

Array application: random sequence of cards

```
#include<iostream>
#include<string>

int main() {
    std::string rank[13] = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
    std::string suit[4] = {"C", "D", "S", "H"}; // Club, Diamond, Spade, Heart
    std::string deck[52];

    for (int i = 0; i < 13; i++)
        for (int j = 0; j < 4; j++)
            deck[4*i + j] = rank[i] + suit[j];

    int N; std::cin >> N;
    for(int i=0; i<N; i++) {
        double u = rand()/((double)RAND_MAX+1);
        std::cout << deck[(int)(u*52)] << " ";
    }
    std::cout << std::endl;
}
```

Note: Sample is with replacement (same card can appear multiple times).

Array application: shuffle and deal from a deck of cards

Problem: Print N random cards from a deck.

Algorithm: Shuffle the deck, then deal.

- Consider each card index i from `0` to `51`.
  - Calculate a random index r between `i` and `51`.
  - Exchange `deck[i]` with `deck[r]`
- Print the first N cards in the deck.

Implementation:

```
for (int i = 0; i < 52; i++) {
        double u = rand()/((double)RAND_MAX+1);
        int r = i + (int) (u * (52-i));
        std::string t = deck[r];
        deck[r] = deck[i];
        deck[i] = t;
    }
```

each value between i and 51 equally likely

exchange deck[i] and deck[r]

Array application: shuffle a deck of 10 cards (trace)

```
for (int i = 0; i < 10; i++) {
        double u = rand()/((double)RAND_MAX+1);
        int r = i + (int) (u * (10-i));
        std::string t = deck[r];
        deck[r] = deck[i];
        deck[i] = t;
    }
```

i	r	deck									
		0	1	2	3	4	5	6	7	8	9
		2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣
0	7	9♣	3♣	4♣	5♣	6♣	7♣	8♣	2♣	10♣	J♣
1	3	9♣	5♣	4♣	3♣	6♣	7♣	8♣	2♣	10♣	J♣
2	9	9♣	5♣	J♣	3♣	6♣	7♣	8♣	2♣	10♣	4♣
3	9	9♣	5♣	J♣	4♣	6♣	7♣	8♣	2♣	10♣	3♣
4	6	9♣	5♣	J♣	4♣	8♣	7♣	6♣	2♣	10♣	3♣
5	9	9♣	5♣	J♣	4♣	8♣	3♣	6♣	2♣	10♣	7♣
6	8	9♣	5♣	J♣	4♣	8♣	3♣	10♣	2♣	6♣	7♣
7	9	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣
8	8	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣
9	9	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣

Q. Why does this method work?

- Uses only exchanges, so the deck after the shuffle has the same cards as before.
- $N - i$ equally likely values for deck[i].
- Therefore $N \times (N-1) \times (N-1) \ldots \times 2 \times 1 = N!$ equally likely values for deck[].

Initial order is immaterial.

Note: Same method is effective for randomly rearranging any type of data.

Coupon collector

Coupon collector problem

M different types of coupons.

Collector acquires random coupons, one at a time, each type equally likely.

Q. What is the expected number of coupons needed to acquire a full collection?

Example: Collect all ranks in a random sequence of cards (M =13)

Sequence

9♣	5♥	8♠	10♠	2♦	A♣	10♦	Q♥	3♦	9♠	5♠	9♦	7♦	2♣	8♠	6♦	Q♦	K♣	10♣	A♠	4♦	J♥

Collection

2	3	4	5	6	7	8	9	10	J	Q	K	A
2♦	3♦	4♦	5♥	6♦	7♦	8♠	9♣	10♠	J♥	Q♥	K♣	A♣
2♣			5♠			8♠	9♠	10♦		Q♦		A♠
							9♦	10♣				

22 cards needed to complete collection

Array application: coupon collector

Coupon collector simulation

- Generate random int values between 0 and M −1.
- Count number used to generate each value at least once.

Key to the implementation

- Create a `bool` array of length M. (Initialized to `false`)
- When r generated, check the rth value in the array.
  - If `true`, ignore it (not new).
  - If `false`, count it as new distinct value (and set rth entry to true)

```
#include<iostream>
#include<cstdlib>

int main() {
    srand(time(0));
    const int M = 10;
    int cards = 0; // number of cards collected
    int distinct = 0; // number of distinct cards
    bool found[M] = {}; // initialize to false
    while (distinct < M) {
        int r = rand() % M;
        cards++;
        if (!found[r]) {
            distinct++;
            found[r] = true;
        }
    }
    std::cout << cards;
    return 0;
}
```

Array application: coupon collector (trace for M = 6)

```
bool found[M] = {};
while (distinct < M) {
    int r = rand() % M;
    cards++;
    if (!found[r]) {
        distinct++;
        found[r] = true;
    }
}
```

r	found						distinct	cards
	0	1	2	3	4	5		
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

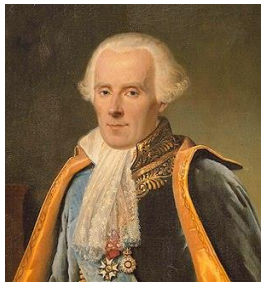
Simulation, randomness, and analysis (revisited)

Coupon collector problem

M different types of coupons.

Collector acquires random coupons, one at a time, each type equally likely.

Q. What is the expected number of coupons needed to acquire a full collection?



Pierre-Simon Laplace
1749-1827

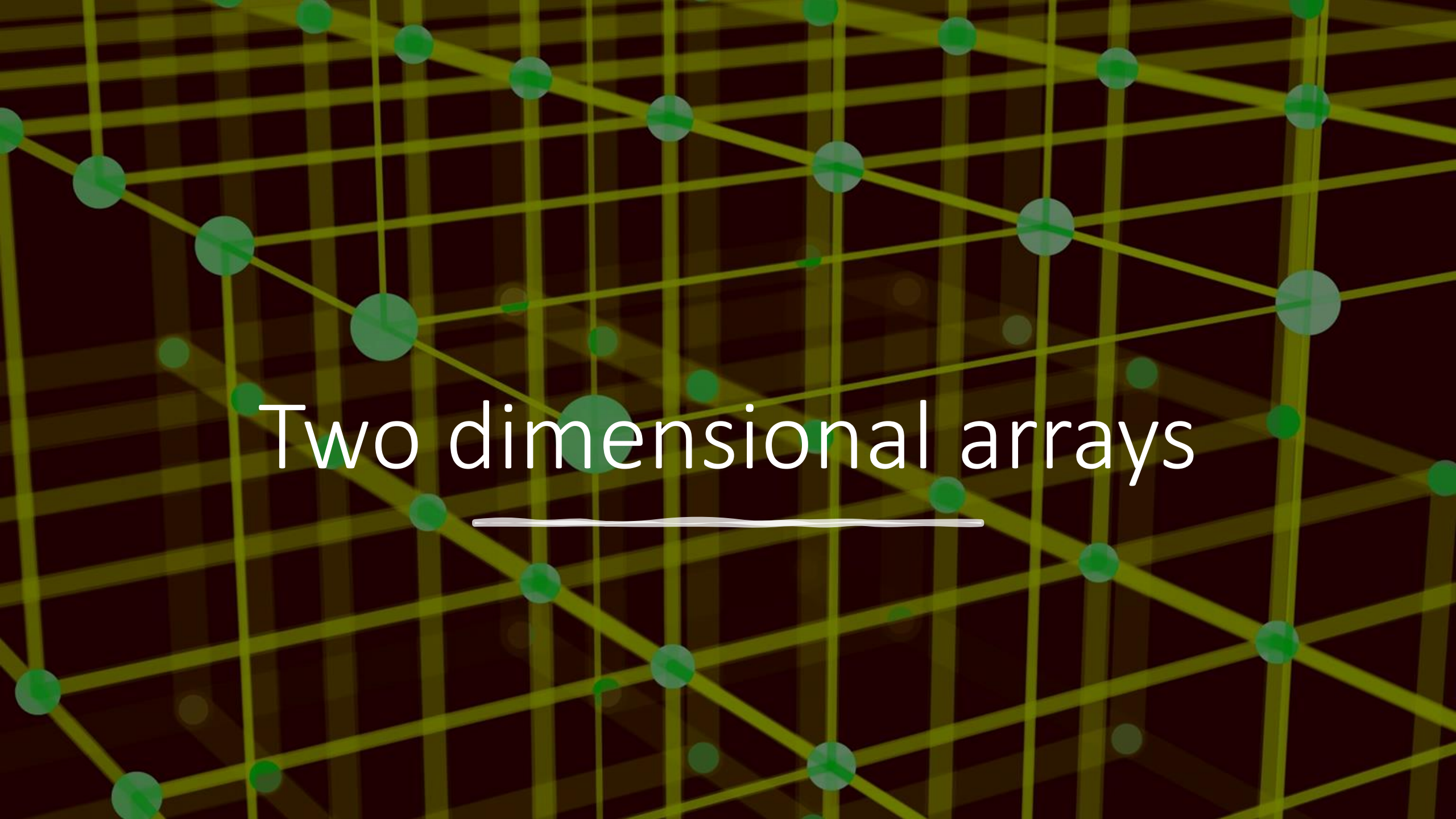
A. (known via mathematical analysis for centuries) About $M \ln M + .57721M$.

type	M	expected wait
playing card suits	4	8
playing card ranks	13	41
Pokemon TCG	15,078	153,768
Magic TG	22,630	

Remarks

- Computer simulation can help validate mathematical analysis.
- Computer simulation can also validate software behavior.

Example: Is rand() simulating randomness?


Two dimensional arrays

Two-dimensional arrays

A two-dimensional array is a doubly-indexed sequence of values of the same type.

Examples

- Matrices in math calculations.
- Grades for students in an online class.
- Outcomes of scientific experiments.
- Transactions for bank customers.
- Pixels in a digital image.
- Geographic data
- ...

course id

student id	0	1	2	3	4	5	6	…
0	B	A	B	C	A	D	A	
1	A	B	B	C	A	C	C	
2	C	C	A	A	C	B	B	
3	C	A	B	B	A	A	C	
4	A	D	A	C	C	B	A	
5	B	C	D	A	B	C	A	
6	A	B	A	B	B	A	C	
…								

Main purpose. Facilitate storage and manipulation of data.

C++ language support for two-dimensional arrays

Basic Support

operation	typical code
Declare an array	`int a[1000][1000];`
Declare and initialize an array	`int a[2][3] = { {10, 20, 30}, {15, 25, 35} };`
Refer to an array entry by index	`a[i][j] = b[i][k] + c[k][j];`
Refer to number of rows	`std::size(a)`
Refer to row i	`a[i]`
Refer to number of columns	`std::size(a[i])`

no way to refer to column j

a

a[1]

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]	a[0][6]	a[0][7]	a[0][8]	a[0][9]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]	a[1][6]	a[1][7]	a[1][8]	a[1][9]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]	a[2][6]	a[2][7]	a[2][8]	a[2][9]

a 3-by-10 array

Two-dimensional arrays: Initialization options

operation	typical code
Only declaration, no initialization	`int a[1000][1000];`
Initialize to default value (0 for numeric data types)	`int a[1000][1000] {};`
Declare and initialize an array	`int a[2][3] = { {10, 20, 30},` `{15, 25, 35} };`
Left –most dimension can be omitted when given initialization list	`int a[][3] = { {10, 20, 30},` `{15, 25, 35} };`
Remaining entries initialized to default value	`int a[2][3] = { {10, 20, 30} }`

Remaining entries initialized to default value	`int a[][3] = { {10, 20},`
	`{},`
	`{15} };`

Can be initialized using 1-d initialization list (row-major order) Remaining entries initialized to default value	`int a[2][3] = {10, 20, 30, 15};`

Application of arrays: vector and matrix calculations

Mathematical abstraction: vector
C++ implementation: 1D array

Mathematical abstraction: matrix
C++ implementation: 2D array

Vector addition

```
double c[N];
for (int i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

0.2	0.5	1.2	+	0.4	0.3	0.2	=	0.6	0.8	1.4

Matrix addition

```
double c[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
```

0.2	0.5	1.2		0.4	0.3	0.2		0.6	0.8	1.4
0.7	0.9	0.1	+	0.9	0.5	0.8	=	1.6	1.4	0.9
0.9	1.0	0.0		0.7	0.1	1.0		1.6	1.1	1.0

Application of arrays: vector and matrix calculations

Mathematical abstraction: vector
C++ implementation: 1D array

Vector dot product

```
double sum = 0.0;
for (int i = 0; i < N; i++)
   sum += a[i]*b[i];
```

.30	.60	.10	·	.50	.10	.40	=	.25

i	a[i]	b[i]	a[i]*b[i]	sum
0	0.3	0.5	0.15	0.15
1	0.6	0.1	0.06	0.21
2	0.1	0.4	0.04	0.25

end of loop trace

Mathematical abstraction: matrix
C++ implementation: 2D array

Matrix multiplication

```
double c[N][N] = {}; // init to 0
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

.70	.20	.10		.80	.30	.50		.59	.32	.41
.30	.60	.10	*	.10	.40	.10	=	.31	.36	.25
.50	.10	.40		.10	.30	.40		.45	.31	.42

Pop quiz 3 on arrays

Q. How many multiplications to multiply two N-by-N matrices?

```
double c[N][N] = {};
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

1. N
2. N^2
3. N^3
4. N^4

Pop quiz 3 on arrays

Q. How many multiplications to multiply two N-by-N matrices?

```
double c[N][N] = {};
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

1. N
2. N^2
3. N^3
4. N^4

Nested for loops: $N \times N \times N$

Self-avoiding random walks

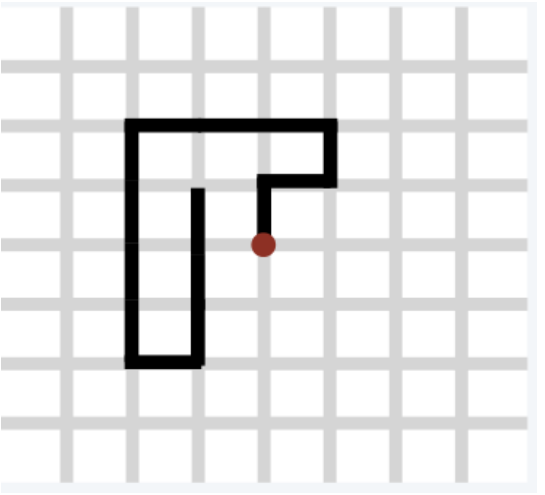
A cat walks around at random in a city, never revisiting any intersection.

Q. Does the cat escape?

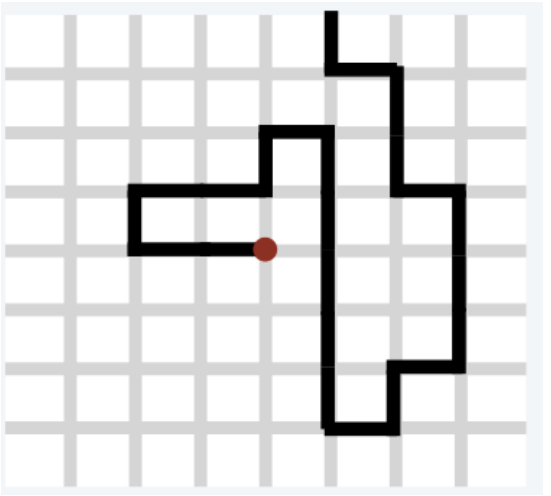
Model: a random process in an N-by-N lattice

- Start in the middle.
- Move to a random neighboring intersection but do not revisit any intersection.
- Outcome 1 (escape): reach edge of lattice.
- Outcome 2 (dead end): no unvisited neighbors.

Q. What are the chances of reaching a dead end?

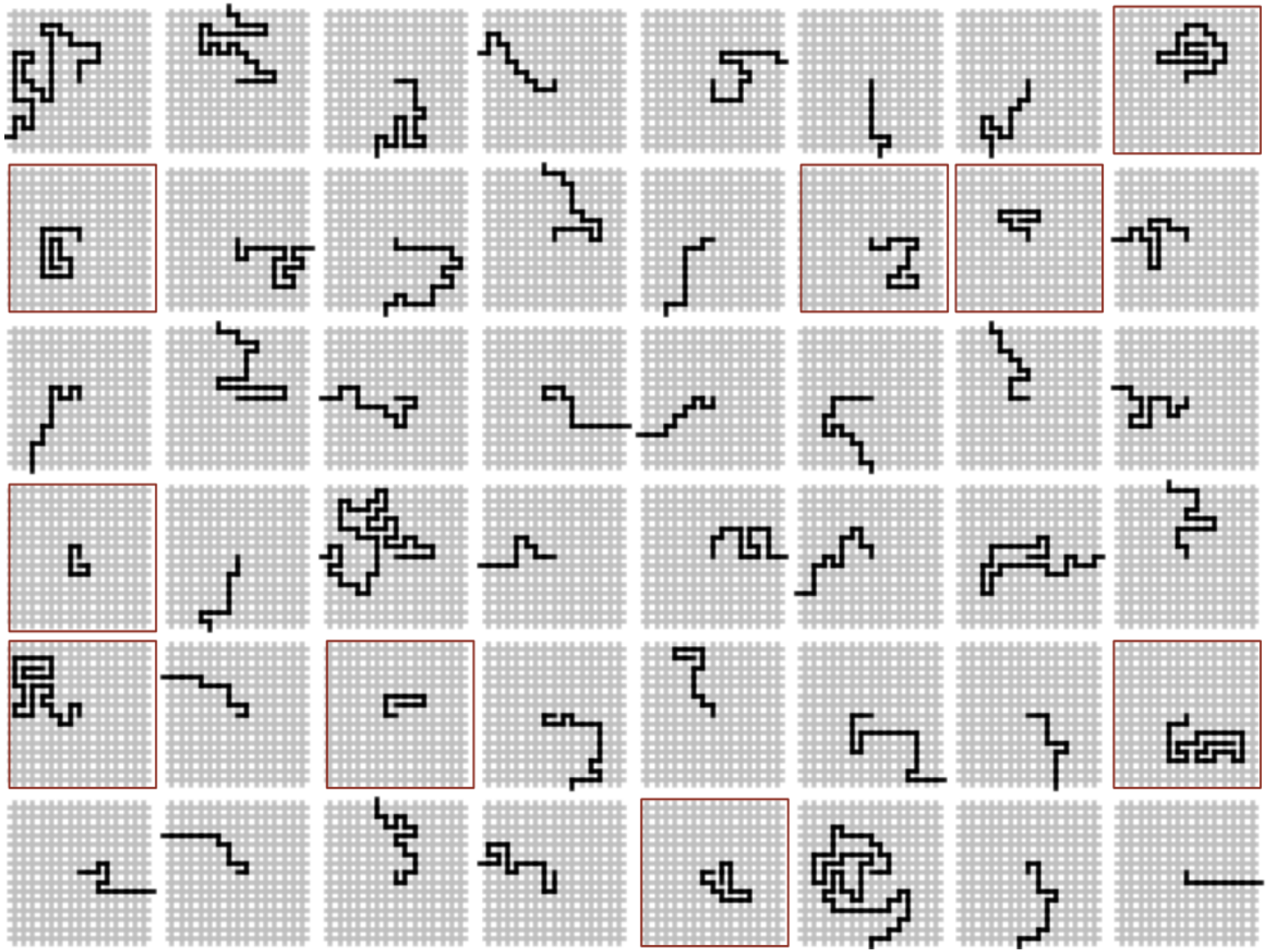


dead end



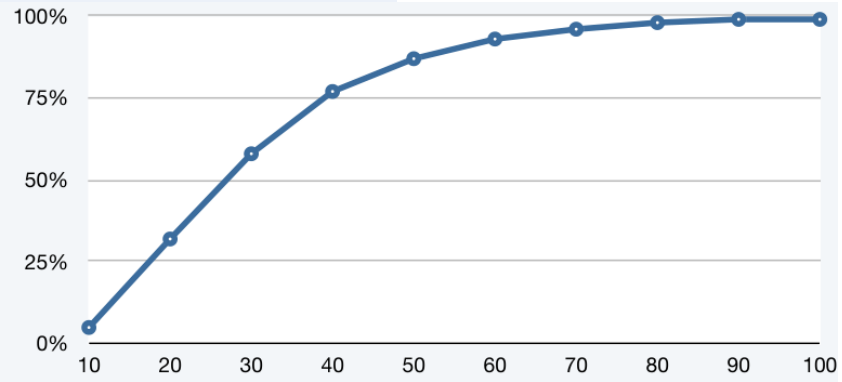
escape

Self-avoiding random walks



Application of 2D arrays: self-avoiding random walks

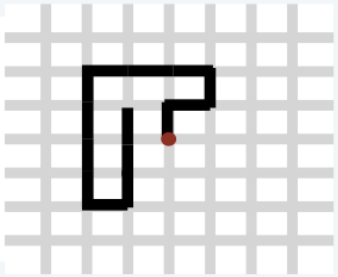
```
const int N = 20, trials = 1000;
int deadEnds = 0;
for (int t = 0; t < trials; t++)  {
    bool a[N][N] = {};
    int x = N/2, y = N/2;
    while (x > 0 && x < N-1 && y > 0 && y < N-1)  {
        if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            { deadEnds++; break; }
        a[x][y] = true;
        int r = rand() % 4;
        if (r==0)      { if (!a[x+1][y]) x++; }
        else if (r==1) { if (!a[x-1][y]) x--; }
        else if (r==2) { if (!a[x][y+1]) y++; }
        else           { if (!a[x][y-1]) y--; }
    }
}
cout << (100*deadEnds/trials) << "% dead ends";
```



Simulation, randomness, and analysis (revisited again)

Self-avoiding walk in an *N*-by-*N* lattice

- Start in the middle.
- Move to a random neighboring intersection (do not revisit any intersection).



Applications

- Model the behavior of solvents and polymers.
- Model the physics of magnetic materials.
- (many other physical phenomena)

Paul Flory
1910-1985
Nobel Prize 1974

Q. What is the probability of reaching a dead end?

A. Nobody knows (despite decades of study). ← Mathematicians and physics researchers cannot solve the problem.

A. 99+% for $N > 100$ (clear from simulations). ← YOU can!

Computational models play an essential role in modern scientific research.

Remark: Computer simulation is often the *only* effective way to study a scientific phenomenon.

Your first data structure

Arrays: A basic building block in programming

- They enable storage of large amounts of data (values all of the same type).
- With an index, a program can instantly access a given value.
- Efficiency derives from low-level computer hardware organization (stay tuned).