computer have memory to store data. every program gets a piece of it to use

as we create and use more variables, more space is allocated to a program

overhead is just some extra memory the computer needs to use to manage everything
recall function spaces

remember how functions create spaces to do operations and store data

```c
int fxn2() {
    int x = 10;
    return 5;
}

int fxn1() {
    return fxn2();
}

int main() {
    int n = 5;
    fxn1();
}
```
memory addresses

Every location in memory has it.

```
memory

int x
fxn2()
fxn1()
int n
main()

<table>
<thead>
<tr>
<th>address</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>5</td>
</tr>
</tbody>
</table>
```

Variable at address 0 is x
Variable at address 3 is n
pointers

variables that can point to memory addresses

pointer operations:

```c
int* x;  // declares a pointer to an integer

x = &y;  // assigns pointer to address of y

a = *x;  // dereferences pointer for access to value x points to

*x = 5;  // dereferences pointer to set the value x points to
```
```cpp
#include <iostream>
using namespace std;

int main() {
    int x, y;
    int *px;  // declares a pointer to an integer
    x = 2;
    y = -5;

    px = &x;  // assign's x's memory address to px
    cout << "px: " << px << endl;
    cout << "*px: " << *px << endl;

    px = &y;  // assign's y's memory address to px
    cout << "px: " << px << endl;
    cout << "*px: " << *px << endl;
}
```

```
px: 0012AA3C
*px: 2
px: 0012AA4C
*px: -5
```
int x = 5;
int* px = &x;
int y = 15;
int* py = &y;
int a = -4;
int* pa = &a;
int* py2 = &y;

some statements (changes persist) what's the output?
cout << x + y
20
cout << *px + *py
20
y = *px; cout << y;
5
px = py; cout << *px;
5
py = pa; *pa = y; cout << *py;
5
px = *x; cout << *px;
error
cout << *py2 + *py + *px + *pa;
20
pointers + arrays

Array variables are actually pointers that can’t point to anything else (pointee is fixed)

\[
x \rightarrow \begin{array}{cccccccccc}
0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & 81 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\x is a pointer to the first element in the array. Note that in memory, the elements of arrays are laid out consecutively. Thus, if we know the location of the first element, then we can get to any element.

```
int x[10];    // declares a pointer to an integer
cout << x;    // prints the array’s address
cout << x[0]; // prints the value of the first element
cout << *x;   // also prints the value of the first element
cout << &x[0]; // prints the first element’s address, which is also the array’s address
cout << &x[1]; // prints the second element’s address
cout << x + 1; // also prints the second element’s address
```
Pointers arithmetic

When you print memory addresses, they are not consecutive, why is that?

Because memory addresses are **byte-aligned**, and an integer takes 4 bytes.

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A1400</td>
<td>x[0]</td>
</tr>
<tr>
<td>0A1404</td>
<td>x[1]</td>
</tr>
<tr>
<td>0A1408</td>
<td>x[2]</td>
</tr>
<tr>
<td>0A140C</td>
<td>x[3]</td>
</tr>
<tr>
<td>0A1410</td>
<td>x[4]</td>
</tr>
</tbody>
</table>
```

Now let’s dereference `x`.

- `cout << x;` prints the array’s address: 0A1400
- `cout << x + 1;` prints the 2nd element’s address: 0A1404
- `cout << x + 2;` prints the 3rd element’s address: 0A1408

Because `x` is pointing to an integer, it automatically knows to increment `x` by 4 to get to the address of the next integer. Now let’s dereference `x`.

```
*x  # Dereferences x, same as x[0];
*(x + 1)  # Gets x’s 2nd element, same as x[1];
*(x + 2)  # Gets x’s 3rd element, same as x[2];
*(x + 5)  # Gets x’s 6th element, same as x[5];
.
.
.
```

...
some statements

*x + *y
*(x + 9) + *(y + 4)
y[9] + *(x + 4)
*(y + *x)
y[*y[9] + *(x + 6)]
*(x + y[9])
*(x + *(y + *(x + *(y + 9))))
*(x + *(y + *x))

what's the output?

6
error
7
-4
4
3
2
error
headers + array snippets

const int N = 7;
double x[N] = {5, 10, 20, -10, 20, 100, -20};
double* px = &x;
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum += x[i];
}
cout << "sum: " << sum << endl;

sum += *(x + i);
reports the sum of the array: 125

const int N = 7;
double x[N] = {5, 10, 20, -10, 20, 100, -20};
double* px = &x;
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum += x[i];
    x[i] = sum;
}
cout << "sum: " << sum << endl;

const int N = 7;
double x[N] = {5, 10, 20, -10, 20, 100, -20};
double* px = &x;
for (int i = 1; i < N; i++) {
    x[i] = x[i] + x[i-1];
}
cout << "sum: " << x[N-1] << endl;

*(x + i) += *(x + i - 1)
... << *(x + N - 1) << ...
also reports the sum of the array: 85
also sets x to the following:

<table>
<thead>
<tr>
<th>5</th>
<th>15</th>
<th>35</th>
<th>25</th>
<th>45</th>
<th>145</th>
<th>125</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
pointers + functions

element function header for sum function with some pointer input

    int sum(int* a, int n)

why the n? again arrays or pointers to the beginning of arrays don’t know their own size.

pointers as input arguments:

1. you can change a in the function and when it finishes a will keep those changes
2. to protect a from being changed, define the input argument as const (i.e. “const int* a”)

    if a is declared as const, cout << *(a + 5); *(a + 5) = 10; okay
    compile error

to call the function: use the pointer name (no extra asterisks): int sum = sum(x, 10);

    int sum = sum(*x, 10); compile error, since *x is an integer and not an integer pointer

can my function return a pointer? YES! well sort of. sum can return a pointer to the array,
just do “int* sum(int* a, int n)”
int sum(int x[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += x[i];
    }
    return sum;
}
“int *x” and “int x[]” are pretty much equivalent for parameter declarations

```c
int sum(int *x, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(x + i);
    }
    return sum;
}
```
```c
int sum(int *x, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *x; x++;
    }
    return sum;
}
```

“int *x” and “int x[]” are pretty much equivalent for parameter declarations.
deleting duplicates in arrays with pointer access

If there's a duplicate, return it. Else, return -1.

```c
int find_duplicate(int* x, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (*(x + i) == *(x + j)) {
                return *(x + i);
            }
        }
    }
    return -1;
}
```

Cool, but let's go and delete that duplicate from the array, so let's return the position of the first instance, so we can remove it.

```c
int find_duplicate(int* x, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (*(x + i) == *(x + j)) {
                return i;
            }
        }
    }
    return -1;
}
```

Now how do we delete it? We can’t just set it to 0, that’s another value. We’ll replace it with the last element, and then say the array has shrank.

```c
int remove_dup(int* x, int n, int index) {
    *(x + index) = *(x + n - 1);
    return n - 1;
}
```

Shoots, what if there’s more duplicates? How do we remove them all? Just do it over and over again.

```c
int remove_duplicates(int* x, int n) {
    int dup_index = find_duplicate(x, n);
    while (dup_index != -1) {
        n = remove_dup(x, n, dup_index);
    }
    return n;
}
```

Lastly, put it to the test!

```c
int main() {
    int x[9] = {1, 2, 1, 0, 2, 1, 2, 5, 1};
    int* px = &x, py = &x;

    int s1 = remove_duplicates(px + 3, 8);
    int s2 = remove_duplicates(px, s1);
    int s3 = remove_duplicates(py, 8);
}
```

How’s x look after each call? what’s s1, s2, s3?