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

<table>
<thead>
<tr>
<th>function</th>
<th>address</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>0</td>
<td>int x</td>
<td>10</td>
</tr>
<tr>
<td>fxn1()</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fxn2()</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int n</td>
<td>3</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- variable at address 0 is x
- value is 10
- variable at address 3 is n
- value is 5
pointers

variables that can point to memory addresses

pointer operations:

```
int* x;

x = &y;

int a = *x;

*x = 5;
```

declares a pointer to an integer

assigns pointer to address of y
dereferences pointer to access the value x points to
dereferences pointer to set the value x points to

assuming y happens to exist

```
int y [10]

x

int y [10]

x

int y [10]

x
```
```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* py = &y;
int* px = &x; int a = -4;
int y = 15; int* pa = &a;
int* py2 = &y;

some statements (changes persist)

`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

what’s the output?
array variables are actually pointers that can’t point to anything else (pointee is fixed)

x points to the first element in the array. 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];
cout << x;
cout << x[0];
cout << *x;
cout << &x[0];
cout << &x[1];
cout << x + 1;
x = x + 1;
```

declares a pointer (with fixed pointee) to an integer
prints the array’s address
prints the value of the first element
also prints the value of the first element
prints the first element’s address, which is also the array’s address
prints the second element’s address
also prints the second element’s address
ERROR! x cannot be reassigned to something else
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.

```
0A1400  x[0]
0A1404  x[1]
0A1408  x[2]
0A140C  x[3]
0A1410  x[4]
```

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, C++ 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];
...  // ...
```
pointer arithmetic practice

some statements

\*x + \*y
\*(x + 9) + \*(y + 4)
y[9] + \*(x + 4)
\*(y + \*x)
y[\*(x + 1) + \*(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
convert these code snippets to use pointer access instead of array access. What do they do?

```c++
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;
```

```
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;
```

```
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;
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum += *(x + i);
}
```

```
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 - 1);
}
```

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

```
sum += *(x + i);
*(x + i) = sum;
```

reports the sum of the array: 125

sets x to the following:

<table>
<thead>
<tr>
<th></th>
<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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

```
s(x + i) += (x + i - 1)
```

... *(x + N - 1) ...

also reports the sum of the array: 125

also sets x to the following:

<table>
<thead>
<tr>
<th></th>
<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>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

example 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
   “assignment of read-only location”

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! functions can return pointers. In this case, sum can return a pointer to the array a, 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 sum(int *x, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *(x + i);
    }
    return sum;
}
sum example v2

"int *x" and "int x[]" are nearly equivalent for parameter declarations

<table>
<thead>
<tr>
<th>return type</th>
<th>function name</th>
<th>input array</th>
<th>input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>sum</td>
<td>int *x, int n</td>
<td>int n</td>
</tr>
</tbody>
</table>

```c
int sum(int *x, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += *x; x++;
    }
    return sum;
}
```
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 to delete that duplicate from the array, 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.

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

Shoots, what if there are more duplicates? How do we remove them all?

```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);
        dup_index = find_duplicate(x, n);
    }
    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, s1);
}
```

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