27. Pointers and Arrays

Objectives

- Pointer arithmetic
- Understand the relationship between pointers and arrays
- Allocate and deallocate memory for dynamic arrays in the heap
- Understand array parameters in functions
- Implement stacks and queues using dynamic arrays
- Understand and implement directed graph using adjacency matrix
- Model social media graphs

In this set of notes, we see how pointers can be used to create dynamic arrays, i.e., pointers that point to arrays of with variable sizes.

Pointers are not arrays. But rather they have the address of the first value in an array. However in terms of syntax and usage, pointers do seem like arrays. I'll talk about the some of the similarities and differences between arrays and pointers. But the point is, once p points to an array, then the i-th index value of the array that p points to is p[i], which is exactly the array notation that you're already used to.

An array is like a container of values. We store information in these containers. (At this point you have only seen array of integers, of doubles, etc. Later on you will learn about complex values, values such as products in a company where each product contains product name, price, UPC code, etc.) The problem is that the arrays you have seen up to this point have fixed sizes. But no one can predict how large a container should be. You might think an array of size 10000 is enough. But what if after some time, you actually need 100000? Think for instance of a game. At the beginning of the game, maybe there are only 10 enemy spaceships and at most 100 lasers are flying around in the screen. But at a later level, maybe there are 1000 enemy spaceships and 1000000 lasers. Is that enough? Maybe not for an even later level.

Dynamic arrays allows you to handle the situations like the above and they are therefore extremely important. There are actually many, many, many types of containers: fixed size arrays, dynamic arrays, and then in CISS350, I'll show you linked lists, trees, hashtables, etc.

Pointers also allow you to create complex 2D arrays which are not rectangles. For instance if you want to create a 2D array where the first row has 1000000 values, the second has 5000, the third has 20000, and the fourth has 20, it can be done. Without pointers, you would have to create a 2D array with 4 rows and each with 1000000 values.

Pointer arithmetic

Try this:

```
int x = 42;
int * p = &x;
int * q = p + 1;
std::cout << (unsigned int)(p) << ' '
          << (unsigned int)(q) << '\n';
```

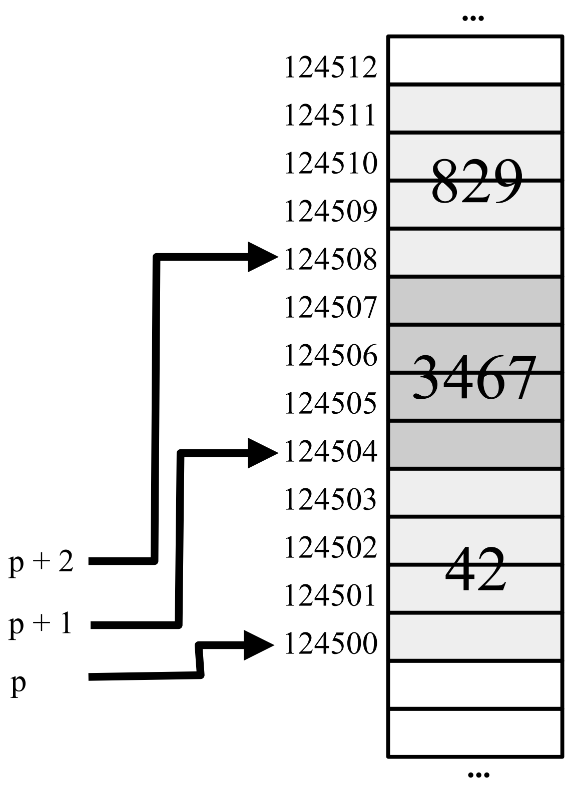
Aha! You can add integers to pointers. Well this is not exactly shocking since addresses are integers.

Note the address value stored in `q` ... numerically speaking the value of `q` is actully the numerical value of `p` plus 4. when you add 1 to the address of an `int` address – IT'S NOT ONE!!! Why 4? Because … **4 comes from `sizeof(int)`**.

But why does it work this way? Because this allows you to think like this:

> "`p + 1` points to the `int` value that is immediately after the `int` value that `p` points to."

Of course p + 2 has address value that is 8 away from 8. So if p has address value 124500, then you should have this picture in your mind:



Now you might be wondering: "Wait … `p` points to `x` and I do see that the value of `x` is 42. But what is the 3467 that `p + 1` points to?".

`p + 1` simply points to an integer (because `p` is a pointer to `int`), i.e., the four bytes starting at `p + 1` as an integer just happens to be 3467. It does not belong to any variable. It's just part of your computer's memory. And of course it could be garbage and the value would be different next time you run the same program.

The above applies to any type and not just `int`. In general suppose `T` is a type (`int` or a `char` or a `double` or a `bool` or what-have-you), and you have the following:

```
T * p = &x;
```

and suppose a value of type `T` uses k bytes (which can be obtained using `sizeof(T)`), then numerically the address value of `p + 1` is always the address value of `p` plus `k`. Also, if we have

```
T * p = &x;
T * q = p - 1;
```

then `q` is k smaller than `p`.

Remember ...

- if `p` is an `int` pointer (and if you're using a 32-bit machine), then `p + 1` does **not** mean "next address value after `p`". Rather, it means address of the next **integer** after the **integer** `p` points to.
- If `q` is a pointer that points to a `double`, then `q + 1` is the address of the next **double** after the **double** `q` points to.
- If `r` is a pointer that points to a character, then `r + 1` is address of the next **character** after the **character** `r` points to.

Get it?

By the way:

`sizeof(double)`	is 8	(a double value takes up 8 bytes)
`sizeof(char)`	is 1	(a char value takes up 1 byte)

However the `sizeof(int)` can vary depending on your machine.

If you want to measure the difference between two pointers, you have to type cast the address values not to unsigned int, but to int so that you can get negative values. Here's an example:

```
int x = 42;
int * p = &x;
int * q = p + 1;
int * r = p - 1;
std::cout << (int) q - (int) p << ' '
          << (int) r - (int) p << '\n';
```

Exercise. Think about this: Is `(int) (p + 1)` the same as (`(int) p) + 1`?

Exercise. There are four pointer values (i.e., address values) in the following code segment. The first one is `p`. As for the rest:

- Is `p + 2` larger or smaller than `p`? How far apart are they?
- Is `p - 1` larger or smaller than `p`? How far apart are they?
- Is `p - 3` larger or smaller than `p`? How far apart are they?

```
int x = 42;
int * p = &x;
std::cout << (unsigned int)(p)  << ' '
          << (unsigned int)(p + 2) << ' '
          << (unsigned int)(p - 1) << ' '
          << (unsigned int)(p - 3) << '\n';
```

Verify your answer by running the program.

Exercise. There are four pointer values (i.e., address values) in the following code segment. The first one is `p`. As for the rest:

- Is `p + 2` larger or smaller than `p`? How far apart are they?
- Is `p - 1` larger or smaller than `p`? How far apart are they?
- Is `p - 3` larger or smaller than `p`? How far apart are they?

```
double d = 3.14;
double * p = &d;
std::cout << (unsigned int)(p)  << ' '
          << (unsigned int)(p + 5) << ' ';
          << (unsigned int)(p - 2) << ' ';
          << (unsigned int)(p - 4) << '\n';
```

Verify your answer by running the program.

Exercise.

- Can we use pre- and post- increment and decrement operators on pointers?
- What about augmented assignment operators?
- What is the numeric value of the difference in address value of `q`, `r`, `s` and `p` in the following:

```
int x = 42;
int * p;
int * q = p;
int * r = p;
int * s = p;
q++;
```

```
r += 5;
s--;
std::cout << (int) q - (int) p
          << ' '
          << (int) r - (int) p
          << ' '
          << (int) s - (int) p
          << '\n';
```

(Recall that if you have a newer laptop, you might need to use **`long long int`**.)

Exercise. What is the output:

```
double d = 3.14;
double * p = &d;
double * q = p;
--q;
double * r = q;
r += 4
double * s = r;
--s;
std::cout << (int) q - (int) p
          << ' '
          << (int) r - (int) p
          << ' '
          << (int) s - (int) p << '\n';
```

Exercise. You have the following:

```
char c = 'a';
char * p = &c;
char * q = p + 8;
```

How much larger is `q` when compared with `p`? Verify it yourself. (If you get something different from the expected on your computer, don't panic. Talk to me.)

Exercise. For a 32-bit machine, the integer values you can represent is about -2 billion to 2 billion. More exactly, the range of values is from

$$-2^{31}, ...2^{31} - 1$$

(Why 31? It's 1 less than 32, the number of bits for a 32-bit machine.)

There is another integer type called the **long integer**. The formal name is `long` or `long int`. Example:

```
long i = 0L;
long int j = 42; // OK ... C++ will type cast
                 // 42 to 42L for you
```

(Most programmers type `long` instead of `long int`.)

(a) How many bytes of memory is used by a `long` value on your compiler? (Write a simple program.)

(b) What is the output of the following code segment?

```
long * p = new long;
std::cout << (int) p - (int) (p + 2) << '\n';
```

(First do this without your compiler. Then, check with your compiler.)

Exercise. There is yet another integer type called the **long long integer**. The formal name is `long long` or `long long int`.
Example:

```
long long i = 0LL;
long long int j = 42; // OK ... C++ will type
                      // cast 42 to 42LL for you
```

(Most programmers just use `long long` instead of `long long int`.)

(a) How many bytes of memory is used by a `long long` value on your compiler? (Write a simple program.)

(b) What is the output of the following code segment?

```
long long * p = new long long;
std::cout << (int) p - (int) (p + 2) << '\n';
```

(First do this without your compiler. Then, check with your compiler.)

Memory layout of local variables

Suppose you have two integer variables declared next to each other:

```
int x = 42;
int y = 43;
```

Then the address of y is **4 smaller** than the address of x – assuming that **`sizeof(int)` is 4**. (If on your machine, the `sizeof(int)` is 8, then the address of `y` is 8 smaller.)

Caveat: Note that that difference between `&x` and `&y` can be more than 4 on your computer, but it cannot be less than `4`. The point is that an integer takes up `sizeof(int)` number of bytes. But your computer (and your compiler) might pad some extra bytes between the memory used by two adjacent integer variables. On my Windows machine, 8 extra bytes were padded. On my linux machine, the padding is 0. In the notes below, I'm going to assume that the amount of extra padding is 0. To check on the padding of your machine, using the above code, print the address of x and the address of y and subtract. If the difference is 12, then you know that the padding is 12 – 4 = 8 since the `sizeof(int)` is 4.

First run this:

```
int x = 42, y = 43, z = 44;
std::cout << (unsigned int) &x << ' '
          << (unsigned int) &y << ' '
          << (unsigned int) &z << '\n';
int * p = &x;
int * q = p - 1;
std::cout << (unsigned int)(p) << ' '
          << (unsigned int)(q) << '\n';
```

(Recall: you might want to use `unsigned long long`.) AHA … so … `q` is indeed pointing to `x`! Try this:

```
int x = 42, y = 43, z = 44;
std::cout << (unsigned int) &x << ' '
          << (unsigned int) &y << ' '
          << (unsigned int) &z << '\n';
int * p = &x;
int * q = p - 1;
std::cout << (unsigned int)(p) << ' '
          << (unsigned int)(q) << '\n';
*q = 0;
std::cout << x << '\n';
```

Sure enough, `*q = 0` changes `x` to 0!!!

Caveat: In my case, because my linux machine does not pad extra bytes between the memory of `x` and the memory of `y`, for `q` to point to `y`, I only need `q = p - 1`. On my Windows machine, there is a padding of 8 bytes, which means that for my Windows machine, I would need to do `q = p - 1 - 2`, i.e., I have to subtract 2 more because of the 8-byte padding.

So if the address of `x` is 1000 (and assuming there is no extra padding), then the address of `y` is 996 and the address of `z` is 992.

(In some rare cases, your computer might arrange to have the addresses going up instead of down. But that's rare.)

In general if you have two variables x and y of type T which are declared next to each other (x first, then y), then

address of `y` = address of `x - sizeof(T)`

Exercise. Verify what I just said. For instance look at this:

```
double a = 3.14, b = 2.71, c = 1.41;
double * p = &b;
```

Check that `p + 1` points to `a` and `p - 1` points to `c`. Since a `double` takes up 8 bytes, the memory address value in `p + 1` will be 8 larger than the value in `p` and the memory address value in `p - 1` will be 8 larger than the value in `p`

Exercise. What is the output?

```
int x = 42, y = 0, z = 1000;
int * p = &x;
int * q = p - 1;
*q = 5;
q = p - 2;
*q = 99;
std::cout << x << ' ' << y << ' ' << z << std::endl;
```

(Again I'm assuming no extra padding; otherwise you would need to adjust the following).

Exercise. Look at this:

```
int a, b, c, d, e, f, g;
int * p = &a;
```

Using a for-loop, set `a`, `b`, `c`, `d`, `e`, `f`, `g` to `0`, `1`, `2`, `3`, `4`, `5`, `6`. Of course you have to verify. So ...

```
int a, b, c, d, e, f, g;
int * p = &a;

// DO YOUR STUFF HERE USING A POINTER

std::cout << a << ' ' << b << ' ' << c << ' '
          << d << ' ' << e << ' ' << f << ' '
          << g << '\n';
```

Violà! Even though I have variables a, b, c, d, e, f, g (which does not form an array), I can treat them as though they form an array if I use a pointer to scan over them!!! (As another exercise, replace the print statement with a for-loop that prints a,b,c,d,e,f,g.

Exercise. You are given the information that a `double` takes up 8 bytes. Suppose we have

```
double x = 3.14159;
double y = 2.718;
double z = 0.0;
```

Suppose that in the memory layout of the above program, the value for `y` is immediately after the value for `x` and the value for `z` is immediately after `y`. If the address of the value of `x` is 4800000, what is the address of the value for `y`? What is the address of `z`?

Pointer arithmetic and arrays

Try this:

```
int x[3] = {1, 2, 3};

std::cout << (unsigned int) &x[0] << ' '
          << (unsigned int) &x[1] << ' '
          << (unsigned int) &x[2] << '\n';

int * p = &x[0];
std::cout << (unsigned int) p << ' '
          << (unsigned int) (p + 1) << ' '
          << (unsigned int) (p + 2) << '\n';
```

Note two things:

- the values in the array are laid out in **ascending** address values and
- there are **no gaps** in the layout of the memory between values of the array (there's no memory padding – see previous section)

Since `sizeof(int)` is 4,

the address of `x[i + 1]` is the address of `x[i]` plus 4

Note in particular the big difference between a bunch of `int` variables declared together and an array of integer values. In the previous section, you saw that the addresses of a bunch of variables go down. For an array the address of the values go up. For a bunch of variables, there might be extra byte padding. But in the case of an array, there's no extra byte padding.

In the same way, if we have

```
double y[4];
```

since `sizeof(double)` is 8, the addresses of `y[0]`, `y[1]`, `y[2]`, `y[3]` go up by a step of 8.

In general if `z` is an array of type `T` values, then

the address of `z[i + 1]` is the address of `z[i]` plus `sizeof(T)`

This is great because

if `p` points to `z[i]`, then `p + 1` points to

`z[i + 1]`

and

if `p` points to `z[0]`, then `p + i` points to `z[i]` and so `*(p + i)` is the same as `z[i]`

You will see why this is important when we talk about dynamic memory management of arrays in the memory heap.

Exercise. What is the output of this code segment:

```
int x[] = {2, 3, 5, 7, 11, 13};
int * p = &(x[2]);
std::cout << *p << std::endl;
++p;
std::cout << *p << std::endl;
```

Exercise. What is the output of this code segment:

```
double x[] = {0.0, 2.718, 3.141, 42.0};
double * p = &(x[1]);
std::cout << (*p + *(p + 1)) << std::endl;
```

Exercise. What is the output of this code fragment:

```
int x[] = {5, 4, 3, 2, 1};

for (int * p = &x[0]; p <= &x[4]; ++p)
{
    std::cout << *p << ' ';
}
```

Exercise. Complete this code segment.

```
int x[] = {1, 3, 5, 7, 9};

int sum = 0;

// Complete the following to compute the sum
// of the array.
// You must not use an integer index variable but
// rather integer pointer variables.
```

Exercise. Complete this code segment.

```
int x[] = {1, 5, 3, 7, 9};



int * p = &x[0];
int * q = &x[4];

int max = *p;

// Complete the following to compute the maximum
// value in the array.
// You must not use an integer index variable but
// rather integer pointer variables.
```

Exercise. What is the output of this code segment?

```
char x[] = {'h', 'e', 'l', 'l', 'o'};
char * p = &(x[0]);
char * q = &(x[4]);
*p = 'j';
*(p + 2) = 'L';
while (p <= q)
{
    std::cout << *p;
    p++;
}
std::cout << std::endl;
```

(First do this without your compiler. Next, check with your compiler.)

Exercise. What is the output of this code segment:

```
int x[] = {2, 3, 5, 7, 11, 13};
int * p = &(x[2]);
(*p)++;
std::cout << *p << std::endl;
*(p++);
std::cout << *p << std::endl;
```

Can you explain what's happening? (By the way, do you need the parentheses?)

Exercise. What is the output of this code segment:

```
char x[] = "c++ rules";
char * p = &x[0];

while (*p)
{
    std::cout << *p++;
}
```

Can you explain what's happening? (By the way, do you need the parentheses?)

Static arrays

One of the problems with arrays is that they have fixed size.

```
      int x[SIZE]
```

The `SIZE` must be constant.

(WARNING: Some C++ compilers allow you to declare arrays with variable sizes. However this is not standard. Therefore do NOT use variable size for arrays. You do want your C++ code to be as portable as possible, right?)

The nice thing about an array is that it is a local variable, where the memory of the values in the array is allocated within the scope of its declaration and when the array goes out of the scope, the memory is reclaimed. For instance:

```
if (a < 0)
{
    int x[SIZE];
    // ... do something useful with x …

} // x goes out of scope, and array of values
  // used by x is reclaim by the program.
```

In other words, `x` an automatic variable.

Such arrays are called **static arrays**.

The next 4 experiments are extremely important in your understanding of the array, in the context of addresses. Make sure you run them all!

First experiment:

```
#include <iostream>

int main()
{
    int x[5];
    std::cout << x << std::endl;
    return 0;
}
```

You will see an address!!! That tells you that the array `x` is somehow an address!!!

Second experiment:

```
#include <iostream>

int main()
{
    int x[5];
    std::cout << x << std::endl;
    std::cout << &x[0] << std::endl;
    return 0;
}
```

This tells you that if you think of `x` as a pointer, then **`x` is the address of the first value** of your array of value.

So what's happening?!?

Whenever you use `x` as though it's a value, the compiler will actually replace `x` with `&x[0]`, i.e., the address of the first value of of the array:

`x` is the same as `&x[0]`

You also know that you can perform pointer arithmetic. Since `x` (besides being an array/collection of values) can be thought as a pointer, you can think about the pointers:

```
x, x + 1, x + 2, x + 3, x + 4
```

which are the same as

```
&x[0], &x[1], &x[2], &x[3], &x[4]
```

We'll verify this fact using the next experiment …

Third experiment:

```
#include <iostream>

int main()
{
    int x[5];

    for (int i = 0; i < 5; ++i)
    {
        std::cout << x + i << ' ';
    }
    std::cout << std::endl;

    for (int i = 0; i < 5; ++i)
    {
        std::cout << &x[i] << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

The above is about the addresses of the values in an array. What about the values of the array? You know that you can access the values in the array using `x[0],x[1],x[2],x[3],x[4]`

But since `x`, `x + 1`, `x + 2`, `x + 3`, `x + 4` points to the above values, we conclude that you can also access the values of the array using:

`*x,*(x + 1),*(x + 2),*(x + 3),*(x + 4)`

Let's check that …

Fourth experiment ...

```
#include <iostream>

int main()
{
    int x[5] = {2, 3, 5, 7, 11};

    for (int i = 0; i < 5; ++i)
    {
        std::cout << *(x + i) << ' ';
    }
    std::cout << std::endl;

    for (int i = 0; i < 5; ++i)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

You now arrive at the conclusion that

`x[i]` is like `*(x + i)`

In fact given ***<u>any</u>*** pointer `p` (not just an array) and integer `i`, in C/C++,

`p[i]` is the same as `*(p + i)`

Remember that!!!

So let me summarize. When you declare this array variable:

```
int x[5];
```

- You can think of `x` as a pointer that points to the first value in the array of 5 integers, i.e. `x` has the address of `x[0]`. In fact in your code, `x`, is actually translated into `&x[0]`.
- The values `x[0]`, `x[1]`, `x[2]`, `x[3]`, `x[4]` are laid out in memory in increasing address values with no gaps.
- The following addresses are the same:

`x + i`	is the same as	`&x[i]`

- The following values are the same:

  `*(x + i)` is the same as `x[i]`

Exercise. You are given that the address of the first value of the following array

```
int x[100];
```

is 72008004. Assuming that each `int` value takes up 4 bytes of memory, what is the address of the value `x[3]`? What is the memory address of `x[56]`?

Exercise. You have the following declarations:

```
int x = 0;
double y = 2.14;
int a[100];
double b[100];
```

Assuming that an `int` takes up 4 bytes and a `double` takes up 8 byte, the address of `x` is 120004000, what is the address of `a[3]` and `b[20]`? (Assume that there is no gaps in the memory layout of variables and address value grows.)

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = a;
std::cout << *(b + 2) << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = (a + 2);
std::cout << *(b + 1) << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = (a + 3);
std::cout << *(b - 2) + b[1] << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = (a + 3);
std::cout << *(b + 1) + b[-2] << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int b = 0;
for (int i = 0; i < 5; ++i)
{
    b += *(a + i);
}
std::cout << b << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int b = a + 3;
std::cout << *(b - 1) << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = a + 4;
std::cout << *(b - 3) + *b << '\n';
```

Verify!

Exercise. What is the output of this code fragment:

```
int a[5] = {2, 3, 5, 7, 11};
int * b = a;
int * c = a + 1;
int s = 0;
for (int i = 0; i < 3; ++i)
{
    s += *c - *b;
    ++b;
    ++c;
}
std::cout << s << '\n';
```

Verify!

Exercise. Rewrite the following so that the array bracket operator `[]` is not used.

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

int main()
```

```
{
    srand((unsigned int) time(NULL));
    const int SIZE = 100;
    const double MAX = double(RAND_MAX);

    double x[SIZE];
    for (int i = 0; i < SIZE; i++)
    {
        x[i] = rand() / MAX * 2 - 1;
    }

    double min = x[0];
    int index = 0;
    for (int i = 1; i < SIZE; i++)
    {
        if (x[i] < m)
        {
            min = x[i];
            index = i;
        }
    }

    std::cout << "min:" << min
              << " at index:" << index
              << '\n';

    return 0;
}
```

Dynamic arrays (in the heap)

With pointers you can allocate memory for an array (on the heap) with **variable size**. Try this:

```
int s;
std::cin >> s;

int * x = new int[s]; // s can be a variable!!!

// Do something with array that x points to.
//
// (Technically speaking x points to the first value
// of the array in the heap.)

delete [] x;
```

Basically the command

```
      new int[s]
```

tells the heap:

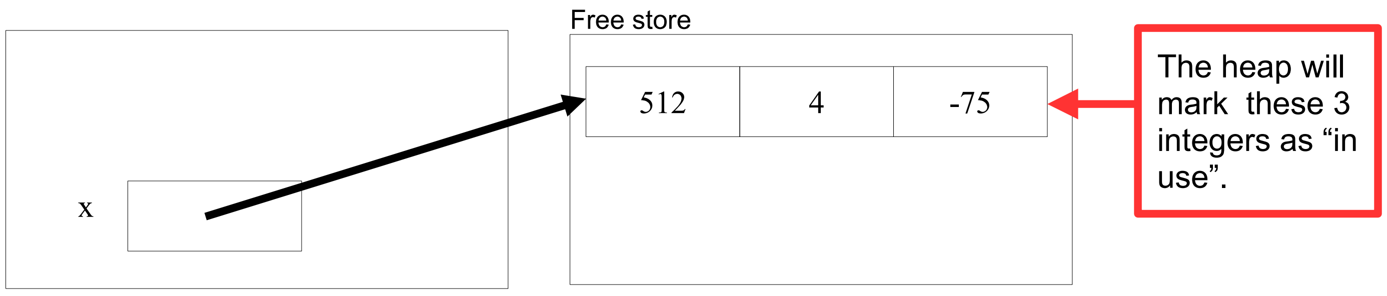
> "hey heap … gimme a chunk of an `int` array of `s` values"

Note that `s` can be a variable!

Mr Heap then goes through its memory and looks for an unused **contiguous** chunk of memory for an array of `s` `int` values, marks this chunk as "in use" (i.e. not free) and **returns the address of the *first* integer value of this array**. Therefore, after calling

```
int * x = new int[s];
```

`x` points to (i.e., has the address of) the **first** `int` value of an array of `s` `int` values. Again, this array of values is in the heap and the number of `int` values allocated is exactly `s`. I will usually say `x` points to the array although technically speaking `x` has the address of the first `int` value of this array, If the value for s is 3, then the memory model looks like this (I just put random integers into the array):



When you do not need the array that `x` points to, you deallocate the memory of this array by doing this:

```
delete [] x;
```

This will release the whole array of `int` values at address `x` back to the heap so that the heap can reuse this memory for another memory allocation.

A **big mistake** is to do this:

```
delete x;
```

which will **only release one value** (i.e. the single value that `x` points to), not the whole array, back to the heap. The problem is that the compiler will not tell you that this is a mistake! This will result in a **memory leak**. This means that if you do

```
int * x = new int[10];
delete x;
```

you are going to have a memory leak of 9 `int`s. For a 32-bit machine, that's a loss of 9x4=36 bytes. The right thing to do is

```
int * x = new int[10];
delete [] x;
```

So remember that in general,

- If you use `new` for a single value, then you should do `delete`.
- If you do `new []` for a whole array, you must do `delete []`.

Here’s an example:

```
int * x;

x = new int;       // ask for one int value ...
delete x;          // ... give back one

x = new int[10];   // ask for 10 int values ...
delete [] x;       // ... give back 10

x = new int[42];   // ask for 42 int values ...
delete [] x;       // ... give back 42

x = new int;       // ask for one int value ...
delete x;          // ... give back one
```

As in the case of dynamic memory management, it’s also a good idea to set a pointer to NULL after deallocation:

```
int * x;

x = new int;       // ask for one int value ...
delete x;          // ... give back one
x = NULL;

x = new int[10];   // ask for 10 int values ...
delete [] x;       // ... give back 10
x = NULL;
```

Exercise. Run this … and see the program crash ...

```
int * x;
while (1)
{
    x = new int[5000];
}
```

Exercise. Run this … and see the program crash ...

```
int * x;
while (1)
{
    x = new int[5000];
    delete x;
}
```

Exercise. Run this … and see what happens:

```
int * x;
while (1)
{
    x = new int[1000];
    delete [] x;
}
```

Stop after a couple of minutes.

Exercise. Fixit time ...

```
int * w;
int * x = new int;
int * y = new int[100];
int * z;
delete w;
delete [] x;
delete y;
delete [] z;
```

After

```
int * x = new int[s];
```

your `x` points to the first value in the array. How do you get to the i-th value in this array? It's just

```
x[i]
```

Remember? Recall that the address of the `i`-th value in this array is

```
x + i
```

Therefore, dereferencing, the value at the address is

```
*(x + i)
```

Recall that `x[i]` is just a shorthand for `*(x + i)`:

`x[i]` is the same as `*(x + i)`

Most programmers use `x[i]` since we want to think of `x` as an array (although technically it's a pointer.) The above explanation was already presented in the previous section. Why am I repeating? BECAUSE IT'S IMPORTANT!!!

Try this:

```
int size;
std::cin >> size;
double * x = new double[size];
for (int i = 0; i < size; i++)
{
    std::cin >> x[i];
}
double product = 1.0;
for (int i = 0; i < size; i++)
{
    product *= x[i];
    std::cout << product << '\n';
}
delete [] x;
```

Exercise. What's wrong with this:

```
int size;
std::cin >> size;
double * x = new double[size];
for (int i = 0; i < size; i++)
{
    std::cin >> x[i];
}
double product = 1.0;
for (int i = 0; i < size; i++)
{
    product *= x[i];
    std::cout << product << '\n';
}
delete x;
```

Exercise. Write a program that continually prompts the user for a value for integer variable `i`, allocate an array of integers of size `i` for pointer `p`, put `i` random integer values into the array, sort the array using bubblesort, print all the values `p` points to, deallocate the memory. Terminate the program when the user enters a 0 for `i`.

Exercise. Write a program that

- Prompts the user for s.
- Creates a dynamic array (on the heap) and declare pointer p to point to an array of s integer values.
- Sets all values in the array to 1.
- Sets all values at index 0, 2, 4, … and toggle the value between 0 and 1, i.e., if the value is 1, set it to 0 and if it’s 0, set it to 1.
- Sets all values at index 0, 3, 6, … and toggle the value between 0 and 1, i.e., if the value is 1, set it to 0 and if it’s 0, set it to 1.
- Sets all values at index 0, 4, 8, … and toggle the value between 0 and 1, i.e., if the value is 1, set it to 0 and if it’s 0, set it to 1.
- Etc.
- At the end of this process, prints the index values where the value is 1.
- Deallocates the array.

Comparing pointers and arrays

There are three concepts here:

- static arrays,
- dynamic arrays in the heap, and
- pointers.

You have to be clear about the three because there are similarities and differences. Most of the comparisons below are already presented earlier. However, because of the similarities and differences, it's best I put everything here so that you can study all the facts in one place.

A pointer can be used to access any value in your computer if you have the address of that value (and if you have access rights). In particular, if a pointer points to the first value of an array (static or dynamic array), using pointer arithmetic, you can access value of the array.

In terms of syntax, there are many similarities between a pointer and an array (static of not). This is deliberate – the designers of C++ made it so.

Static and dynamic arrays

SIMILARITY #1: Array
Both static arrays and dynamic arrays are arrays (i.e., linear collection) of values, not just one value.

SIMILARITY #2: Homogeneity
Both types of arrays are homogeneous, i.e., all the values in an array have the same type.

ASIDE: Once you know more about pointers, even this can be blurred. For instance it's possible to store integers, doubles, and characters in an integer array. But I won't go into this.

SIMILARITY #3: Bracket operator.
The i-th value of a static array and the i-value of a dynamic array can both be accessed using the bracket operator.

```
char x[10] = {'a', 'b', 'c'};
char * y = x;
int size = 10;
char * z = new char[size];
//
// index-2 value of x is x[2] or y[2]
// index-2 value of array z is pointing to is z[2]
```

Remember that `z[2]` is a shorthand for `*(z + 2)`.

SIMILARITY #4: Bracket operator and out of bound access
After you declare declare a static array or allocate memory for a dynamic array (in the heap) with a size s, the array has s values starting with index 0 and ending with index s – 1. HOWEVER, it's possible to access values **OUTSIDE** the array – remember what I said about pointers being able to access any value in your computer (as long as you have the address and the access rights). For instance is x is a static array of size 10 or a pointer to a dynamic array of size 10 in the heap, then `x[i]` is just the value at location `x + i`. You can have `x[42]`, `x[-5]`, etc.!!! You can access the value at address `x + 42` and `x + (-5)` as long as you have the access rights!!!

ASIDE: I've said before that both data and code resides in your computer's memory. This means that a pointer can potentially access **CODE**. If you can change data, then you can change code in your code!

Now for the differences …

DIFFERENCE #1: Memory location.
The most obvious difference is the location of the values of the array. In the case of

```
const int size = 10;
char x[size];
```

the memory allocation is local (i.e. in the current execution block). When you exit the block, the memory use for this array is reclaimed automatically. Therefore static arrays are automatic variables. Dynamic arrays created in the heap using `new[]`, i.e.

```
int size = 10;
char * y = new char[size];
```

lives in the heap, not in the local scope. (Of course the pointer `x` is local). The values of this array is not reclaimed automatically; `x` is local so the pointer value of `x` is reclaimed. You must deallocate the memory of this array yourself (when you're done using the array) by doing this:

```
delete [] y;
```

DIFFERENCE #2: Size.
The second difference is that the size of the array must be a constant for static array:

```
const int size = 10;
char x[size];       // size must be constant
```

However the size used in creating an array in the heap can be any variable expression:

```
int size = 10;
char * y = new char[size];
...
delete [] y;
```

DIFFERENCE 3. Initialization.
An array can be initialized. For instance

```
const int size = 10;
char x[size] = {'a', 'b', 'c'};
```

There is no corresponding syntax for the case for a dynamic array on the heap.

```
int size = 10;
char * y = new char[size] {'a', 'b', 'c'}; // WHAT?!?
delete [] y;
```

You have to perform assignment:

```
int size = 10;
char * y = new char[size];
y[0] = 'a';
y[1] = 'b';
y[2] = 'c';
delete [] y;
```

Static arrays and pointers

SIMILARITY 1. Array name is like a pointer.
You can do this:

```
const int SIZE = 10;
char x[SIZE];
char * y = x; // Remember that x is &x[0]
```

The reason (as stated above) is because in the above `x` is the same as `&x[0]`. So the above make `x` like a pointer (syntactically speaking).

DIFFERENCE 1: Array name is constant pointer, not just pointer.
However static arrays are not exactly pointers. If x and y are pointers of the same type (with values), then you can assign between them:

```
x = y;
```

But try this:

```
char x[10];
char * y;
char * z = new char[10];
char w[10];

y = x; // OK
x = z; // NOPE!!!
```

So what's `x` if `x` is an array since it's not a pointer? You can view `x` as a constant pointer. In other words, you can think of `x` as a pointer but the address of `x` cannot be changed. So for this code fragment:

```
char x[10];
char * y;

y = x; // OK
```

If you want to make `y` as close to `x` as possible, you can do this:

```
char x[10];
char * const y = x; // OK
```

In that case the address value in `y` cannot be changed.

DIFFERENCE 2: `sizeof()`.
Another difference is in the behavior of the `sizeof()` function. Try this:

```
char x[10];
char * y = x;

std::cout << sizeof(x) << '\n';
std::cout << sizeof(y) << '\n';
```

The `sizeof()` on an array returns the size of memory used for the array of values. The `sizeof()` on a pointer returns the size of memory used for the pointer itself, i.e., the memory used for the address of the first value of the array, not the array.

Array and pointer parameters

I've already talked about pointers and arrays. Let's see what happens when they are used as parameters of functions.

When you make a function call like this:

```
void f(int x)
{}

int main()
{
    int y = 42;
    f(y);           // give value of y to x of
                    // function f()
    return 0;
}
```

you can pretty much understand the data passed from `main()` to `f()` by running this code segment:

```
int y = 42;
int x = y;   // give value of y to x
```

(This is pass-by-value.)

However there is one case where this does not work. The following that passes an array to a function does work:

```
void f(int x[])
{}

int main()
{
    int y[100] = {1, 2, 3};
    f(y);
    return 0;
}
```

You might be tempted to run this code segment:

```
int y[100] = {1, 2, 3};
int x[] = y;
```

You ***will*** get a error.

However **this works**:

```
int y[100] = {1, 2, 3};
int * x = y;
```

So here's the thing you need to know:

An array parameter is like a pointer

Now the question is this: Is the pointer parameter a ***constant*** pointer? Let's try an experiment:

```
void f(int x[])
{
    int * y = new int[10];
    x = y; // can parameter x be changed?
}

int main()
{
    int y[100] = {1, 2, 3};
    f(y);
    return 0;
}
```

It turns out that the above does work. This means that the **`x` is not a constant pointer**. This means that

```
void f(int x[])
{}
```

is the same as

```
void f(int * x)
{}
```

You can verify this piece of information using the following trick to get your compiler to tell you the truth. Compile this program:

```
void f(int x[])
{
    abc; // a deliberate error
}

int main()
{
    return 0;
}
```

Your compiler will give you an error. For g++, here's the error:

```
c.cpp: In function 'void f(int*)':
c.cpp:3:3: error: 'abc' was not declared in this
scope
```

AHA!!! The function header of `f()` is really

```
void f(int *)
```

and ***not***

```
void f(int [])
```

So here's the extremely important principle to remember:

An array parameter is actually a non-constant pointer

Default values

There's another thing that we should investigate: What about default values? Let's try this:

```
void f(int x[] = {1, 2, 3})
{}

int main()
{
    f();
    return 0;
}
```

You will get an error. Of course you will. You already know that pointers cannot be initialized with an array. In other words the following is invalid:

```
int main()
{
    int * x = {1, 2, 3};
    return 0;
}
```

You can try other cases (with constantness):

```
const int * a = {1, 2, 3};
int * const b = {1, 2, 3};
const int * const c = {1, 2, 3};
```

None of them will work.

In general the only default value you should use is NULL:

```
void f(int x[] = NULL)
{}
```

There ***is*** one case where a pointer can be initialized to point to values and only for certain compilers. Therefore you shouldn't use the following in real code:

```
const char * x = "hello world";
```

Again. Don't use this in real code. It's better to just be simple, like this:

```
char x[] = "hello world";
```

Functions with array parameters: where to start

This is an old example:

```
int sum(int x[])
{
    int s = 0;
    for (int i = 0; i < 10; ++i)
    {
        s += x[i];
    }
    return s;
}
```

This adds up the values x[0], …, x[9]. Great. But this one is even better because it allows to specify how many values to add:

```
int sum(int x[], int size)
{
    int s = 0;
    for (int i = 0; i < size; ++i)
    {
        s += x[i];
    }
    return s;
}
```

You can use it like this:

```
int a[] = {1, 3, 5, 7, 2, 4, 6, 8};
std::cout << sum(a, 8) << '\n';
```

which will add up all the 8 values in the array. Of course if I want to sum up only the first 4 numbers, I would do this:

```
int a[] = {1, 3, 5, 7, 2, 4, 6, 8};
std::cout << sum(a, 8) << '\n';
std::cout << sum(a, 4) << '\n';
```

But what if I want to compute `x[4] + x[5] + x[6]`? In other words, I want to specify the starting index of the summation. The function starts with index position 0. Recall that in the notes on functions and arrays, I have to rewrite the above function so that it accepts a starting index value, like this:

```
int sum(int x[], int start, int size)
{
    int s = 0;
    for (int i = start; i < size; ++i)
    {
        s += x[i];
    }
    return s;
}
```

Actually it's **not necessary at all (if you know pointers)!**

You see I can do this:

```
#include <iostream>

int sum(int x[], int size)
{
    int s = 0;
    for (int i = 0; i < size; i++)
    {
        s += x[i];
    }
    return s;
}


int main()
{
    int a[] = {1,3,5,7,2,4,6,8};
    std::cout << sum(a, 8) << '\n';
    std::cout << sum(a, 4) << '\n';
    std::cout << sum(a + 4, 3) << '\n';

    return 0;
}
```

to compute `x[4] + x[5] + x[6]`. In this case, the value for size is the number of terms that is added.

Neat right? This means that when writing functions for arrays, you **never have to specify an option for a starting index value!!!** (For languages that do not support pointers and pointer arithmetic, the alternative is almost always slower.)

Exercise. What is the output of this program:

```
#include <iostream>

int sum(int x[], int size)
{
    int s = 0;
    for (int i = 0; i < size; i++)
    {
        s += x[i];
    }
    return s;
}

double avg(int x[], size)
{
    return double(sum(x, size)) / size;
}
```

```
int main()
{
    int a[] = {1, 3, 5, 7, 2, 4, 6, 8};
    std::cout << avg(a + 4, 3) << '\n';
    return 0;
}
```

Exercise. Rewrite the following program so that the prototype of the max function looks like this:

```
int max(int x[], int size);
```

Modify the test code in `main()` so that the same result is produced.

```
#include <iostream>

int max(int x[], int start, int end)
{
    int m = x[start];
    for (int i = start + 1; i <= end; i++)
    {
        if (m < x[i]) m = x[i];
    }
    return m;
}

int main()
{
    int a[] = {1, 3, 5, 7, 2, 4, 6, 8};
    std::cout << max(a, 0, 2) << '\n';
    std::cout << max(a, 4, 6) << '\n';
    std::cout << max(a, 6, 6) << '\n';

    return 0;
}
```

Functions with array parameters: where to start and where to end

Let's continue with the example from the previous section:

```
int sum(int x[], int size)
{
    int s = 0;
    for (int i = 0; i < size; ++i)
    {
        s += x[i];
    }
    return s;
}


int main()
{
    int a[] = {1,3,5,7,2,4,6,8};
    std::cout << sum(a, 8) << '\n';
    std::cout << sum(a, 4) << '\n';
    std::cout << sum(a + 4, 3) << '\n';

    return 0;
}
```

Again, in function sum, you want to think of x as a pointer to an array of values. When you call

```
... sum(a + 4, 3) ...
```

You are basically telling the function to start at the value which is 4 integers away from the address a (i.e., the 5th integer in the array).

I can rewrite the `sum` function like this so that it's really clear that the array parameter is really a pointer parameter:

```
int sum(int * start_pointer, int size)
{
    int s = 0;
    for (int i = 0; i < size; ++i)
    {
        s += start_pointer[i];
    }
    return s;
}
```

Since you're using a starting pointer address to specify where to find the first value, why not specify the pointer address of the last value? In that case your code becomes **more uniform** like this:

```
int sum(int * start_pointer, int * end_pointer)
{
    int s = 0;
    for (int * p = start_pointer;
         p <= end_pointer; ++p)
    {
        s += *p;
    }
    return s;
}
```

You do have to know for instance that ++p on a pointer will move the pointer p to the next integer (and I've already talked about it.)

Now, it's actually more common to specifying the ending address value to be the value that is just **one step outside** the array of values to be processed:

```
int sum(int * start_pointer, int * end_pointer)
{
    int s = 0;
    for (int * p = start_pointer;
         p < end_pointer; ++p)
    {
        s += *p;
    }
    return s;
}
```

Simplifying the parameter names a little, I get …

```
int sum(int * start, int * end)
{
    int s = 0;
    for (int * p = start; p < end; ++p)
    {
        s += *p;
    }
    return s;
}
```

(Instead of `start` and `end`, it's also common to name the pointers `begin` and `end`.)

In this form, `main()` would have to look like this:

```
int main()
{
    int a[] = {1, 3, 5, 7, 2, 4, 6, 8};
    std::cout << sum(a, a + 8) << '\n';
    std::cout << sum(a, a + 4) << '\n';
    std::cout << sum(a + 4, a + 7) << '\n';

    return 0;
}
```

Let me repeat: the general practice is that `end` points to one value beyond what you want to process. So the values processed are

`*start`, `*(start + 1)`, `*(start + 2)`, …, `*(end - 1)`

Note that no values are process if `start` equals `end`.

Exercise. Complete and test this function that does the obvious:

```
void println(int * start, int * end);
```

(It's clear when you should stop.) Here's the format of the output:

```
{5, 1, 2, 4}
```

A newline should be printed. When `start` and `end` have the same value, the output looks like this:

```
{}
```

Exercise. Complete and test this function that does the obvious:

```
int max(int * start, int * end);
```

(It's clear when you should stop.) You maybe assume at least one value is processed, i.e., assume `start` <= `end - 1`.

Exercise. Complete and test this function:

```
int * max(int * start, int * end);
```

that returns the address of the maximum value in `*start`, ..., `*(end - 1)` if `start <= end - 1`. Otherwise `NULL` is returned. You maybe assume at least one value is processed, i.e., assume `start` <= `end - 1`.

Exercise. Complete and test this function:

```
int * linearsearch(int * start, int * end, int target);
```

that returns the address of the first `target` (left-to-right) that occurs in the array of values `*start`, ..., `*(end - 1)`. If `target` does not occur in the array, `NULL` is returned.

Exercise. Complete and test this very important function:

```
void fill(int * start, int * end, int value);
```

that copies `value` to the values at `start`, `start + 1`, etc.

Exercise. Complete and test this very important function:

```
void copy(int * start0, int * start1, int * end1);
```

that copies the value at `start1` to the value at location `start0`, etc. (It's clear when you should stop.)

Exercise. Complete and test this very important function:

```
void resize(int ** start, int * size, int newsize);
```

Initially, `*start` describes an array of `size` integers in the heap. When we return from the above function call, `*start` contains the address of an array of `newsize` integers in the heap and it contains (as many as possible) the values from the original array. For instance if you have

```
int * p = NULL, size = 0;
```

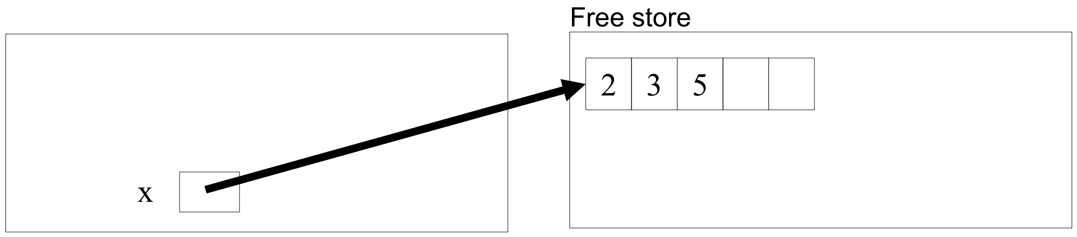
after calling

```
resize(&p, &size, 5);
```

`p` points to an array of 5 integers in the heap and `size` is changed to 5. Suppose we continue the above with

```
p[0] = 2;
p[1] = 3;
p[2] = 5;
```

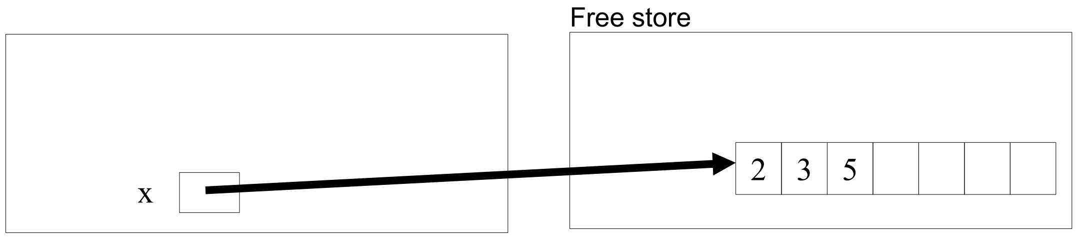
Here's the memory model:



Next if we call

```
resize(&p, &size, 7);
```

`p` points to an array of 7 integers in the heap and `size` is changed to 7. Furthermore the first three values of the new array `p` points to are 2, 3, and 5. The memory model at this point looks like this:



(Note: `newsize` can be smaller than `size`.)

Difference of pointers: binary search

Note that we have been using addition operator pointers:
Here's continue with the example from the previous section:

```
int a[] = {1,3,5,7,2,4,6,8};
int * p = &a[1];
int * q = &a[4];
```

That's addition of a pointer value and a constant integer.

Go ahead and try this:

```
int a[] = {1,3,5,7,2,4,6,8};
int * p = &a[1];
int * q = p + 3;
std::cout << p + q << '\n';
```

Read the error message carefully.

You **cannot add pointers**.

However try this:

```
int a[] = {1,3,5,7,2,4,6,8};
int * p = &a[1];
int * q = &a[4];
std::cout << p - q << '\n';
```

It works. This tells you (of course) that the integer `q` points to is 3 integers away from the integer that `p` points.

Why is this important?

Recall from the previous section, when it comes to functions that computes on a section of an array, i.e. a subarray, you want to call the function with two pointers. For instance:

```
int sum(int * start, int * end)
{
    // returns sum of *start, ..., *(end - 1)
    ...
}
```

Sometimes you want to know how many terms of the array is involved. This is

`end - start`

This is the size of the subarray that the function works with.

Another reason why the difference of pointers is important is because of binary search. In the scenario when you are performing binary search on a subarray where you specify the subarray using index values, the function looks like this:

```
int binarysearch(int x[], int start, int end,
                 int target)
{
    int left = start, right = end - 1;
    while (left <= right)
    {
        int mid = (left + right) / 2;
        ...
    }
}
```

See chapter on 1D arrays of CISS240.

This version has a problem. Remember that an integer is usually (at least for now) is made up of 32 bits. So the range of an integer value is -2^31, …, 2^31 – 1. So when your `left` and `right` are both large, for instance they are both very close to 2^31 – 1, then `left + right` will overflow and become a negative integer!!!

The way to fix this is to compute the `mid` index like this:

```
int binarysearch(int x[], int start, int end,
                 int target)
{
    int left = start, right = end - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        ...
    }
}
```

This version will not cause an integer overflow. So from now on, you must use this version of binary search.

In the above, `right` is the largest index of the subarray the current iteration of your binary search is working on.

You can also do this where `right` is **one index beyond** the subarray your binary search is working on:

```
int binarysearch(int x[], start, int end,
                 int target)
{
    int left = start, right = end;
    while (left < right)
    {
        int mid = left + (right - left) / 2;
        ...
    }
}
```

This is now the index value version of binary search that you must use:

```
int binarysearch(int x[], int start, int end,
                 int target)
{
    int left = start, right = end;
    while (left < right)
    {
        int mid = (left + right) / 2;
        if (x[mid] == target)
        {
            return mid;
        }
        else if (x[mid] > target)
        {
            right = mid; //  note: no "- 1"
        }
        else
        {
            left = mid + 1;
        }
    }
}
```

What about a pointer version of binary search? Using the CISS240 version as a guide, when you try this:

```
int * binarysearch(int * start, int * end,
                   int target)
{
    int * left = start;
    int * right = end;
    while (left < right)
    {
        int * mid = (left + right) / 2;
        ...
    }
}
```

you'll see that it does not even compile … because you **cannot add pointers**.

```
...
        int * mid = (left + right) / 2;
...
```

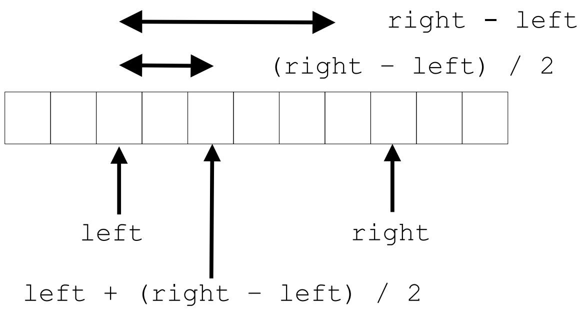
The solution is to compute the middle pointer in a different way:

```
int * binarysearch(int * start, int * end,
                   int target)
{
    int * left = start;
    int * right = end;
    while (left < right)
    {
        int * mid = left + (right - left) / 2;
        ...
    }
}
```

The quantity

```
(right - left) / 2
```

is half the number of integer from `left` to just before `right`. It's slightly less than half if the number of such values is odd. Add this to `left` would then of course give you the address of the integer in the middle of integers **`*left,…,*(right - 1)`**.



This middle pointer computation is just like the middle index computation.

From now on you should use the midpoint calculation

mid = left + (right - left) / 2

for both middle index and middle pointer computation for binary search.

By the way, in general, there's no reason to compute the difference of two addresses other than the address of two values in the same array. One last thing: if you want to store the difference of two pointers, you need to know that there's a type for pointer differences:

```
std::ptrdiff_t d = p - q;
```

You need to `#include <cstddef>` if you want to use `std::ptrdiff_t`.

Exercise. Implement the pointer version of bubblesort and binary search:

```
void bubblesort(int * start, int * end);
int * binarysearch(int * start, * end, int target);
```

In your `main()`, get integers `n`, `start`, `end`, `target` from the user and then do the following:

- Allocate memory for a dynamic integer array of size `n` and put random integers (from 0..9) into the array.
- Print the array.
- Bubblesort `x[start]`, …, `x[end-1]`.
- Print the array.
- Perform binary search for `target` in `x[start]`, …, `x[end-1]`

and print the address and the value at that address.

Exercise. Let x be an integer array of size n. Let y be an array of integer pointers where y[0] points to x[0], y[1] points to x[1], etc. Write a bubblesort on y so that y[0], ..., y[n - 1] each points to a unique values in x and *y[0] <= *y[1], *y[1] <= *y[2], etc. The values in x are not moved. For instance say x is {1, 5, 0} (when n = 3). Initially y[0] points to x[0], y[1] points to x[1] and y[2] points to x[2]. After your bubblesort is done, y[0] points to x[2], y[1] points to x[0] and y[2] points to x[1].

Stack using a dynamic array

A **stack** is like an array except that it supports the following operations:

- You can put a value into the stack. This is called **push**. The value you just put into the stack is called the **top** of the stack.
- You can remove a value from the stack. The value that is removed is always the last one that was put into the stack. This is called **pop**.
- You have access to the number of values in the stack. I'm going to call this the **size** of the stack.
- You can also look at the value that is to be popped. I'll call this **peek**. Peeking at a stack returns a copy of the value, it does not remove the top.

Think of a stack of plates at a buffet restaurant: When you add a plate to the stack, that new plate goes on top. When you take a plate, you take the one on top (well … usually).

You can implement a stack using a dynamic array. The top of the stack is the value at the last index of the array. If you push a value onto the stack, you will need to enlarge the array by 1. If you pop the stack, you will need to replace the array with another one that is 1 size smaller; of course you need to copy the original values to the new stack (except for the top.)

Pushing the stack is not new, you have actually seen this in an earlier exercise on the `array_append() function.`

Stacks are very important and are used in many areas of CS including AI, language processing, etc. A stack should be thought of as a memory device. You can ask the stack "Hey stack … what was the **last thing** I put into you?"

Exercise. Write a program that implements a stack of integers using dynamic arrays. Test it thoroughly! You can test your stack implementation with the following:

```
int main()
{
    int * stack = NULL;
    int stack_size = 0;

    println(stack, stack_size);    // Get {}

    push(&stack, &stack_size, 5);
    println(stack, stack_size);    // Get {5}
    push(&stack, &stack_size, 3);
    println(stack, stack_size);     // Get {5, 3}

    push(&stack, &stack_size, 9);
    println(stack, stack_size);     // Get {5, 3, 9}

    int x;
    x = peek(stack, stack_size);
    std::cout << x << std::endl;    // Get 9
    println(stack, stack_size);     // Get {5, 3, 9}

    pop(&stack, &stack_size);
    println(stack, stack_size);     // Get {5, 3}

    return 0;
}
```

There are many ways to implement a stack. Later you'll see that a more efficient way of implementing a stack is to use a **singly linked list**, something that uses pointers.

Queue using a dynamic array

A **queue** is like an array except that it supports the following operations:

- You can put a value into a queue. This is called **enqueue**. The value you just put into the queue is called the **back** of the queue.
- You can remove a value from the queue. The value that is removed is always the value in the queue that has been in the queue the longest. This is called the **front** of the queue. This operation is called **dequeue**.
- You have access to the number of values in the queue. I'm going to call this the **size** of the queue.
- You can also look at the value of the front and back of the queue without removing these values. These operations are sometimes called **front** and **back**.

You think of a queue as a line of people in front of a ticket booth: When a person joins a queue, he/she joins it at the back. When the booth is ready to serve the next customer, the person to leave the queue to go to the booth is the one in front of the queue.

You can implement a queue using a dynamic array. The front of the queue is the value at index 0 of the array. The back of the queue is the value at the last index of the array. If you enqueue a value into the queue, you will need to enlarge the array by 1. If you dequeue the stack, you will need to replace the array with another one that is 1 size smaller.

Like stacks, queues are very important and are used in many areas of CS including AI, language processing, etc. A queue should be thought of as a memory device. You can ask the queue "Hey queue … what was the **earliest thing** that you can think of?"

Exercise. Write a program that implements a queue of integers using dynamic arrays. Test it thoroughly! You can test your queue implementation with the following:

```
int main()
{
    int * queue = NULL;
    int queue_size = 0;

    println(queue, queue_size);     // Get {}

    enqueue(&queue, &queue_size, 5);
    println(queue, queue_size);     // Get {5}

    enqueue(&queue, &queue_size, 3);
    println(queue, queue_size);    // Get {5, 3}

    enqueue(&queue, &queue_size, 9);
    println(queue, queue_size);    // Get {5, 3, 9}

    int x;
    x = front(queue);
    std::cout << x << std::endl;   // Get 5
    x = back(queue, queue_size);
    std::cout << x << std::endl;   // Get 9
    println(queue, queue_size);    // Get {5, 3, 9}

    dequeue(&queue, &queue_size);
    println(queue, queue_size);    // Get {3, 9}

    return 0;
}
```

There are many ways to implement a queue. Later you'll see that a more efficient way of implementing a queue is to use a **doubly linked list**, something that uses pointers (and is very similar to singly linked list.)

Set using a dynamic array

A **set** is a container of values where you don't care about the number of times a value occurs and you don't really care about the index position of a value. It supports the following operations:

- You can **add** a value to a set. You can put it anywhere you like in your dynamic array. For instance you can put is at the last index position, like the `array_append()` function. Since you don't really care how many times a value occurs, if you try to add a value into the set and the set already contains that value, you simply do NOT add the value into the set, i.e., a set does not contain duplicates.
- You can **remove** a value from a set. If you attempt to remove a value not in the set, then nothing happens – the set is unchanged.
- You can check if a value is **in** a set. I'll call this `has_member()` function.
- You have access to the number of values in the set. I'll call this **size**.

You can think of a set also as a memory device except that you don't care about when you started remembering a value.

Exercise. Write a program that implements a set of integers using dynamic arrays. Test it thoroughly! You can test your set implementation with the following:

```
int main()
{
    int * set = NULL;
    int set_size = 0;

    println(set, set_size);         // Get {}

    add(&set, &set_size, 5);
    println(set, set_size);         // Get {5}

    add(&stack, &set_size, 3);
    println(set, set_size);         // Get {5, 3}

    add(&set, &set_size, 5);
    println(set, set_size);         // Get {5, 3}

    add(&set, &set_size, 3);
    println(set, set_size);         // Get {5, 3}

    bool b;
    b = has_member(set, set_size, 5);
    std::cout << b << std::endl;    // Get 1
    println(set, set_size);         // Get {5, 3}

    b = has_member(set, set_size, 3);
    std::cout << b << std::endl;    // Get 1
    println(set, set_size);         // Get {5, 3}


    b = has_member(set, set_size, 2);
    std::cout << b << std::endl;    // Get 0
    println(set, set_size);         // Get {5, 3}

    remove(&set, &set_size, 3);
    println(set, set_size);         // Get {5}

    remove(&set, &set_size, 0);
    println(set, set_size);         // Get {5}

    return 0;
}
```

There are many ways to implement a set. The above is actually not efficient. Later you will see that better ways to implement a set includes using a balanced tree and a hashtable.

Array of pointers and multi-dimensional dynamic arrays

I want you to think deeply about this

If I do "int x[10];" I have a fixed size 1D array x of size 10.

and

If I do "int * x = new int[10];" x points to a 1D array of size 10.

Exercise. Take a deep breath … Draw the memory model of the following code fragment. Study the code very carefully.

```
int ** p;
p = new int*[3];
// Draw a memory model here

for (int i = 0; i < 3; i++)
{
    p[i] = new int[5];
    // Update memory model
}

p[1][1] = 42;
p[2][3] = 24;
// Update memory model

for (int i = 0; i < 3; i++)
{
    delete [] p[i];
    // Update memory model
}

// Update memory model
```

So what exactly is `p` pointing to (so to speak?)

Exercise. Take a deep breath … Draw the memory model of the following code fragment. Read the code very carefully.

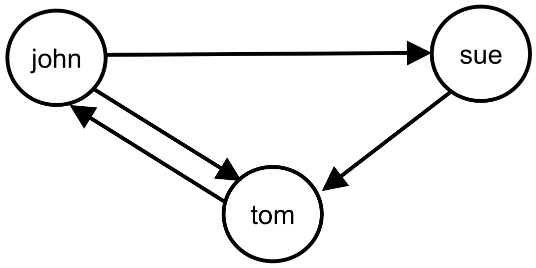
```
int ** p;
p = new int*[3];
// Draw a memory model here

for (int i = 0; i < 3; i++)
{
    p[i] = new int[i + 2];
    // Draw a memory model
}
```

```
p[1][1] = 42;
p[2][3] = 24;
// Update memory model

for (int i = 0; i < 3; i++)
{
    delete [] p[i];
    // Update memory model
}

// Update memory model
```

See the power of pointers yet?

Graphs and social media

One of the most important structures in CS is the graph. A **directed graph** is just a bunch of dots and arrows between the dots. For instance here's a social media graph:



say this represents "follows the twitters of". For instance John follows Sue's twitters. In general, nowadays social media type graphs are extremely huge and complex – and they keep growing. Trying to get information out of social media graphs without computers is impossible. (Obviously information gathered from social media graphs of all types would be extremely valuable.)

One way to represent the above graph is to use a table or spreadsheet like this:

	john	sue	tom
john	0	1	1
sue	0	0	1
tom	1	0	0

It's pretty obvious how you should interpret the table. Like I said, in the real world, graphs can be extremely huge. So we need to find a way to put the table into a program and answer interesting questions such as:

- Is there a sequence of arrows joining X to Y?

Clearly the table looks like a 2D array!

0	1	1
0	0	1
1	0	0

This matrix (a matrix is just a 2D array of numbers) that described a directed graph is called the **adjacency matrix** of the graph.

Of course we need to remember that john has a row and column index of 0, sue has a row and column index of 1, and tom has a row and column index of 2. So we can have an array of strings. Don't forget that strings are character arrays (containing the null character '\0'). So the index-name associate can be described by a 2D array of characters:

'j'	'o'	'h'	'n'	'\0'					
's'	'u'	'e'	'\0'						
't'	'o'	'm'	'\0'						

In the above, I'm assuming that there are 3 persons in the social media graph and their names are at most 9 characters long (don't forget that you need one character space for the null character.)

Therefore all in all, the space requirement is like this:

```
char name[3][10];
int matrix[3][3];
```

a total of 9 integers and 30 characters. In general, if there are k people, then the space requirement is k x k integers and k x 10 characters (if names have a maximum string length of 9.)

Exercise. Write a program that will allow you to manage the above social media graph, assuming that there will always be only 3 persons and their names are at most 9 characters long. Your program of course should be able to display the names and the social media matrix. You should allow the user to change the names and the matrix (but always assuming that there are 3 persons in the social media graph.)

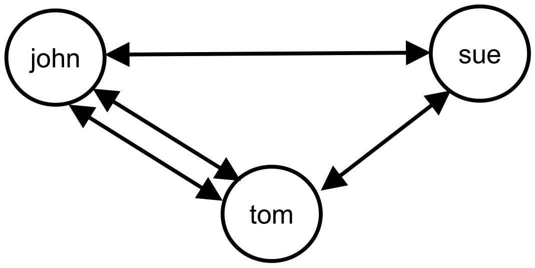
Exercise. Now improve on your above program by allowing the user to add and delete person(s) from the social media graph.

Exercise. Now improve on your above program so that if a person's name has 6 characters, then you use 7 characters for this person. Furthermore, your program should allow names of length greater than 9. This will save some memory. For instance for tom, you only need 4 characters for him.

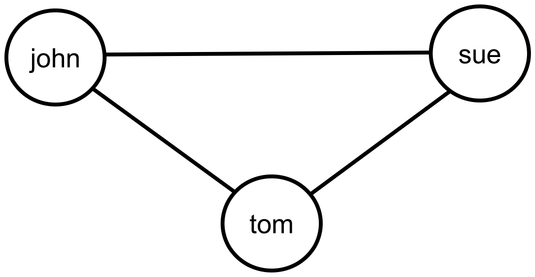
Exercise. Now, how can we save memory for the social media matrix? You see the problem is: If there are k people, then the matrix is made up of k x k integers. Think about this: How many people do you think have a facebook account? It's approximately 1.2 billion. If you use the above 2D array to represent the social media graph in FB, then the matrix needs 1.2 billion x 1.2 billion integers!!! That's 1.44 x 10^{18} integers!!! [HINT: Most people are not friends of everyone in the world. In other words, in a row of a social media adjacency matrix, most of the values are actually 0s.]

Exercise. Path connectivity problem. Now add the following to your program: The user to enter two names, X and Y, and the program prints a sequence of names going X to Y, following the arrows. If possible, print the sequence that is the shortest. [This is not easy! You will probably need **everything** in the notes up to this point!]

By the way, in the real world, many graphs are “bidirectional”:



In that case, the directed graph diagram is usually drawn like this:



where each line represents the fact that the relationship between two dots go both ways. A graph where the lines do not have arrowheads is called an **undirected graph**.

Both directed and undirected graphs appear everywhere in CS and engineering. As long as you have relationships between data, you will have graphs. And they have been around for a very long time – a family tree is a directed graph. But it’s also used in modern scenarios such as social media graphs, a graph of nodes in a computer network, food web, etc.

Solutions

Exercise. Implement the pointer version of bubblesort and binary search:
`int * bubblesort(int * start, * end)`
`int * binarysearch(int * start, * end, int target)`
In your `main()`, get integers `n`, `start`, `end`, `target` from the user and then do the following:

- Allocate memory for a dynamic integer array of size `n`. and put random integers (from 0..9) into the array.
- Print the array.
- Bubblesort `x[start]`, ..., `x[end-1]`.
- Print the array.
- Perform binary search for `target` in `x[start]`, ..., `x[end-1]` and print the address and the value at that address.

Solution is below. Don't memorize the code. Understand the motivation and intention of bubblesort and binarysearch, implement them using index values, then convert to pointers.

```
#include <iostream>

// print *start, .., *(end - 1)
void println(int * start, int * end) // or begin and end
{
    for (int * p = start; p < end; ++p)
    {
        std::cout << (*p) << ' ';
    }
    std::cout << '\n';
}

// bubblesort *start, .., *(end - 1)
void bubblesort(int * start, int * end)
{
    for (int * q = end - 2; q >= start; --q)
    {
        for (int * p = start; p <= q; ++p)
        {
            if (*p > *(p + 1))
            {
                int t = *p; *p = *(p + 1); *(p + 1) = t;
            }
        }
    }
}

// binarysearch for target in *start, .., *(end - 1)
int * binarysearch(int * start, int * end, int target)
{
    while (start < end)
    {
        int * mid = start + (end - start) / 2;
        if (*mid == target)
        {
            return mid;
        }
        else if (*mid > target)
        {
            end = mid;
        }
        else
        {
            start = mid + 1;
        }
    }
    return NULL;
}

int main()
{
    int n, start, end, target;
    std::cin >> n >> start >> end >> target;
    int * x = new int[n];
    for (int i = 0; i < n; ++i)
    {
        x[i] = rand() % 10;
    }
    print(&x[0], &x[n]);

    bubblesort(&x[start], &x[end]);
    print(&x[0], &x[n]);

    int * p = binarysearch(&x[start], &x[end], target);
    std::cout << p << ' '
              << (p != NULL ? *p : -1) << '\n';

    return 0;
}
```