Up to now, variables have only held a single value. Single valued variables are said to be *scalar*.

Now we introduce one of the two kinds of *compound variables*, that is variables which can hold multiple values.

An *array* is designed to hold *homogeneous* values. That is, every value in an array is of the same type. For example—

```
int a;
int c[10];
double x[22];
string names[30];
```

While `a` is a conventional (scalar) integer, we would say

`c` is an array of 10 integers.

`x` is an array of 22 doubles

`names` is an array of 30 strings

The size, or *dimensionality*, of the array must be declared so that compiler knows how much storage to set aside.

Element Reference

We can access individual elements of an array via an *index*, like so—

```
c[3] = 13;
x[20] = x[17] - x[3];
cout << names[0];
```

Notice that the "first" element of the array actually has index `0`. The implication is that there is no such thing as `c[10]`, `x[22]` or `names[30]`.

The indices of an array of size n run from 0 to n-1.

Initialization of Arrays

Arrays are initialized using an *initialization block*

```
int a = 3;
int fib[10] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
int daysInMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

It is often easier to represent irregular data such as the number of days in a month in a table like this than to write an algorithm to try to generate the data (of course we'll still need an algorithm to handle leap years).

Note that it is not necessary to initialize every element of an array.

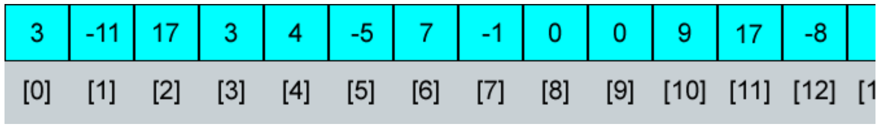
```
int a[10] = {4, -5, 7, -11, 9};
```

initializes `a[0]` through `a[4]` while leaving `a[5]` through `a[9]` uninitialized.

As you might expect, specifying too many elements (overspecifying) is a compile-time error.

Array Storage

When we diagram how an array get's stored in memory, it looks a lot like a stream (in fact we started with the stream drawing).



The elements in an array are stored adjacent to one another much as they are in a stream. There are a couple of important differnces however.

1. A stream contains characters. We can have arrays of any legitimate C++ type at all, the only condition is that every element in a particular array must be of one type.
2. A stream is accessed sequentially, using a pointer that starts at the beginning and moves through the stream, element by element.
3. An array is accessed directly, simply by specifying the *index* of the desired element (this is actually what is meant by the term *random access*).

Basically, streams are like video tape where you have to move through the tape to get to the scene you want while arrays are more like DVD's, allowing you to cut directly to the desired element.

In the figure above we see an array of `int`s. The element at index `6` is `7`, while that at `11` is `17`.

Be careful with arrays of `int`s that you don't mix up the *index* or *position* of an element (which is always an `int` for any kind of array) with the *value* stored *in* the element.

Formal Syntax

The formal syntax for an array declaration is

Array declaration :

Form:
Type Identifier [size];
Type Identifier [size] = {Initialization-list};

Example:

```
int marks[30];
char letterGrades[5] = {'F','D','C','B','A'};
```

Interpretation: space is created for `marks`, an unitialized array of 30 integers, and for `letterGrades`, an array of 5 chars with `letterGrades[0]` initialized to `'F'`, `letterGrades[1]` initialized to `'D'` and so on.

Let's see arrays in action

RUN average.cpp

```
int main(){
    float numbers[10];
    float sum = 0;
    int i;          // array index

// Fill up the array
    for (i = 0; i < 10; i++)
        numbers[i] = fabs(sin(2*3.14159*i/10));

    cout << "There are 10 numbers as follows: " << endl;

    for (i = 0; i < 10; i++) {
        cout << numbers[i] << ' ';
        sum += numbers[i];
    }

    cout << "\n\nTheir average is " << sum/10 << endl;
    return 0;
}
```

One significant problem with this example is the repeated use of the literal 10. It is much better to make this a constant, as follows:

RUN average_better.cpp

```
const int SIZE = 10;

int main(){
    float numbers[SIZE];
    float sum = 0;
    int i;          // array index

// Fill up the array
    for (i = 0; i < SIZE; i++)
        numbers[i] = fabs(sin(2*3.14159*i/SIZE));

    cout << "There are " << SIZE << " numbers as follows: " << endl;

    for (i = 0; i < SIZE; i++) {
        cout << numbers[i] << ' ';
        sum += numbers[i];
```

```
    }

    cout << "\n\nTheir average is " << sum/SIZE << endl;
    return 0;
}
```

This example shows the real advantages of constants over literals. Now, to change our array size we only have to make a single change. Using literals (as in the first example) we would have to make lots of them (everywhere SIZE appears, in fact).

Of course, the moment one sees the above, it makes you want to do this—

RUN average_illegal.cpp

```
int main(){
    int size;
    float sum = 0;
    int i;          // array index

    cout << "Please input the size of the array: ";
    cin >> size;

    float numbers[size];

// Fill up the array
    for (i = 0; i < size; i++)
        numbers[i] = fabs(sin(2*3.14159*i/size));

    cout << "There are " << size << " numbers as follows: " << endl;

    for (i = 0; i < size; i++) {
        cout << numbers[i] << ' ';
        sum += numbers[i];
    }

    cout << "\n\nTheir average is " << sum/size << endl;
    return 0;
}
```

Unfortunately, you can't. The compiler will complain. So will the Teaching Machine. *Array size cannot be variable.* It must be known at the compile time (because the compiler has to set space aside in memory for the array).

Passing Arrays to Functions

Passing arrays *by value* is problematic because passing by value always implies making a copy and copying large arrays

1. is slow
2. doubles the memory usage

Thus we would like to pass arrays as we do objects, that is by reference.

Actually, they are technically passed by address because arrays are an original feature of C and C didn't have pass-by-reference. The difference at your stage is not worth worrying about except to point out that—

arrays are not objects. They predate objects.

Let's move our averaging operation into a function where it should be.

average_function.cpp

```
float average(float data[], int size){
    float sum = 0.;
    for (int i = 0; i < size; i++)
        sum += data[i];
    return sum/size;
}
```

The *prototype* for the function

    float average(float data[], int size)

contains *two* parameters. `data` is declared to be an `array of floats`. The second `int` parameter, `size`, is passed in to say how large the array is. This is done because the data array is not an object and does not know its own size.

The compiler knows the size of an array when it is created, but that is not the same thing.

To see why not consider a program where we want the average of two different arrays:

RUN

average_function.cpp

```
const int SIZE_S = 10;
const int SIZE_C = 5;

int main(){
    float sines[SIZE_S];
    float cosines[SIZE_C];
    int i;    // used to index through arrays


// Fill up the arrays
    for (i = 0; i < SIZE_S; i++)
        sines[i] = fabs(sin(2*3.14159*i/SIZE_S));
```

```
    for (i = 0; i < SIZE_C; i++)
        cosines[i] = fabs(cos(2*3.14159*i/SIZE_C));

    cout << "The average of the sines is ";
    cout << average(sines, SIZE_S) << endl;
    cout << "& the average of the cosines is " ;
    cout << average(cosines, SIZE_C) << endl;

    return 0;
}
```

Although this example is somewhat artificial, it serves to make the point

Two different arrays `sines` and `cosines` have been created each with its own size. Since the sizes `SIZE_S` and `SIZE_C` are constants, the compiler knows how much space to allocate in memory for each of the two arrays.

Averaging, however is carried out *at run time*. The `average` function gets called twice, once for each array. It *has* to be able to accommodate both sizes, thus the `size` argument is variable.

Searching and Sorting

Arrays are commonly used to store large amounts of homogenous data, for example telephone directories or corporate sales figures,

Thus a common task is to search an array for a particular piece of data or a data with a particular characteristic.

Problem: build a function to find the *position* of the largest piece of data in an array of doubles.

Analysis:

Clearly a loop is involved since I'll have to search the entire array—and `for` loops are a good match for arrays

concentrate on the body of the loop, assume I know the position of the largest piece of data so far

We'll hold that in a variable called `position,` so we should get something like

```
for i = 0, i < arraysize
    if data[i] > data[position]
        position = i
```

Now all we have to do is get it started—how about we set position to 0?

Header: `int getLargest(double data[], int size)`

Algorithm:

```
set position to 0
for i = 1, i < size     // Notice adjustment to get started
if data[i] > data[position]
    position = i
```

Question: What happens if there are two or more largest pieces of data? Where will `position` end up? Is this sensible?

Final Comment: in working with problems like this we have to be careful to distinguish between the *position in the array* and the *data at the position*. That is, between the *location of the data* and the *actual data*. Here we compare the data, but we track the position.

Median Problem

Problem: Find the median grade for a class whose data is stored in a file.

Analysis:

1. We're going to need an array to hold the grades. How big should it be?

If we want to handle different classes, we won't know the class size until we read the data in from the file (which is when we *run* the program). But we have to size the array when we *build* the program.

One approach is to create an array large enough to handle any reasonable class size and then only use part of it. We'll also have to watch out for a class bigger than we planned for.

So we'll pick a `MAX_CLASS_SIZE` and declare a `float grades[MAX_CLASS_SIZE]`

2. Clearly we're going to have to read the data in from a file. That should probably be done by a function.

We should give the function the array and tell it about MAX_CLASS_SIZE and then *let it tell us the actual size*

3. In order to find the median we're going to have to *sort* the file into either ascending or descending order. Another funtion. It will need the grades array and the actual class size.

4. compute the median.

Major Data:

```
int MAX_CLASS_SIZE
float grades[MAX_CLASS_SIZE]
```

Headers:

```
int getGrades(float gradeData[], int maxSize) // return actual size
void sort(float gradeData[], int size)
```

Algorithm:

```
Read values into grades[0..N-1]
Sort grades in increasing order
if N is even
    median = (vals[N/2-1] + vals[N/2]) / 2
else
    median = vals[N/2]
```

Final Comment:

Why N/2 in the last line?

If N is odd, for example 5

- There are 5 elements in the array
- They are *numbered* from 0 to 4
- The middle one is no 2 (0, 1, **2**, 3, 4)
- N/2 is of course 2

This page last updated on Wednesday, March 24, 2004