ARRAYS

Consider the following problem:

"write a program to read three integers and calculate their sum"

There are two approaches, each with its own advantages.

Approach 1:

```
int sum, num1, num2, num3;

cin >> num1 >> num2 >> num3;
sum = num1 + num2 + num3;
```

Upon completion of these lines, the three values input are available for further use – stored in the variables `num1`, `num2`, `num3`.

Approach 2:

```
int sum, num, i;

sum = 0;
for (i=0;i<3;i++)
{
    cin >> num;
    sum += num;
}
```

Here, only the last value read is kept in the variable `num`.

Now suppose we want to find the sum of 50 values?

The second form could easily be modified, just by changing the 3 to 50 as in

```
for (i=0;i<50;i++)
```

But suppose we would prefer the availability of the 50 values for further work.

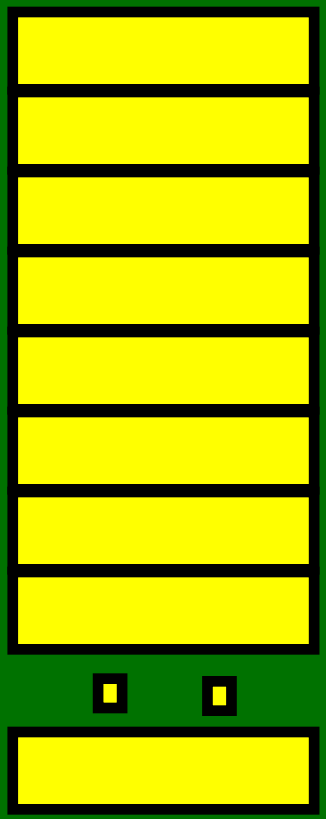
We need a way of extending the idea of variables like `num1`, `num2`, `num3` to 50 or even more values.

C++ provides such a feature, allowing us to declare a collection of variables of the same type which can be accessed by using a name and a number.

It's the **array**.

Consider a sequence of adjacent memory locations, each capable of holding a value of one particular type.

C



size depends on type

Give all the memory one name.

Each individual entry is referenced using an **index**.

The first entry is called `C[0]`, the next `C[1]`, and so on.

Each such entry is called an **element** of the array.

The index is often called the **subscript** (like the mathematical concept of c_0, c_1, . . .).

The important concept peculiar to C++ is that the first element in all arrays is that with index 0, and not one.

As with any variable in C++, we must declare an array before we can use it.

But this time, not only do we have to specify what the type is, and what its name is, but how many elements we want.

C++ wants to know the memory required.

Note that an array is a collection of things of the same type

Note that an array is a collection of things of the same type – we can't have `C[0]` being a `float`, `C[1]` an `int`, `C[2]` a `char`.

Back to our example.

Suppose we want to sum 500 integers, but still have the values available for further processing.

```
int sum, i, num[500];

 sum = 0;
 for (i=0;i<500;i++)
 {
        cin >> num[i];
        sum += num[i];
 }
```

Note the use of an expression `i` for the required element of the array.

The term `num[i]` can be used to represent the i^{th} element of the array (numbered from 0) just like any other single variable.

Each element of an array is a variable.

The subscript can be any integer expression.

Let's now look at the syntax of array declaration.

The declaration

type *name*[*size*];

specifies that the variable named can be used with subscript values from 0 to *size*-1.

`size` must be a constant integer expression.

Here are some examples:

```
const int noRakeOff = 25;
float MonthlySales[12];
double RakeOff[noRakeOff];
char Letters[10];
```

Again, a reminder that subscripts start at zero.

Operations are performed on an array on a one-by-one basis – as simple variables.

```
if (MonthlySales[5] < MonthlySales[6])
    RakeOff[5] = (MonthlySales[0]+
        MonthlySales[11])/2.;
```

Other methods of referencing arrays will be seen later.

```
const int howmany = 500;
float Sum, i, Ave;
float Num[howmany], Difference[howmany];

/* initialise variables */
for (i=0;i<howmany;i++)
   cin >> Num[i];

Sum = 0;
for (i=0;i<howmany;i++)
   Sum += Num[i];
Ave = Sum/howmany;
for (i=0;i<howmany;i++)
   Difference[i] = Num[i]-Ave;
```

At some stage in the program design process, the programmer must decide how much memory is reasonable to allocate for an array.

This is based on the number of values that are expected.

If we don't know that number when we are writing the program, we have to provide sufficient for any (expected) problem.

```
const int MAX = 500;
  int howmany;
  float Sum, i, Ave;
  float Num[MAX], Difference[MAX];

  cin >> howmany;
  for (i=0;i<howmany;i++)
       cin >> Num[i];

  Sum = 0;
  for (i=0;i<howmany;i++)
       Sum += Num[i];
  Ave = Sum/howmany;
  for (i=0;i<howmany;i++)
       Difference[i] = Num[i]-Ave;
```

So how big do we specify?

Not too many but not too few.

GREAT!!!!

Too much wastes memory.

Not enough limits the size of data sets.

What happens if we use an index which is beyond our declared limits?

e.g.

```
float B[20];


B[100] = 6.0;
B[20] = 5.0;
```

NO ERROR.

What happens if we use an index which is beyond our declared limits?

e.g.

```
float B[20];


B[100] = 6.0;
B[20] = 5.0;
```

NO ERROR. NO WARNING.

It may not even be detected !!!

We're using memory not set aside for B.

Recall the recent example

```
const int MAX = 500;
int howmany;
float Sum, i, Ave;
float Num[MAX], Difference[MAX];
```

Note how the constant was used to set limits on the arrays declared.

It would then be a simple matter to increase that limit if larger arrays are needed.

It is good programming practice to ensure that the values of subscripts on arrays never exceed the limit specified in the declaration.

This means that warnings can be given to users about exceeding array limits.

Let's see an example.

```
const int maxsize = 100;
int Sum, HowMany, i, Num[maxsize];

cin >> HowMany;
if (HowMany > maxsize)
{
     cout << "Too many values to be read."
          << " Only the first "
          << maxsize << " will be read.\n";
     HowMany = maxsize;
}
Sum = 0;
for (i=0;i<HowMany;i++)
     cin >> Num[i];
```

How do we initialise an array?

```
type varname[size] = {val0, val1, ...};
```

The number of values must be less than or equal to the size.

If it is short, other entries will be zeroed.

The expressions in the braces must be of an acceptable type.

```
int DaysInMonth[12] = {31,28,31,30,31,
          30,31,31,30,31,30,31};

float Heights[5] = {12.3,5.7,4.6,
           18.4,2.0};

char MonthLetter[12] = {'J','F','M','A',
           'M','J','J','A','S','O','N','D'};

int LettersInMonthNames[] =
{7,7,5,5,3,4,4,6,9,7,8,8};
```

12

Arrays can also be declared `const`.

Character arrays can be declared (and used) just as any other array of basic type.

```
char varname[size];
```

We saw earlier the existence of character string constants like `"characters"` where the characters between " are stored one to a byte, with a null character NUL (or zero byte) at the end to indicate the last character.

At that time we had no way of naming a string constant – we needed a string variable.

Now we have it – the character array.

We can use character arrays one character at a time, just like other arrays.

But we can also manipulate C-strings within such arrays.

A character array can be initialised by using a string as its initial value.

```
char Month[8] = "January";
const char ProgrammersName[]
          = "James Bond";
```

The second example would be allocated 13 bytes.

How about input/output of strings?

We've seen that the iostream library can handle the basic data type `char`.

```
char FirstInitial;

cin >> FirstInitial;
cout << FirstInitial << endl;
```

iostreams can also handle C-strings in character arrays:

```
char FirstInitial, Surname[20];

cin >> FirstInitial >> Surname;
cout << FirstInitial << ". " << Surname
   << endl;
```

But there are problems.

iostreams uses whitespace as an input terminator.

iostreams uses whitespace as an input terminator. This means that we cannot get a space into a character array via cin string input.

```
char FullName[50];
cin >> FullName;
```

would read

```
James Bond<ret>
```

into `FullName`

iostreams uses whitespace as an input terminator. This means that we cannot get a space into a character array via cin string input.

```
char FullName[50];
cin >> FullName;
```

would read

```
James Bond<ret>
```

into `FullName` as ”`James`" with a null character at the end of 5 characters.

The quotes aren't part of the contents of the array.

If, in this case, the input consisted of more than 49 characters without intervening whitespace, the array would overflow by using elements not declared.

So, can we read spaces into a character array? Can we protect ourselves from such overflows?

We'll see soon.