Arrays and Functions

Arrays
An array is nothing more than a list of data. All the data are stored in a structure under a single name, and all values must be of the same type. An array has a specified number of slots, and those slots are numbered consecutively starting from 0. The number of a slot is called its index. Thus, the indexes of the values in an array range from 0 up to the length of the array minus one.

Here is rundown of the nuances of working with arrays in C++

Declaring an array in C++
The syntax is
   `datatype nameOfArray[numberofSlots];`
For example, to declare an array of 5 ints called numbers:
   `int numbers[5];`

It is possible to use an initializer list to declare and initialize the array in one statement:
   `int numbers[] = {11, 13, 9, 7, 4};`

Types of Arrays
Arrays can be either static or dynamic.

A static array is created as shown above. A static array is fixed in size at the time you create it.

C++ also supports the idea of dynamic arrays. That topic gets us into the field of dynamic memory allocation, and pointers. Pointers are beyond the scope of an introductory course. Suffice it to say that the name of the array is actually a reference to a memory address. In other words, if you were to print out the name of the array to cout, you’d learn the number of the location in memory where the array starts.

Example: Using Arrays in C++
Write a C++ application that inputs 5 numbers from the user and then outputs them to the screen.

Solution:

```cpp
#include <iostream>
using namespace std;

int main()
{
    int numbers[5];
    int max;
    for (int i = 0; i < 5; i++)
    {
        cout << “Enter #” << i << “: “;
        cin >> numbers[i];
    }
    for (int j = 0; j < 5; j++)
    {
        cout << numbers[j] << endl;
    }
    return 0;
}
## Functions

Functions help us break a long, complicated problem into smaller components. Each of those smaller components is called a function and is really just a mini-program that takes inputs, does some work, and may return some output. These notes help us understand the various ways functions can be declared, used, and called.

| Functions | In math, a function is something that generates a new value from some other value. 
For example, 
\[ f(x) = 5x + 2 \] 
In C++, a function is a group of statements that either 
- generate a new value or values 
- performs some useful task 
given some other value or values |
| --- | --- |
| Advantages of functions | We’ve already seen functions 
- every program has a function called main 
Could stick everything in main 
- works ok for small programs but not for big ones. 
Why use functions? 
1. Break program into manageable pieces 
2. Focus on just one part and perfect that part 
3. Simultaneously work on different parts 
4. Can call functions multiple times. |
| Two types of functions | Standard vs. user-defined 
- some return values 
- others do not (return type = void) 
Standard functions are usually stored in libraries that you must include in your program using 
#include. |
| Standard functions | Here are a few examples of standard functions: 
\[
\begin{align*}
\text{abs}(x) & \quad \text{//comes from <cstdlib>} \\
\text{ceil}(x) & \quad \text{//comes from <cmath>} \\
\text{cos}(x) & \\
\text{exp}(x) & \\
\text{fabs}(x) & \\
\text{floor}(x) & \\
\text{pow}(x,y) & \\
\text{tolower}(x) & \quad \text{//comes from <cctype>} \\
\text{toupper}(x) & 
\end{align*}
\] |
| User-defined | These are functions that you define |
### Functions

You specify
- name of function
- number of parameters
- data type for each parameter
- return type

```
return-type function-name(list-of-parameters)
{
    statements-to-execute;
}
```

### Example

Write a function that returns the absolute value of a number. Call it absval.

```
int absval(int number)
{
    if (number < 0)
    {
        return -number;
    }
    else
    {
        return number;
    }
}
```

Note: “return” is the keyword that C++ functions use to report the value generated or computed by the function to the rest of the program.

### More User-defined Functions

Write a function called `raise` that computes the value of a number raised to a power.

```
double raise(double base, int exp)
{
    double result = base;
    for (int i = 2; i <= exp; i++)
    {
        result *= base;
    }
    return result;
}
```

### How Do You Call a Function?

If the function returns a value:

```
variable-that-receives-value =
    name-of-function(list of parameter values);
```

Example:
```
double x = 3.2;
int a = -4;
```
### Void vs. value-returning functions

Some functions return values; others do not.

If a function does not return a value
- it’s return type should be “void” (not int, double, etc.)
- it doesn’t need a “return” statement
- call it just by placing its name on a line (it shouldn’t be used in a situation that requires a value to be returned.

Example:
```c
void say_hi()
{
    cout << “Hi.”;
}

int main()
{
    ...
    say_hi();
}
```

### Where do you define a function?

A function’s definition must appear before the statement in which it is called.

That means that, if raise has not already been declared, this is illegal:

```c
int main()
{
    double x_cubed = raise(3.2,3);
    ...
}
```

To ensure that functions are declared before they are used, use a function prototype.

### Function Prototypes

A function prototype is a function header listed near the top of a program file in which it is called.

Example:
```c
double larger(double x, double y); // prototyp
```

```c
int main()
{
    double x, y;
    cin >> x >> y;
    cout << “The larger number is “ << larger(x,y) << endl;
```
```cpp
return 0;
}

double larger(double x, double y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

**Example**

Write a function that returns whether or not to draft a fantasy football player. The decision should be based on the following:

For quarterbacks, the player should be drafted if his average yardage per game exceeds 250 yards, his touchdown total for the year is greater than 12, and his interceptions is less than 10.

For running backs, the player should be drafted if his average yardage per game exceeds 80 yards, his touchdown total for the year is greater than 10, and his number of fumbles for the year is less than 4.

For wide receivers, the player should be drafted if his average yardage per game exceeds 70 yards, his touchdown total for the year is greater than 8, and his number of catches for the year is greater than 50.

Also write a main function that calls this function to determine whether or not to draft the player.

**Scope of identifiers: an introduction**

Variables may be declared...

1. inside blocks such as functions or loops
2. outside of any block. Example:

```cpp
#include <iostream>
using namespace std;
int x;
int main()
{
    // x can be accessed here
}
```

Where can variables be accessed?

- that’s called the variable’s **scope**.

**Local vs. Global**

Local identifier:
- declared inside a block
- not accessible outside the block

Global identifier
- declared outside every function definition
- accessible by a function if
  1. the identifier is declared before the function definition
2. the function name is different from the identifier
3. all parameter names are different from the identifier
4. all local identifiers have names different from the identifier

These same rules apply both to identifiers that identify functions and those that identify variables.

| Special rules regarding blocks within blocks | A **block** is the code within a pair of braces
|                                              | { this is a block
|                                              | }
|                                              | An identifier declared within a block is accessible
|                                              | - within the block from the point at which it is declared until the end of the block
|                                              | - by those blocks that are nested within the block if the nested block doesn’t have an
|                                              | identifier with the same name.

| Example involving scope | What follows is an example that tests whether we understand the scope of identifiers (i.e. where an identifier can be accessed). Where can each of the identifiers that appear in this example be accessed?
|                        | #include <iostream>
|                        | using namespace std;
|                        | const double rate = 10.50;
|                        | int z;
|                        | double t;
|                        | void one(int x, char y);
|                        | void two(int a, int b, char x);
|                        | void three(int one, double y, int z); |
|                        | int main()
|                        | { int num, first;
|                        | double x, y, z;
|                        | char name, last,
|                        | . . .
|                        | return 0;
|                        | }
|                        | void one(int x, char y)
|                        | { . . .
|                        | }
|                        | int w;
|                        | void two(int a, int b, char x)
|                        | { int count;
```plaintext
void three(int one, double y, int z)
{
    char ch;
    int a;
    
    int x;
    char a;
}
```

### Functions can be overloaded

Multiple functions in a program can have the same name as long as they differ in the number type order of their parameters.

The full name of a function is actually the function name plus the list of parameters.

You cannot distinguish functions by return type, only by their parameter lists.

Giving different functions the same name but different parameter lists is called **overloading functions**.

#### Example: function overloading

- `int largest(int num1, int num2);`
- `int largest(int num1, int num2, int num3);`
- `double raise(double base, int exp);`
- `double raise(double base, double exp);`

Both of these were acceptable examples of function overloading.

This is not good:
- `int largest(int num1, int num2);`
- `double largest(int num1, int num2);`

It’s not good because they differ only by return type.

### Pass-by-value vs. pass-by-reference

Normally, when you pass an actual parameter to a function in a function call, this is what happens:

1. computer makes a copy of the value of the actual parameter
2. the copy is passed to the function, which stores it in the formal argument
3. any changes made to the formal argument are temporary – the actual arguments are not affected.

This is called “pass-by-value“ or “call-by-value.”
There is another way to pass parameters to functions – it is called “pass by reference.”
- the “&” symbol in the example above show how you pass something by reference.

We’ll talk about pass-by-reference in a moment. First, though, let’s do an example of pass-by-value

<table>
<thead>
<tr>
<th>Example: pass-by-value</th>
<th>What is output by the following code?</th>
</tr>
</thead>
<tbody>
<tr>
<td>void double_num(int num)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>num *= 2;</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; num;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>int main()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int favorite_number = 7;</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; favorite_number;</td>
<td></td>
</tr>
<tr>
<td>double_num(favorite_number);</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; favorite_number;</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

We see that the value of favorite_number is not changed by the call to double_num.
- because the parameter is passed by value.

<table>
<thead>
<tr>
<th>Pass-by-reference</th>
<th>It is possible to have the function make a change to an argument’s value and have that change be permanent.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>How?</td>
</tr>
<tr>
<td></td>
<td>Put an “&amp;” at the end of the data type name.</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>void double_num(int&amp; num)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>num *= 2;</td>
</tr>
<tr>
<td></td>
<td>cout &lt;&lt; num;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>int main()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>int favorite_number = 7;</td>
</tr>
<tr>
<td></td>
<td>cout &lt;&lt; favorite_number;</td>
</tr>
<tr>
<td></td>
<td>double_num(favorite_number);</td>
</tr>
<tr>
<td></td>
<td>cout &lt;&lt; favorite_number;</td>
</tr>
<tr>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

In this case, the value of favorite_number is actually changed by the function call.
- the & symbol makes all the difference!
| **How does pass-by-reference work?** | “int” is different from “int&”  
- `int` → a number  
- `int&` → the name of a memory location  

When you pass an `int` to a function, you are not giving the function permission to play with the value in the memory space.  

When you pass an `int&`, on the other hand, you are giving the function the keys to the value’s memory location. |
| --- | --- |
| **Another reference parameter example** | Write a program that asks the user to enter a numeric grade and then prints the corresponding letter grade. Use these functions:  
```cpp
void get_score(int& score);  
void print_grade(int score);  
``` |
| **Stream parameters** | Streams should always be passed by reference because the contents of the stream must be changeable by the function.  
This makes sense – with output streams, you are pushing values to output into the stream. With input streams, you are removing values from it. If a function doesn’t have permission to modify the streams, how can the streams be written or read correctly?  
Example:  
```cpp
void show_result(ostream& out1, string desc, double val);  
``` |
| **Functions that return multiple values** | Use value-returning functions if a function returns just 1 value.  
Use void functions with reference parameters if the function must return multiple values. |
| **Example** | Write a function that reads a list of numbers from a stream that ends with -1 and returns the min, max and average of them.  
```cpp
void statistics(istream& in, double& min, double& max,  
double& avg)  
{
    double num, average, count;  
in >> num;  
``` |
```c
min = num;
max = num;
while (num != -1)
{
    count++;
    if (num < min)
    {
        min = num;
    }
    if (num > max)
    {
        max = num;
    }
    sum += num;
in >> num;
}
avg = num/count;
```

**Arrays and Functions**

To declare a function that takes an array parameter, include the [] with the name of the parameter.

To pass an array to a function, do not include the [].

Here’s an example: this function takes an array of int:

```c
void takesAnIntArray(int numbers[], int length) {
}
```

Notice how the array’s length is also passed, because arrays do not know their own length.

Here’s a possible function call:

```c
int numbers[5];
takesAnArray(numbers, 5);
```

Again, no [] with the array when you call the function.

Arrays are always passed by reference!
- This means that, if you change a value in the array in the function, that change becomes permanent!