**Arrays**

1. **Declaring and Referencing Arrays**

An array is declared by giving the name and the size of the array. E.g.,

*int* score [5] = { 0, 1, 2, 3, 4 }

All indexed variables for one array are of the same type, which is called the **base type**.

Even though some compilers allow the use of variables while declaring the size of arrays, we **should not** be using this since many compilers do not allow such operations. Therefore, we can instead use a different kind of array whose size can be determined when the program is run (later)

1. Diagram

   Description automatically generated**Arrays in Memory**

Recall that a *simple variable in memory* is described by 2 pieces of information: an **address** in memory (location of the first byte of that variable) and the **type** of variable, which determines the number of bytes of memory the variable requires.

As for arrays, the location of the indexed variables is always **placed next to one another in memory**. Therefore, the computer will only need to remember the address for a[0], which can be used to calculate the address for a[2] or a[3].

*Note* that if we declare array at indexes that are out of the scope of the array, **other variables at those excess memory locations are modified** (like moreStuff). This leads to serious and hard-to-find bugs for the program.

1. **Initializing Arrays**

Array can be initialized when declared (we can then omit declaring size)  *int* children [] = {2, 12, 1} ;

This is equivalent to defining values for each index after declaration.

Indexes not initialized with a value will be given a value of 0 of the given type, but will NOT be given this value if arrays are declared within functions, including main() function.

1. **C++11 Range-based for-Statement \*\***

This is a new type of for loop that iterates over every element in an array

for (datatype varname : array){

// varname is successively set to each element in the array.

}

*NOTE* that while defining varname, we can change the datatype as we need to for our use. For example, using int x : arr will be pass-by-value and will not change the array, but we can do int& x : arr

We can also use the keyword **auto** to automatically determines the type of element inside array.

1. **Arrays in Functions**

We can use both array indexed variables and the whole array as arguments to functions.

**Indexed Variables as Function Arguments**

Indexed variables work the same way as simple variables, as they are evaluated as the value. Indexed variables can be either call-by-reference or call-by-value

**Entire Array as Function Arguments**

If we want to set up a function having a formal parameter for an entire array, it is neither a call-by-value or call-by-reference parameter; It is a new kind of formal parameter, called the **array parameter**.

The a is int a [ ] , with the square brackets and no index expressions inside. This will then be replaced by the array (by passing its name to the function argument).

Text

Description automatically generated

This array parameter can behave like call-by-reference parameter, which changes the array. However, there are some differences.

Recall that the computer only stores the address of the first index of an array. When an array argument is plugged in for its corresponding formal parameter, all that is plugged in is *the address of the array’s first indexed variable* (pointer to the first index variable). The type of the array argument passed matches the defined type of the array parameter.

🡪 Function only knows the starting point in memory (address of first index) and increment size while traversing array (type). It **does NOT know the array size**, so that’s why we have to pass the size as well!

1. **The const Parameter Modifier**

We know that functions can alter array parameter as well as array arguments passed. However, there are some functions that do not change the array values, so we can tell the compiler that this is not our intention. If we mistakenly change the array values, the compiler will help us give the error.

*void* showTheWorld(***const*** *int* a[ ], *int* sizeOfA)

1. **Functions Returning an Array**

To return an array, we will have to return a pointer that points to an array. (which is array name)

**Multidimensional Array**

A multidimensional array is an array of arrays

To declare a multidimensional array, we just declare the size of the dimensions within declaration

*char* page[30][100];  
*int* matrix[2][3];  
***double* threeDPicture[10][20][30];**

**Passing Multidimensional Arrays into functions**

When a multidimensional array parameter is given in function heading, the size of the first dimension may not given, but **REMAINING DIMENSION SIZES MUST BE GIVEN IN SQUARE BRACKETS**.

*Note* that even if we include the size for the first dimension in brackets, the program will ignore this and treat it like a comment only 🡪 We still need to pass them as other arguments.

The reason for this rule is that since they are "arrays of arrays". For example, the two-dimensional array parameter

*void* displayPage(*const char* p[][100], *int* sizeDimension1)

is a parameter for an array of arrays, the first dimension is really the index of the array and is treated just like an array index for an ordinary, one-dimensional array. The second dimension (and so on) is part of the description of the base type, which are arrays of characters of size 100 at each index 🡪 Have to include in the brackets.