**Arrays as parameters.**

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

|  |  |  |
| --- | --- | --- |
|  | void procedure (int arg[]) |  |

This function accepts a parameter of type "array of int" called *arg*. In order to pass to this function an array declared as:

|  |  |  |
| --- | --- | --- |
|  | int myarray [40]; |  |

it would be enough to write a call like this:

|  |  |  |
| --- | --- | --- |
|  | procedure (myarray); |  |

Here you have a complete example:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // arrays as parameters  #include <iostream>  using namespace std;  void printarray (int arg[], int length) {  for (int n=0; n<length; ++n)  cout << arg[n] << ' ';  cout << '\n';  }  int main ()  {  int firstarray[] = {5, 10, 15};  int secondarray[] = {2, 4, 6, 8, 10};  printarray (firstarray,3);  printarray (secondarray,5);  } | 5 10 15  2 4 6 8 10 | [Edit & Run](https://cplusplus.com/doc/tutorial/arrays/) |

In the code above, the first parameter (int arg[]) accepts any array whose elements are of type *int*, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the *for loop* that prints out the array to know the range to iterate in the array passed, without going out of range.

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

|  |  |  |
| --- | --- | --- |
|  | base\_type[][depth][depth] |  |

For example, a function with a multidimensional array as argument could be:

|  |  |  |
| --- | --- | --- |
|  | void procedure (int myarray[][3][4]) |  |

Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  18  19  20  21  22  23  24  25  26  27  28  29  30 | // C++ Program to display the elements of two  // dimensional array by passing it to a function  #include <iostream>  using namespace std;  // define a function  // pass a 2d array as a parameter  void display(int n[][2])  {  cout << "Displaying Values:" << endl;  for (int i = 0; i < 3; ++i)  {  for (int j = 0; j < 2; ++j)  {  cout << "num[" << i << "][" << j << "]: " << n[i][j] << endl;  }  }  }  int main()  {  // initialize 2d array  int num[3][2] = {  {3, 4},  {9, 5},  {7, 1}  };  // call the function  // pass a 2d array as an argument  display(num);  return 0;  } | Displaying Values:  num[0][0]: 3  num[0][1]: 4  num[1][0]: 9  num[1][1]: 5  num[2][0]: 7  num[2][1]: 1 | [Edit & Run](https://cplusplus.com/doc/tutorial/arrays/) |

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | //Calculating Standard Deviation for 10 numbers  #include <iostream>  #include <cmath>  using namespace std;  double calculateSD(double data[]);  int main()  {  int i;  double data[10];  cout << "Enter 10 elements:" << endl;  for (i = 0; i < 10; ++i)  {  cin >> data[i];  }  cout << endl << "Standard Deviation = " << calculateSD(data);  return 0;  }  double calculateSD(double data[])  {  double sum = 0.0, mean, standardDeviation = 0.0;  int i;  for (i = 0; i < 10; ++i)  {  sum += data[i];  }  mean = sum / 10;  for (i = 0; i < 10; ++i)  {  standardDeviation += pow(data[i] - mean, 2);  }  return sqrt(standardDeviation / 10);  } |

Here is what happens if you go out of range when you want to print out array elements using loops:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | #include “iostream”  using namespace std;  int main()  {  int k[]{1, 2, 3, 4, 5, 6, 7, 8};  for (int i = 0; i < 10; ++i)  {  cout << k[i] << ' ';  }  } | 1 2 3 4 5 6 7 8 -866891109 19922564 | [Edit & Run](https://cplusplus.com/doc/tutorial/arrays/) |

As you can see if we go out of range, it will print us additional undetermined values after valid values in existent indices of the array.