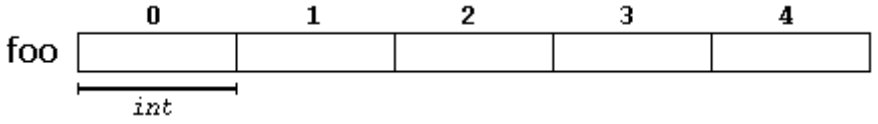
➔ **Arrays**

Theory

- An array is a series of elements of the same type placed in contiguous memory locations
- These elements can be individually referenced by adding an index to a unique identifier.
- That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier).
- Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.
- For example, an array containing 5 integer values of type int called foo could be represented as:



- Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:
    - type name [elements];
    - where type is a valid type (such as int, float...),
    - name is a valid identifier and the elements field (which is always enclosed in square brackets []),
    - specifies the length of the array in terms of the number of elements.
- Therefore, the foo array, with five elements of type int, can be declared as:

```
int foo [5];
```

- NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

➔ **Address Calculation**

- Address = Base_Address + size_of_one_element * index
- Calculate the following addresses:
    - Array of doubles. Base address = 2000, index = 34
    - Array of shorts. Base address = 10504, index = 120
    - 2 dimensional array of ints. 5 rows and 3 columns. Base address = 2000, index = 3, 2
- For the last one, assume row major ordering, that is, the first row (all columns) are stored before the second row. So, to get to the 3rd row, you should have passed 3 full rows (0, 1, and 2). You can use the following formula for a 2D array:

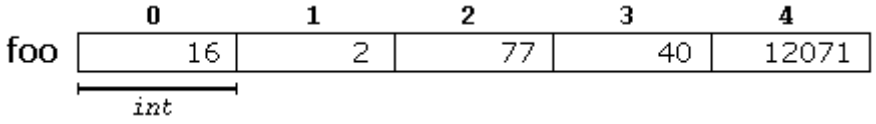
- Address = Base_Address + size_of_one_element*(row_index * Number_of_Columns + column_index)
- **Array of doubles**
  - Size = 8bytes
  - 2000 + 8*34 = 2272
- **Array of shorts**
  - Size = 2bytes
  - 2000 + 2*120 = 2240
- **2D int array**
  - Size = 4bytes
  - 2000 + 4*(3*3+2) = 2044
- An array of longs begins at address 2000, the arr[6] can be found at
  - 2000 + 8*6 = 2048

➔ **Declaring Arrays**

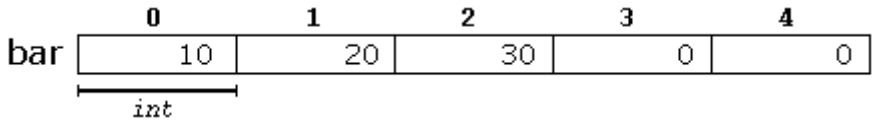
- An array must be declared before it is used. A typical declaration for an array in C++ is:
  - type name [elements];
  - where type is a valid type (such as int, float...),
  - name is a valid identifier
  - The elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.
- Therefore, the foo array, with five elements of type int, can be declared as:
  - int foo[5];

➔ **Initializing arrays**

- By default, regular arrays of local scope (for example, those declared within a function) are left uninitialized.
- This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.
- But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:
  - int foo [5] = { 16, 2, 77, 40, 12071};
- This statement declares an array that can be represented like this:



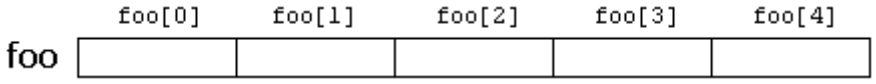
- The number of values between braces {} should not be greater than the number of elements in the array.
- For example, in the example above, foo was declared having 5 elements
- If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:
  - int bar [5] = { 10, 20, 30 };



- When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty [].
- In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:
    - int foo [] = { 16, 2, 77, 40, 12071 };
- After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

➔ **Accessing the values of an array**

- The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type.
- The syntax is: name[index]
- Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



- For example, the following statement stores the value 75 in the third element of foo:
- The following copies the value of the third element of foo to a variable called x:
    - int x = foo[2];
- Therefore, the expression foo[2] is itself a variable of type int.
- The third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2].
- By this same reason, its last element is foo[4].
- Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array.
- At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks:
    - one is to specify the size of arrays when they are declared;
    - and the second one is to specify indices for concrete array elements when they are accessed.

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

- Some other valid operations with arrays:

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

```
// arrays example
```

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

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
 for ( n=0 ; n<5 ; ++n )
 {
  result += foo[n];
 }
 cout << result;
 return 0;
}
```

➔ Reading Values into Arrays and printing out the Values

```
#include <iostream>
using namespace std;
int main()
{
  int i,n,a[100];

    cout<<"Input the number of elements to store in the array :";
    cin>>n;

   for(i=0;i<n;i++)
   {
            cout<<"Element "<<i<<endl;
            cin>>a[i];
    }

  cout<<"The values store into the array are:"<<endl;
  for(i=0;i<n;i++)
   {
            cout<<a[i]<<" ";
           }

 }
```

➔ Sum of Array Elements

```
#include <iostream>
using namespace std;
int main()
```

```
{
  int i,n,a[100];
  Int sum=0;

    cout<<"Input the number of elements to store in the array :";
    cin>>n;

   for(i=0;i<n;i++)
   {
            cout<<"Element "<<i<<endl;
            cin>>a[i];
    }

  cout<<"The sum of values store into the array is:"<<endl;
  for(i=0;i<n;i++)
   {
          sum+=a[i];
   }
    cout<<sum<<endl;
 }
```

➔ **Calling function on Each Element**

```
#include <iostream>
using namespace std;
float area(int);
int main()
{
  int i,n,a[100];
  float circle[100];

    cout<<"Input the number of elements to store in the array :";
    cin>>n;

   for(i=0;i<n;i++)
   {
            cout<<"Element "<<i<<endl;
            cin>>a[i];
    }

  cout<<"Theareas stored into the array is:"<<endl;
  for(i=0;i<n;i++)
   {
          circle[i] = area(a[i]);
   }

  cout<<"The area put into the array are:"<<endl;
  for(i=0;i<n;i++)
```

```
    {
            cout<<circle[i]<<" ";
           }


 }
 float area(int radius)
 {
   return(3.14*radius*radius);
 }
```

➔ **Arrays as parameters**

- 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);
```

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length);

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

```
void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}
```

- 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
- 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.
- **This is pass by reference**