One-Dimensional Arrays

- Demonstrating the need for arrays

  - Suppose that we need to write a program which reads 1000 numbers and prints them in reverse order

```
#include <iostream.h>

void main()
{
 // declare the variables
 int value0;
 int value1;
 int value2;
  ...
 int val999;

 // read the values
 cin >> value0;
 cin >> value1;
 cin >> value2;
  ...
 cin >> value999;

 // print the values
 cout << value999;
 cout << value998;
 cout << value997;
  ...
 cout << value0;
}
```

- Commments

  - Number of statements in the body of *main()*: 3000 !!

    * 1000 declaration statements
    * 1000 *cin* statements
    * 1000 *cout* statements

  - All of the above variables have the same data type

  - All of the above variables have the same name except for an appended number that distinguishes them

- What is an array ?

  - A collection of variables **having the same data type**

  - Each variable **consists of two parts**:

    * The first part is the name of the variable which is the same for all of them
    * The second part is an *index value* enclosed in square brackets

  - Examples

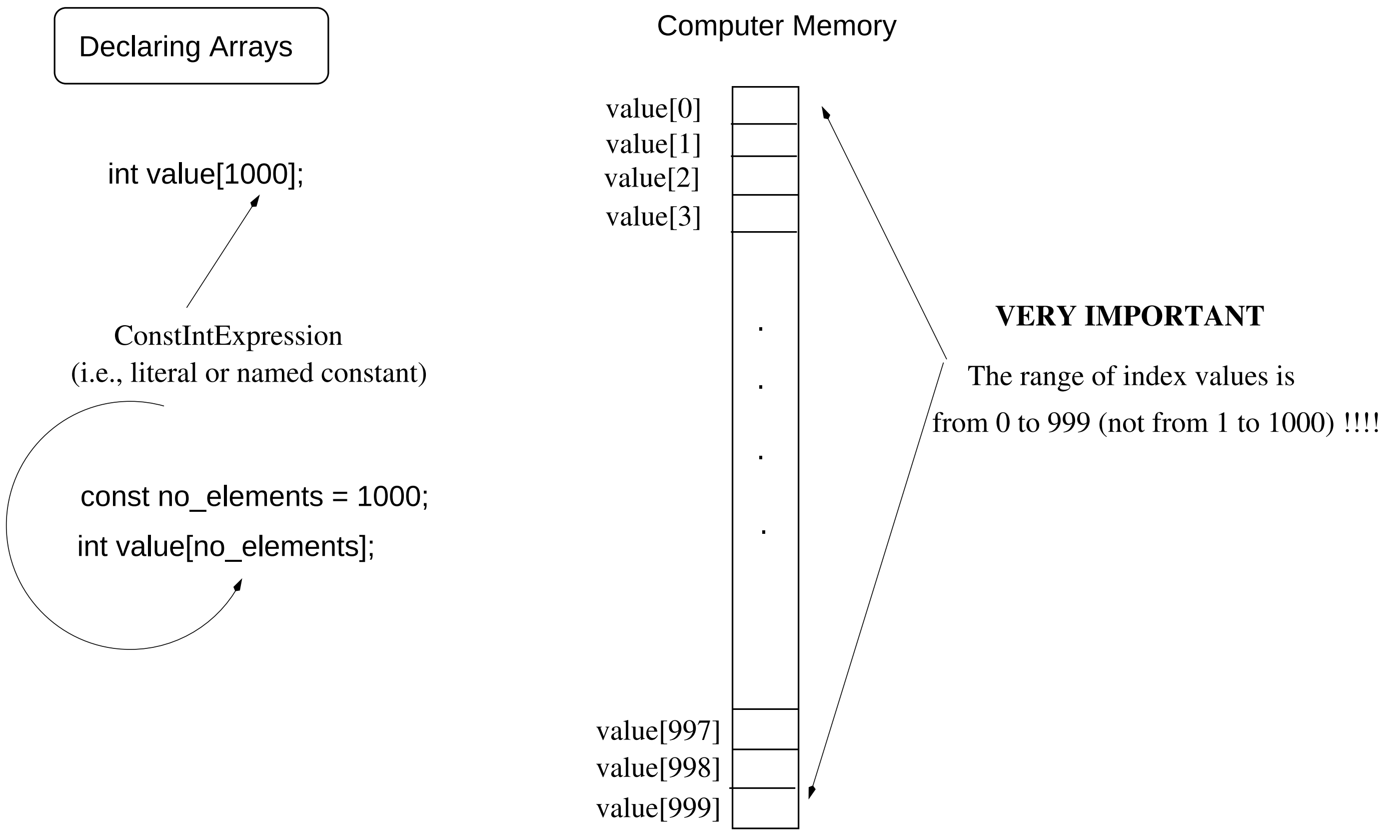
  value[10], value[201], value[999]

  - Because an array is a collection of variables (not just one variable) where each one can be accessed individually, it is called a **structured data type**

- Declaring arrays

  - You reserve space in memory for the number of elements by specifying the type of elements and the actual number of elements you need

```
DataType ArrayName [ ConstIntExpression ]
```



- Accessing individual components

ArrayName [IndexExpression]

  - The *IndexExpression* may be just a constant, or a variable or even an expression

  - The result of *IndexExpression* **must be an integer value**

  - Examples: *value[4]*, *value[2*a+1]* (assuming *a* is an integer)

- Each array element can be treated the exactly the same way as a simple variable

  - *Assign* a value: *value[2]=100;*

  - Read a value into it: *cin >> value[2];*

  - Write its contents: *cout << value[2];*

  - Pass it as a parameter: *y=sqrt[value[2]);*

Without using Arrays

```
#include <iostream.h>

void main()
{
 int value0;
 int value1;
 int value2;
    ......
int value999;

cin >> value0;
cin >> value1;
cin >> value2
    ......
cin >> value999;

cout << value999;
cout << value998;
cout << value997;
     .....
cout << value0;

}
```

Using Arrays

```
#include <iostream.h>

void main()
{
 int value[1000];
 int index;

 for(index=0; index<=999; index++)
   cin >> value[index];

 for(index=999; index >=0; index--)
   cout << value[index];

}
```

allocating 1000 integer variables

or index < 1000

accessing individual components

- Initializing arrays

  - Arrays can be initialized in a declaration

  int value[5] = {10, 20, 12, 13, 33};

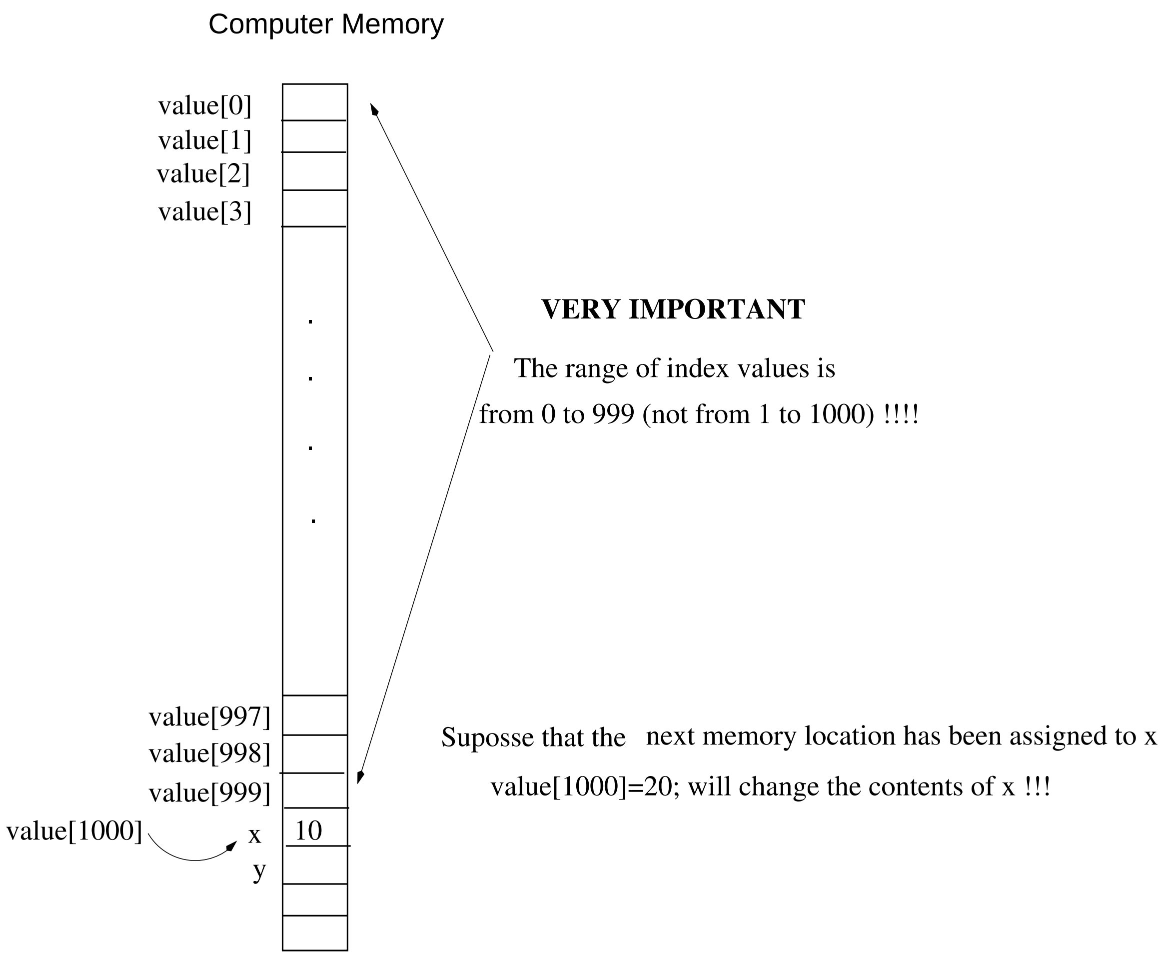
  - Arrays can also be initialized using a loop statement

  for(i=0; i<5; i++)
  cin >> value[i];

- Out-of-bounds arrays

  - Each index value must be between 0 and *size_of_array - 1*

  - **Very important**: C++ does not check for invalid indices !!!



- Common mistake

```
int value[1000];
  ....
for(i=1; i<=1000; i++)
  cin >> value[i];
```

- Manipulating the array as an entire unit (*aggregate operation*)

  - C++ **does not** provide aggregate operations on arrays

  - *Assume the following declarations: int x[50];*
  int y[50];

  - **No** aggergate assignment: *x = y;*

    * How can you do it ?

```
for(i=0; i<50; i++)
  x[i] = y[i];
```

  - **No** aggergate comparison: *x == y;*

    * How can you do it ?

```
same = 1; // Think of it as a Boolean variable
for(i=0; i<50; i++)
  if(x[i] != y[i])
    same=0;
if(same)
  cout << "The arrays are the same" << endl;
else
  cout << "The arrays are different" << endl;
```

    * More efficient condition

```
same = 1; // Think of it as a Boolean variable
for(i=0; (i<50)&&(same); i++)
  if(x[i] != y[i])
    same=0;
if(same)
  cout << "The arrays are the same" << endl;
else
  cout << "The arrays are different" << endl;
```

- **No** aggregate arithmetic: *x = x + y;*

  * How can you do it ?

```
for(i=0; i<50; i++)
  x[i] = x[i] + y[i];
```

Not possible to return an entire array as the value of a value returning function: *return x;*

- Can we pass an array as a whole to a function ? **Yes !!**

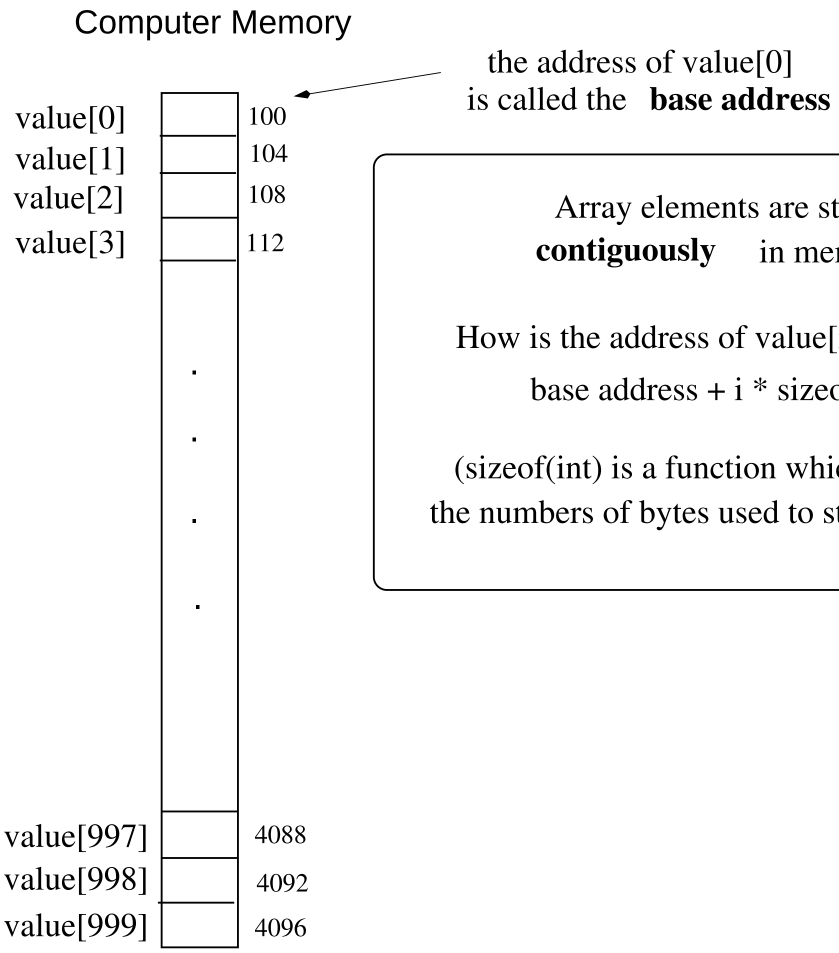
• Passing arrays as parameters to functions

- It is **impossible** to pass an array by value

- Arrays are **always** passed by reference

- When an array is passed to a function, its **base address** (i.e., the address of the first element) is passed to the function

- Every other element can be accessed if the base address is known !!!



Array elements are stored **contiguously** in memory !!

How is the address of value[i] computed ?

base address + i * sizeof(int)

(sizeof(int) is a function which returns the numbers of bytes used to store an integer)

function prototype

instead of &, use []

```
void ZeroOut (float[], int);

void main()
{
  float velocity[30];
  float Angle[9000];

     ...........

   ZeroOut(velocity, 30);
   ZeroOut(Angle, 9000);

     .............
}
```

Common mistake: Call the function as
ZeroOut(velocity[30],30)
(the address of the 30th element
is passed to the function !! not the base addess!!)

function definition

the dimension of the array is passed as a separate parameter

```
void ZeroOut(float arr[], int no_elements)
{
  int i;

  for(i=0; i<no_elements; i++)
     arr[i] = 0.0;
}
```

all the changes made in ZeroOut() will be effective in main()

- Can we prevent a function from modifying an array (as in call-by-value) ?

  * Yes !!! Use the reserved word "const"

```
void Copy(int destination[], const int source, int size)
{
 int i;

 for(i=0; i< size; i++)
 {
   destination[i] = source[i]; // OK
   source[i] = destination[i]; // ERROR !!!
 }
```

- Examples

  - Write a program that calculates the mean and the standard deviation of integers stored in a file (prob. 3. page 643)

  - Given a file with 1000 numbers ranging from 1 to 100, write a program which counts the number of times each number appears in the file

Recommended exercises

- *Quick Check* excersices: 1-8
- *Exam preparation* excersices: 1-4, 7-11