Memory Management
and
Pointers

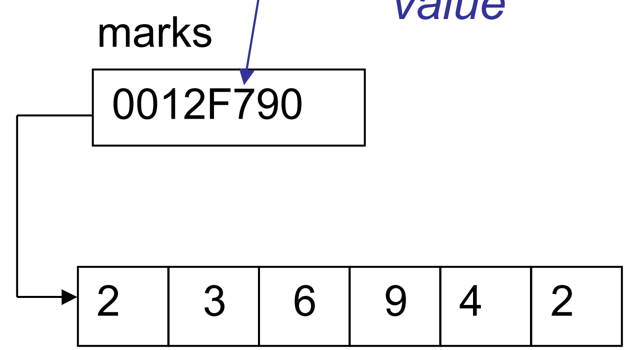
Memory pointers and arrays

- Remember: array variable stores the memory location of the 1st element of the array.

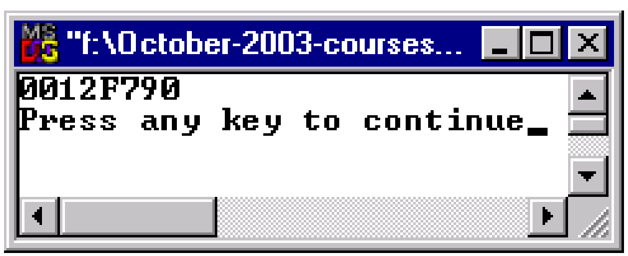
- Memory location of 1st element means literally, the address of the first byte of the 1st element.
- In C++, memory address of any variable is referred to as a ***pointer*** to that variable

This value cant be changed:

Array variable is not an *l-value*




```
int marks[6] = {2, 3, 6, 9, 4, 2};
cout << marks;
```



Array variable is not an l-value

Can't use it on the LHS of an assignment statement

```
marks[0] = 27;
```

ok – change the value at an array position

```
marks = 27;
```

not ok – cant change *where* the array is stored

Declaring a pointer

- Already using pointers without realising:
  - When we pass an argument into a function by reference
  - When we pass an array into a function
- In both cases, what is actually passed is a memory location rather than the value of a variable.
- C++ manages this for us.
  - If we change the value in the variable passed by reference, the value at that memory location is updated.
  - If we change an element of the array, the system calculates the memory address of that memory location, and updates the value stored there.
- It is possible (and often necessary) to write programs that directly manipulate pointers rather than leave the system to manipulate them for us.
- We can explicitly declare a variable which stores as its value a memory location
- At the same time, we specify what type of variable is going to be at that location
  - Syntax to declare that `pInt` is a variable which holds the memory location of some other integer variable:

```
int *pInt;
```

Terminology: pInt is a *pointer variable*

pInt is a *pointer to an integer*

Dynamic memory allocation

- `int *pInt; //pInt is a pointer to an int`
- At compile time, memory is statically allocated to hold the pointer, but *not* the int to which it points.
- Memory for the int itself is allocated dynamically, when the `new` operator is used.

```
int *p1;
p1 = new int;
```

allocate memory for an integer, and store its address in the pointer p1

  - Memory is allocated from the 'freestore' or memory heap.
  - The `new` operator returns a pointer to the memory it has allocated.
  - Notice that the variable pointed to by `p1` is now a variable with no name - we can only refer to it as "the variable pointed to by `p1`", and the syntax for doing this is **`*p1`**
- Danger: if we assign some other address to p1, we will not be able to access the no-name variable which was allocated memory with `new` above.
  - It is still memory reserved by our program, but cannot be accessed, so it just means our program now has a little less spare memory available to it. This is called a ***memory leak***.

Failure to allocate memory

- Future "new" operations will fail if freestore (memory heap) is "full“, so we should always handle the situation where the operation fails
- Older compilers:
  - Test if null was returned by call to *new: i*f new succeeded, the program continues
- Newer compilers *may* use a later C++ standard
  - If new operation fails:
    - Program by default terminates automatically
    - Produces error message
  - Still good practice to use NULL check (just in case)
  - Later, we will also ‘catch the exception’

```
int *p;
p = new int;
if (p == NULL)
{
     cout << "Error: Insufficient memory.\n";
     exit(1);
}
```

Memory Management

- We de-allocate memory using the operation delete

```
double *p = new double;
…
delete p; //memory to which p pointed is now freed up
```

- The size of the freestore varies with implementations
- Typically it is very large
  - Most programs won't use all memory
- But memory should always be managed carefully
  - Memory IS finite, regardless of how much there is!
  - Good practice demands memory management
  - And it is a solid software engineering principle
- Avoid memory leaking from your program
  - Always de-allocate memory before a pointer is assigned a new value
  - Always de-allocate any memory allocated in a function before it returns and its pointer goes out of scope
  - If memory is allocated in a constructor, always de-allocate it in the destructor

Array Variables

- Recall: arrays stored in contiguous memory addresses, and the array variable "refers to" first indexed variable
  - So array variable is a kind of pointer variable!

```
int a[10];
int  *p;
```

a and p are both pointer variables!

- But the array variable is MORE than a pointer variable
  - Array was allocated in memory already
  - The array variable MUST point there…always! It cannot be changed!
  - In contrast to ordinary pointers which can (& typically do) change
  - Can perform assignments like this:

```
p = a;
```

p now points where a points (i.e. to first indexed variable of array a)

  - But not like this

```
a = p;
```

ILLEGAL! Because the array variable is CONSTANT pointer and cannot be an l-value

Dynamic Arrays

- To allocate the memory for an array dynamically, we again use the *`new`* operator
  - Dynamically allocate the space with pointer variable
  - Then treat the pointer like a standard array variable

```
double *d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array, with ten elements, base type double, and stores pointer to it in the variable `d`
- Recall that one of the limitations of the array declaration is that we must specify size first ..
  - … but may not know it until program runs!
  - Must "estimate" maximum size needed
    - Sometimes possible, but “wastes" memory
    - And must take care not to exceed the amount allocated
- The advantage of the dynamic array is that its size can be allocated at runtime.
  - Dynamic arrays can grow and shrink as needed

Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - So should be destroyed at run-time

    ```
    d = new double[10];
    … //Processing
    delete[] d;
    ```

  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: *d* still points there!
    - Should set d = NULL;

      ```
      delete[] d;
      d = NULL;  //more robust code
      ```