# PIC 10A

Lecture 25: Pointers and Arrays

## Arrays and pointers

Consider an an array

```
int numbers[5];
```

```
//Lets assume numbers array gets filled somehow
```

Earlier we noted that you cannot output an array by

```
cout << numbers;
```

But if you try you will not get an error.  Output will be something like

```
0x7FF10A
```

Turns out that numbers is actually the memory address of the first element of the array numbers.

## Array name as a pointer

So numbers is actually same as `&numbers[0]`.

Arrays name is a pointer to the beginning of the array.

```
int numbers = {1,2,3,4,5};

int *p;

p = numbers;

cout << *p << " " << numbers[0];  //What is the output?
```

# Arrays store data consecutively

| | Slot 0 | Slot 1 | Slot 2 | Slot 3 | Slot 4 |
|---|---|---|---|---|---|
| numbers | 3 | 15 | -8 | 276 | 4 |
| | 0E7754 | 0E7755 | 0E7756 | 0E7757 | 0E7758 |

```
int *p;

p = numbers; // p points to numbers[0]
```

Because arrays store data in consecutive order in memory we can write

```
p++;  // p now points to numbers[1]

cout << *p;
```

# Arrays and pointers

```
int numbers={1,2,3,4,5};
int *p =numbers;

cout << *(p+2);
```

What is the output? Where is p pointing to?

Address of the ith slot is `numbers + i`.

We can access the ith slot by `*(numbers+i)`.

Why do we need parenthesis?

```
*(numbers+3) = 7; // What does the array store now?
```

## [ ] as a dereferencing operator

We see that `*(numbers+3)` is just like `numbers[3]`. The `[ ]` act like a dereferencing operator.

Consider again

```
int *p = numbers;
```

We can of course write

```
*(p+3) = 0;
```

But we could also write

```
p[3] = 0; // !!
```

## Difference between arrays and pointers

Wow pointers seem to act just like arrays and vice versa!

Not quite:

```
int numbers[5];
```

numbers is a pointer but we also have reserved 5 consecutive memory slots.

```
int *p;
```

p is a pointer but there is currently no memory reserved where p can point to.

Where as I can refer to numbers[3]. I cannot refer to p[3] unless I make p first point to an array.

# Array names are constant pointers

```
int numbers={1,2,3,4,5};
```

```
numbers is only a int * const
```

This means that the arrays name while being a pointer, is not really a variable. It always points to the beginning of the same array.

```
int a={1,2,3};
int b={4,5,6};
a = b; // This is not legal
```

```
int *p;
int a={1,2,3};
int b={4,5,6};
p=a;
p=b; // This is legal
```

## Example

```
void fun(int* p)
{
    *(p+3)=40;
    p++;
    *p=20;
    return;
}
int main()
{
    int MyArray[5]={1,2,3,4,5};
    int *p;

    p=MyArray;
    fun(p);
    *p=10;

    return 0;
}
```

## Example

```
void fun(int* &p) //note the syntax to pass a pointer by reference
{
   *(p+3)=40;
   p++;
   *p=20;
   return;
}
```

What is the output now?

When pointer is passed by value, the pointers value will not change in main, But the pointer can change its value inside the function and what is more important, it can change whatever it points to permanently!

When a pointer is passed by reference, whatever changes to the pointer are made in a function stay with it.

## Example

```
void my_fun (int* ptr)
{
    ptr += 3;
    *ptr = 30;
    ptr--;
    *ptr = 20;
    return;
}
```

```
int main()
{
    int ary[5] = {1,2,3,4,5};
    int* p = ary;
    p++;
    my_fun(p);
    for (int i=0; i<5; i++)
        cout << ary[i] << " ";
    return 0;
}
```

```
1 2 3 20 30
```

# Example

```
void fun1 (int *p, int *&q)
{
     *p = 100;
     p = p + 2;
     *p = *q;
     *q = *(p+1);
     cout << "p=" << *p << "
q=" << *q << "\n";
}
```

```
int main ( )
{
     int a[5] = {2,4,6,8,10};
     if (fun2 (a,5))
          cout << "true\n";
     else
          cout << "false\n";
     for (int i=0; i<5; i++)
          cout << a[i] << " ";
     return 0;
}
```

```
bool fun2 (int x[], int size)
{
     int *p = x;
     int *q;
     q = p;
     for (int i=1; i<size; i++)
     {
     x[i] += x[i-1];
     cout << x[i] << " ";
     }
     cout << "\n";
     fun1 (p, q);
     return (*p == *q);
}
```

Output:

```
6 12 20 30
p=100 q=20
true
20 6 100 20 30
```

## Dynamic arrays

After you declare an array it has a fixed size:

```
// There is no push_back for arrays I am stuck with an
array of size 10
char MyArray[10];

// This is only legal if MAX_LENGTH is const int.
char MyArray[MAX_LENGTH];
```

The idea is to use pointers to create a dynamic array whose size can be changed.

# Dynamic arrays

```
int *MyArray = new int[10]; // allocates 10 ints in consequtive memeory.
```

Now things like

```
MyArray[3]=6;
```

are possible. Remember `MyArray` is a pointer and [ ] act like dereferencing operator.

# push_back for dynamic arrays

We cannot change size of ordinary arrays.  With dynamic arrays it is easy.

Lets make a push_back function for arrays

```
void push_back(int * &MyArray, int &size, int value)
{
    Int * temp_array = new int[size+1];

    // copy all elements of MyArray to temp_array

        temp_array[size] = value;

    delete [] MyArray;

    MyArray = temp_array;


    ++size;

    return;
}
```

## Functions with dynamic arrays

```
void display(int *MyArray, int size)
{
for (int i=0; i<size; ++i)
   cout << MyArray[i];

return;
}
```

## Using a function that takes an argument of dynamic array

```
int main()
{
   int *p = new int[5];

   // Fill p array
   p+=2;
   display(p, 3);

   return;
}
```

# Swapping dynamic arrays

```
void swap(int * &array2, int * &array1)
{
   int *temp;
   temp=array2;
   array2=array1;
   array1=temp;
}
```

I just swapped two arrays with one switch.  Why can I not do this with regular arrays?