

# Memory Management and Pointers (part 2)

# Function that Returns an Array

- Recall an array type is NOT allowed as return-type of function

- We cannot say

- ```
int [] someFunction();    // ILLEGAL!
```

- But now we have a method to return an array from a function: We declare the function with a return type which is *a pointer*.

- ```
int* someFunction();    // LEGAL!
```

- Since the array is actually a pointer to its first element, the array can now be returned from the function!

# The 'address of' operator

*p is a pointer to a double,  
r is a pointer to a double.*

**double \*p, q, \*r;**

*but notice q is a double  
(not a pointer)*

suppose we would like the double to which  
p points to be the variable q

- we need to store the memory location of q as the value of the variable p
- There is an operator for providing the memory location of a variable: this is the '*address-of*' operator &

&q means the address of (*memory location of*) q

**p = &q;** *p is assigned a value which is the address of q*

# Initialising pointers

- `int *p1; //p1 is a pointer to an int.`
  - At this stage p1 has not been initialised.
  - It holds a ‘garbage value’
  - If we go to the memory location indicated by that garbage value, it could be any piece of memory already being used by our program, or by the system!  
**Very dangerous!!!**
  - If we assign a value to p1, we must make sure it is the address of an integer variable in the program.

```
int sales, *p1;
```

- This might be by allocating the memory dynamically

```
p1 = new int;
```

- Or by using the ‘address of’ operator

```
p1 = &sales;
```

# De-referencing pointers


- De-referencing a pointer means finding the variable to which it points
  - The 'de-referencing operator' is **\***
  - If `p1` is a pointer to a double, then we can refer to **the double to which `p1` points** as **`*p1`**
  - **Once it has been initialised**, can use `*p1` as any other double variable

//assign it to another double variable

**`double myDouble = *p1;`** 

**But if `p1` hasn't been initialised, it contains a garbage address, so this line would be totally unpredictable.**

// assign a value to the variable to which `p1` points

**`*p1 = 45.6;`** 

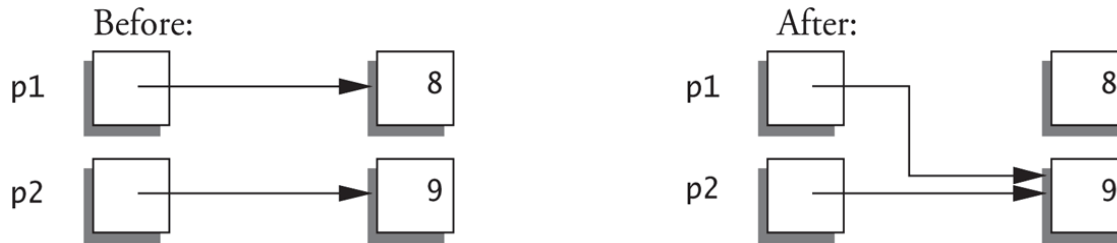
**But if `p1` hasn't been initialised, this would try to overwrite the memory at whatever garbage address it contains, again with dangerous and unpredictable results**

- **we need to allocate some memory space to hold the double variable to which `p1` points before we assign a *literal* value to that double variable**

# Assignment with pointers

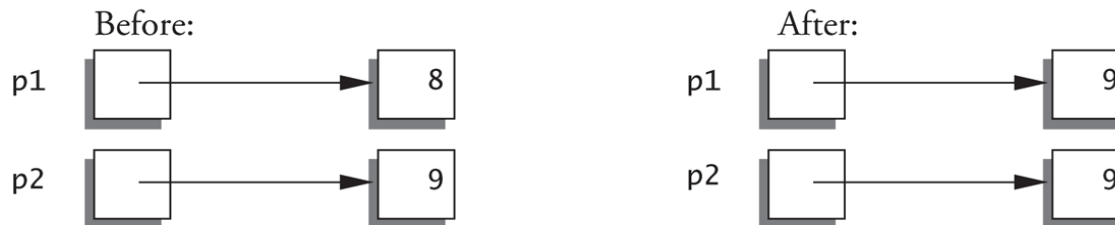
- `int *p1, *p2;`
- Pointer variables can be "assigned": `p1 = p2;`
  - "Make p1 point to where p2 points"

`p1 = p2;`



- Do not confuse with: `*p1 = *p2;`
  - Which assigns "value pointed to" by p1, to "value pointed to" by p2

`*p1 = *p2;`



# Care with the delete operator – dangling pointers!

```
double *p = new double;
```

```
...
```

```
delete p; //memory to which p pointed is now freed up
```

- The memory to which p points has now been de-allocated
  - But p still points there! It is pointing into the ‘unknown’
  - This is called "dangling pointer"
  - If p is now de-referenced (i.e we try to access \*p, the double to which p ‘points’)
    - Unpredictable and often disastrous results!
  - Solution: assign pointer to NULL after delete, and always check a pointer is not NULL before using it

```
delete p;
```

```
p = NULL;
```

```
...
```

```
if (p != NULL)
```

```
    cout << *p;
```

# Basic Pointer Manipulations (1 of 2)

## Display 10.2 Basic Pointer Manipulations

---

```
1 //Program to demonstrate pointers and dynamic variables.
2 #include <iostream>
3 using std::cout;
4 using std::endl;

5 int main( )
6 {
7     int *p1, *p2;

8     p1 = new int;
9     *p1 = 42;
10    p2 = p1;
11    cout << "*p1 == " << *p1 << endl;
12    cout << "*p2 == " << *p2 << endl;

13    *p2 = 53;
14    cout << "*p1 == " << *p1 << endl;
15    cout << "*p2 == " << *p2 << endl;

16    p1 = new int;
17    *p1 = 88;
18    cout << "*p1 == " << *p1 << endl;
19    cout << "*p2 == " << *p2 << endl;

20    cout << "Hope you got the point of this example!\n";
21    return 0;
22 }
```

### SAMPLE DIALOGUE

\*p1 == 42

\*p2 == 42

\*p1 == 53

\*p2 == 53

\*p1 == 88

\*p2 == 53

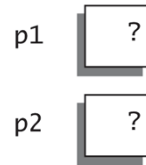
Hope you got the point of this example!

---

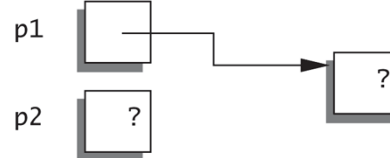


# Explanation of previous display

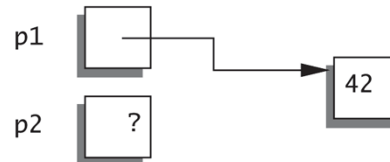
(a)  
`int *p1, *p2;`



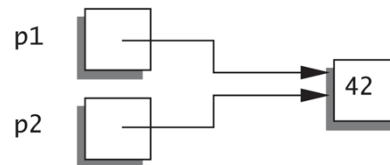
(b)  
`p1 = new int;`



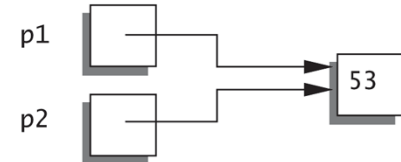
(c)  
`*p1 = 42;`



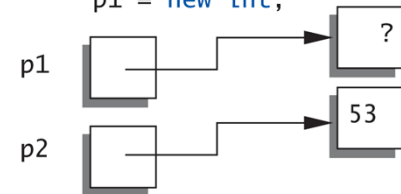
(d)  
`p2 = p1;`



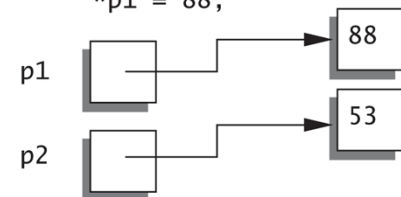
(e)  
`*p2 = 53;`



(f)  
`p1 = new int;`



(g)  
`*p1 = 88;`



# Classes and Pointers

```
class DayOfYear
{
    public:
        ...
        void output();
    private:
        ...
        int month;
        int day;
};
```

```
int main()
{
    DayOfYear birthday;
    ...
    birthday.output();
}
```

Call the member function `output()` on the `DayOfYear` object `birthday`

```
int main()
{
    DayOfYear *birthday;
    birthday = new DayOfYear;
    ...
    (*birthday).output();
}
```

Call the member function `output()` on the `DayOfYear` object pointed to by `birthday`

```
int main()
{
    DayOfYear *birthday;
    birthday = new DayOfYear;
    ...
    birthday->output();
}
```

A neater notation which does the same thing

# The 'this' pointer

- Can use 'this' in a class method to refer to the object on which the method has been called
  - in C++, 'this' is a pointer

`this->day` ← correct

`this.day` ← wrong!!!! – will not compile

- Don't use 'this' unless you really need to
  - the level of indirection is a small extra overhead in your program
  - Much of the time you can avoid it by making sure the formal arguments to a function do not have the same name as the data member of the class

# Storage class of a variable: Automatic vs Static

- **Automatic memory allocation** has been used for all variables we have encountered so far

```
int sales, *p1;
```

- Where sales and p1 are defined inside a function, **memory is allocated automatically** when the function is entered. (by default – although *this can be overridden, see below*)
- it is allocated on the function memory stack
- It only exists while the function is running.
- Note that the memory bound to pointer p1 holds an address
  - The memory allocated would be the same size whether p1 pointed to a variable of type char, or int, or string, a class we have defined, or any other type.
  - If memory is allocated with `new`, and its address stored in p1, this memory will no longer be referenced when the function returns (a memory leak unless `delete` is used before the function returns)

- **Static Memory Allocation**

- Suppose a **global variable** is declared (**outside of a function**)
- Memory for this is allocated 'statically'
- It is allocated from the static memory pool
- It exists as soon as the program execution starts, and remains until the program exits
- The same is true for a local variable in a function *if it is explicitly declared static*
- *The effect of a local variable being declared static is that, although it can only be accessed in the function code, any value assigned to it will still be there the next time the function is entered.*

# Dynamic Memory Allocation

- Dynamic memory is not allocated until the program is running: It uses the *new* operator
  - Memory is allocated from the ‘freestore’ or memory heap
  - The new operator returns a pointer to the memory it has allocated.
  - *See the next overheads for a visual representation of how memory allocation works.*