Dynamic variables

- Memory allocated at run time
- Use as much memory as you need
- Manually return back to the heap when done

Static variables
- Memory allocated at compile time
- Maximum size must be known beforehand
- Auto deallocation

Dynamic variables
- Memory allocated at run time
- Use as much memory as you need
- Manually return back to the heap when done
In C++, dynamic variables can be created using pointers and the operator `new`.

Operator `new` is used to allocate either a single variable, or an array of variables:

- `new dataType;` // single variable
- `new dataType[size];` // an array of variables

Variables allocated using `new` have no names, and must be assigned to a pointer variable:

- `dataType *p = new dataType;`
- `int *p = new int;`

There is no direct access to the dynamically allocated memory – you can only use it through pointer variables.
**Operator “new”**

- Consider:
  - `int* p;`
  - `int x = 42;`
  - `p = &x;`

- And:
  - `int* p = new int;`
  - `*p = 42;`

- In both examples, pointer `p` points to a memory location that stores the value **42**

- In the first example, **42** is stored on the stack and can be accessed through `x` and `*p`

- In the second example, **42** is stored on the heap and can be accessed only through `*p`
OPERATOR “DELETE”

- All memory allocated using `new` must be manually de-allocated using `delete`
- The deallocated pointer variable can be reused by allocating new memory to it
- Operator `delete` is used to deallocate both single variables and arrays of variables:
  - `int *ptrVar = new int; // allocate a single int`
  - `delete ptrVar; // deallocate a single variable`
  - `int *ptrArr = new int[x]; // allocate array`
  - `delete [] ptrArr; // deallocate an array`
- The `delete` operator does not delete the pointer — it deletes (deallocates) the memory that the pointer points to
OPERATOR “DELETE”: DANGLING POINTERS

- The **delete** operator does not delete the pointer — it deletes the memory that the pointer points to.

**Before delete p;**

<table>
<thead>
<tr>
<th>...</th>
<th>500</th>
<th>...</th>
<th>42</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address:</td>
<td>350</td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

```c
int * p
```

**After delete p;**

<table>
<thead>
<tr>
<th>...</th>
<th>500</th>
<th>...</th>
<th>???</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address:</td>
<td>350</td>
<td>...</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

```c
int * p
```

The heap memory is free, but p still stores an address.
A pointer that points to an address that is no longer valid is called a **dangling** pointer.

If you use a dangling pointer, you will either **corrupt data on the heap**, or your program will crash (“segmentation fault”).

To avoid dangling pointers, explicitly set your pointers to NULL after you delete them:

- `delete p;` // deallocate heap memory
- `p = 0;` // set p to NULL

```
... | NULL | ... | ??? | ...
```

Address: 350 500

```
int *p
```
What happens if you don’t delete pointers?

```cpp
void doSomething()
{
    int *p = new int;
    *p = 42;
}
```

When you exit `doSomething()`, `p` goes out of scope.

However, the memory allocated to `p` remains allocated (reserved) – it can not be reused by other variables.

Dynamically allocated memory can only be accessed through `p`.

When `p` goes out of scope, the memory is still allocated, but is not accessible.

This is called a memory leak.

You will eventually run out of memory if you do this!
POINTERS & MEMORY LEAKS

- Loosing the address of dynamically allocated memory results in a memory leak:
  ```cpp
  int x = 5;
  int *p = new int; // allocate memory
  p = &x; // old address lost, memory leak results
  ```

- Memory leak can also result from a double allocation:
  ```cpp
  int *p = new int; // allocate memory
  p = new int; // allocate memory... again
  // old address lost, memory leak results
  ```

- Make sure not to lose the addresses of dynamically allocated memory
- Provide a `delete` for every `new`
**Dynamic arrays**

- Dynamic arrays are allocated on the heap using the `new` keyword:
  
  ```
  int *arrayPtr = new int[10];
  // create a 10-element array dynamically
  ```

- The following is possible now:
  
  ```
  cout << "How many variables do you want? ";
  int nVars;
  cin >> nVars;
  int *array = new int[nVars]; // correct!
  ```

- Pointer `array` now points to the first element of the array we have just created
Both static and dynamic array variables store the address of the first array element

- Static: `int array[5];`
- Dynamic: `int *array = new int[5];`

![Diagram of array and memory addresses]
**Dynamic two-dimensional arrays**

- **Static two-dimensional:**
  - `int a[3][3];`

- **Dynamic two-dimensional:** a dynamic array of dynamic arrays (a pointer to an array of pointers)
To declare a two dimensional array, we first declare an array of pointers (rows), and then declare every row to be an array of columns:

```c
int **a = new int * [3]; // rows
for(int row = 0; row < 3; row++)
{
    a[row] = new int[3]; // columns
}
```

```
memory

a[0] * a[0][0] a[0][1] a[0][2]
 a[1] * a[1][0] a[1][1] a[1][2]
```
Dynamic two-dimensional arrays

- Individual elements of a 2D dynamic array can still be accessed through double subscript:
  ```cpp
  for(int row = 0; row < 3; row++)
      for(int col = 0; col < 3; col++)
      {
          cout << a[row][col] << endl;
      }
  ```

- To deallocate all pointers, you have to loop through the array:
  ```cpp
  for(int row = 0; row < 3; row++)
  {
      delete [] a[row];
  }
  delete [] a;
  ```
Return type of a function can be a pointer: we might want to return the address, not the value.

However, returning an address of a local variable will result in an error:

```c
int* multByTwo(int n)
{
    int nValue = n * 2;
    int *p = &nValue; // get the address of nValue
    return p;         // return the pointer
}
```

// nValue goes out of scope here!
// Our pointer is useless now...
Pointers to dynamically allocated memory can be safely returned: dynamic memory has no scope

Useful example: returning a dynamic array

```cpp
int* allocateArray(int nSize)
{
    int *x = new int[nSize];
    return x;
}
```

```cpp
int main()
{
    int *array = allocateArray(25);
    // do stuff with array
    // remember to deallocate memory when done:
    delete[] array;
    return 0;
}
```
**Pointers & Objects**

- Objects can be created dynamically and assigned to pointers:
  
  ```cpp
  Vector * vec = new Vector(3);
  ```

- What is we want to store dynamic objects in a dynamic array?
  
  - Every *dynamic* object will be a pointer
  - The array itself must be *dynamic* – thus, it will also be a pointer
  - Result: we need a *double pointer*:
    
    ```cpp
    Vector** * vec = new Vector*[5];
    // an 1D array of 5 Vector pointers
    vec[0] = new Vector(3);
    // every element is dynamically allocated
    // How would you deallocate this array?
    ```
POINTERS TO POINTERS

What can this be?

- Vector * vec;
  - A pointer to a static object
  - A dynamically allocated object
  - A dynamically allocated array of static objects

What can this be?

- Vector ** vec;
  - A pointer to a pointer to a static object
  - A pointer to a dynamically allocated object
  - A dynamically allocated array of pointers to static objects
  - A dynamically allocated array of dynamic objects
  - A two-dimensional array of static objects

What can this be?

- Vector *** vec; // try and list all the options
Assigning a pointer to another pointer creates a shallow copy:

```c
int *first = new int[nSize];
// fill array with data
int *second = first;
// now both first and second store the same address,
// but the array is stored in memory once:
```

![Diagram showing shallow copy of an array using pointers](image.png)
SHALLOW COPY & POINTERS

- If we delete either of the two pointers, both will be dangling:

```c
delete [] second;
// The data has been deleted,
// even though first still points to it!
```

![Diagram of pointers first and second]
DEEP COPY & POINTERS

To make a deep copy of an array, iterate through it:

```cpp
int *first = new int[nSize];
// fill array with data
int *second = new int[nSize];
for (int i = 0; i < nSize; i++)
{
    second[i] = first[i];
}
```
**Pointers & Functions**

<table>
<thead>
<tr>
<th>“Pass by value”</th>
<th>“Pass by reference”</th>
</tr>
</thead>
</table>

```cpp
void doStuff(int x)
{
    x++;
}
int y = 42;
doStuff(y);
cout << y << endl;
// The output is 42 :
// y has not being modified
```

```cpp
void doStuff(int &x)
// Notice the “&”
{
    x++;
}
int y = 42;
doStuff(y);
cout << y << endl;
// The output is 43 :
// y has being modified!
```
## Pointers & Functions

<table>
<thead>
<tr>
<th>Pointer parameter: “Pass by value”</th>
<th>Pointer parameter: “Pass by reference”</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void doStuff(int * x)</code></td>
<td><code>void doStuff(int * &amp;x)</code></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>*x = 42;</td>
<td>*x = 42;</td>
</tr>
<tr>
<td>}</td>
<td>x = NULL;</td>
</tr>
<tr>
<td>// *x can be modified,</td>
<td>// Both *x and x</td>
</tr>
<tr>
<td>// but x (address) can</td>
<td>// can be modified</td>
</tr>
<tr>
<td>// not be modified</td>
<td></td>
</tr>
</tbody>
</table>

**Pass by pointer:** Even though the pointer address can not be modified when a pointer is passed by value, the value of the memory location to which the pointer points can still be modified – thus, passing a pointer “by value” is similar to passing the value that it points to “by reference”
**Pointers & Functions**

<table>
<thead>
<tr>
<th>“Pass by pointer”</th>
<th>“Pass by reference”</th>
</tr>
</thead>
<tbody>
<tr>
<td>void doStuff(int *x)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>(*x)++;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>int y = 42;</td>
<td></td>
</tr>
<tr>
<td>doStuff(&amp;y);</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; y &lt;&lt; endl;</td>
<td></td>
</tr>
<tr>
<td>// The output is 43 :</td>
<td></td>
</tr>
<tr>
<td>// y has being modified!</td>
<td></td>
</tr>
</tbody>
</table>

| void doStuff(int &x)  |
| {                  |
|    x++;            |
| }                  |
| int y = 42;       |
| doStuff(y);       |
| cout << y << endl;|
| // The output is 43 : |
| // y has being modified! |