CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S2d**)

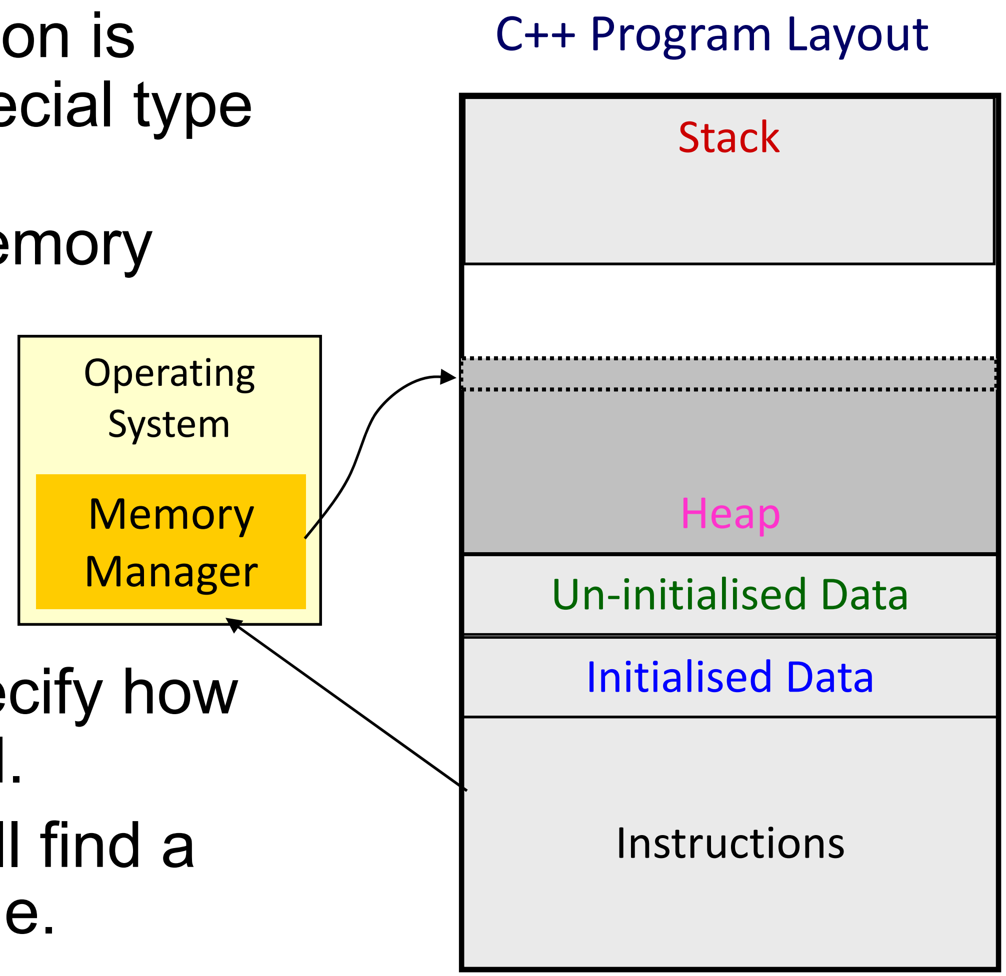
C++ Foundations IV: Dynamics

Dynamics

- If you sensibly can, you should be relying on the standard structures supported within the standard libraries.
- But, as you should see in CSCI203/CSCI803, the standard tools are not appropriate in every situation and you may be able to do something more efficiently in a specific context.
  - CSCI203/CSCI803 and CSCI251/CSCI851 are somewhat opposite in this respect.
- In C++ you may want to dynamically manipulate memory.
  - It's kind of dangerous but can improve performance.

Dynamic Memory Allocation

- Dynamic memory allocation is carried out by using a special type of operator that directly communicate with the Memory Manager.



- A programmer has to specify how much memory is required.
- The memory manager will find a location currently available.

Stack or Heap

- The typical variables we have seen so far have been local.
- They exist only while we are within scope, and the compiler deals with creating and destroying them.
- These types of entities are stored in memory on the stack.

- But there is other memory available, on the free store or heap.
- Allocation of this memory is at run-time, and use of this memory is persistent so we need to explicitly say when we want to stop using it.

Why use dynamic (heap) memory?

- We don’t know how many elements we need.
  - Improved storage efficiency.
  - Although we have to be careful with resizing costs.
- Sharing data/state.
  - Some data is associated with or needs to be known by part of our program, but is owned somewhere else.
- That sharing could be part of persistence outside of the initial creation scope.
- …

`new` and `delete`

- We firstly set up a pointer …

```
int *intpointer;
```

- … then dynamically allocate memory with `new`.

```
intpointer = new int;
```

- `new` is a type safe operation, it returns a pointer to the type given, `int` in this case.
  - That pointer points to an object of the specified type, here an `int`, with the amount of memory required being automatically determined on the basis of the operand type.
- If we use `new` we need to use `delete` to release the memory.

```
delete intpointer;
```

- If we don't we get a **memory leak**.

- Variables can be default initialised, with the default value type and sometimes location dependent.
  - Built-in types defined outside function bodies are initialised to zero, those within are uninitialized, effectively having an undefined values.
- We can also initialise the variables ourselves when we set up the memory,

```
int *p = new int(5);
```

- The type specifier `auto` can come in useful again.

```
auto p1 = new auto(obj);
```

A simple memory leak …

- Consider the following …

```
    int *p;
    p = new int(5);
    cout << p << endl;
//  delete p;
```

- There will be a memory leak.
- On Banshee this could be checked using `bcheck`.
  - Unsure on capa → Initial look didn't find anything but there is likely a profiler of some sort.

new[] ...

- To create a dynamic array we can use the `new[]` operator.

```
int *intVar;

intVar = new int[100]; // dynamic array

for(int i = 0; i < 100; ++i)
  intVar[i] = 25-i; // initialize the array

delete [] intVar; // frees the allocated array
```

A final note on `delete[]` …

- You have to be careful if you have something like a pointer to an array of pointers, …

```
Person **p = new Person* [2];
p[0] = new Person("Peter");
p[1] = new Person("Alex");
```

- Using `delete[] p;` just causes the `p` pointer to be released, not the actual objects themselves.
- You could step through the different index values and use `delete p[index]` on each.
- Or, as will probably be discussed later, you could use a wrapper class.

- Example:

```
float **fVar;
fVar = new float* [10]; // allocate pointer
                         array, 10 float pointers

for(int i = 0; i < 10; ++i)
        fVar[i] = new float[10]; // allocate
                                    memory to each

.   .   .   .
.   .   .   .
for(int i = 0; i < 10; ++i)
        delete [] fVar[i];

delete [] fVar;
```

Dynamic memory management: Problems

- The textbook describes three common problems:
  1. Forgetting to delete memory.
  2. Using an object after it was deleted.
  3. Deleting the same memory twice.

- We will come back to smart pointers and their use in memory management for classes.
  - They take care of these problem.

Some faults: Seg. and Bus.

- Segmentation faults occur when you try to use memory which does not belong to you, typically:
  - Out of bounds array references.
  - Reference through un-initialized or dangling pointers, the latter being pointers to already freed memory.

```
char *s = "Hello";
*s='H';
```

- Compiled with `g++` we get a warning and we get a seg. fault at run time.

- Segmentation faults are to do with memory access violations.
  - You aren’t allowed to access the memory specified.
  - A compiler won’t necessarily care or help.
- Bus faults are similarly run time problems to do with accessing memory.
  - But it relates to trying to access memory that cannot be physically addressed.
  - The memory is invalid for the access type specified, usually to do with memory misalignment.