Memory Management

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Outline

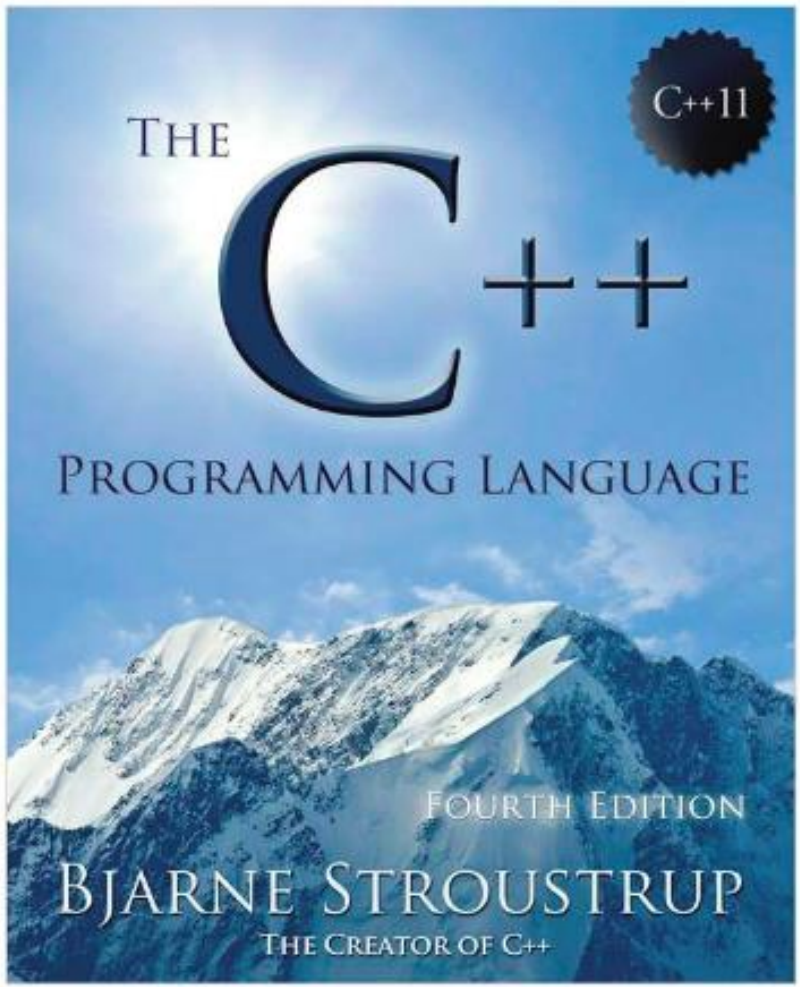
Outline

Outline

Outline

References

- Kernighan & Ritchie (1988), *The C Programming Language, 2nd Edition*
  - This one is co-authored by the creator of C
- Stroustrup (2011), *The C++ Programming Language, 4th Edition*
  - The inventor of C++
- Available in the library!



Outline

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are
  - Is the value a parameter for the function I am in? If yes, I can access it

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are
  - Is the value a parameter for the function I am in? If yes, I can access it
  - Has the value been declared in my current, local scope (between the same set of curly brackets I am in)? If yes, I can access it

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are
  - Is the value a parameter for the function I am in? If yes, I can access it
  - Has the value been declared in my current, local scope (between the same set of curly brackets I am in)? If yes, I can access it
  - Has the value been declared in a surrounding scope (between a set of curly brackets that surrounds my curly brackets)? If yes, I can access it

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are
  - Is the value a parameter for the function I am in? If yes, I can access it
  - Has the value been declared in my current, local scope (between the same set of curly brackets I am in)? If yes, I can access it
  - Has the value been declared in a surrounding scope (between a set of curly brackets that surrounds my curly brackets)? If yes, I can access it
  - In any other case then you won't have access

Scope

- Each variable has a scope - *it is only visible and valid after it has been declared*
  - Performing operations before declaring a variable results in compiler errors
- Each function has a scope
  - It can only access values that were created within that scope
  - Or it can access values passed into that scope (i.e. parameters to the function)
- Scope is another area which can trip up the new programmer. In C based languages, simplified rules of thumb are
  - Is the value a parameter for the function I am in? If yes, I can access it
  - Has the value been declared in my current, local scope (between the same set of curly brackets I am in)? If yes, I can access it
  - Has the value been declared in a surrounding scope (between a set of curly brackets that surrounds my curly brackets)? If yes, I can access it
  - In any other case then you won't have access
    - Unless the value is globally declared, which should be avoided

Scope

- The notion of scope also applies to code we declare inside curly braces

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`
    - `while`

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`
    - `while`
    - `for`

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`
    - `while`
    - `for`
- When we declare a new scope inside a function, the function *cannot* see what is declared inside the new scope

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`
    - `while`
    - `for`
- When we declare a new scope inside a function, the function *cannot* see what is declared inside the new scope
- From within the new scope, we can still access values within the function

Scope

- The notion of scope also applies to code we declare inside curly braces
  - { }
  - Control statement blocks such as
    - `if ... else`
    - `while`
    - `for`
- When we declare a new scope inside a function, the function *cannot* see what is declared inside the new scope
- From within the new scope, we can still access values within the function
  - *Unless* we declare another value with the same name - this hides the value in that scope

Scope

```
void function()
{
    // Main scope of the function
    {
        // Scope A - can see main scope
        {
            // Scope B - can see scope A and main
                scope
        }
        {
            // Scope C - can see scope A and main
                scope

            // Scope B is no longer valid
        }
    }
}
```

Scope and Stack

- So far we have been using automatic variables

Scope and Stack

- So far we have been using automatic variables
  - Variables are allocated and deallocated automatically when our program enters and leaves a scope

Scope and Stack

- So far we have been using automatic variables
  - Variables are allocated and deallocated automatically when our program enters and leaves a scope
- Memory management happens automatically, at compile time, and there is little we have to do.

Scope and Stack

- So far we have been using automatic variables
  - Variables are allocated and deallocated automatically when our program enters and leaves a scope
- Memory management happens automatically, at compile time, and there is little we have to do.
- When looking through a debugger, we can see variables are declared in and out of the stack (push / pop) when we enter and leave a scope

Question With your current knowledge of C, what strategies can be used to ensure variables are visible across different scopes?

Question With your current knowledge of C, what strategies can be used to ensure variables are visible across different scopes?

Answer In previous units we mentioned the use of the `extern` keyword, which can help.

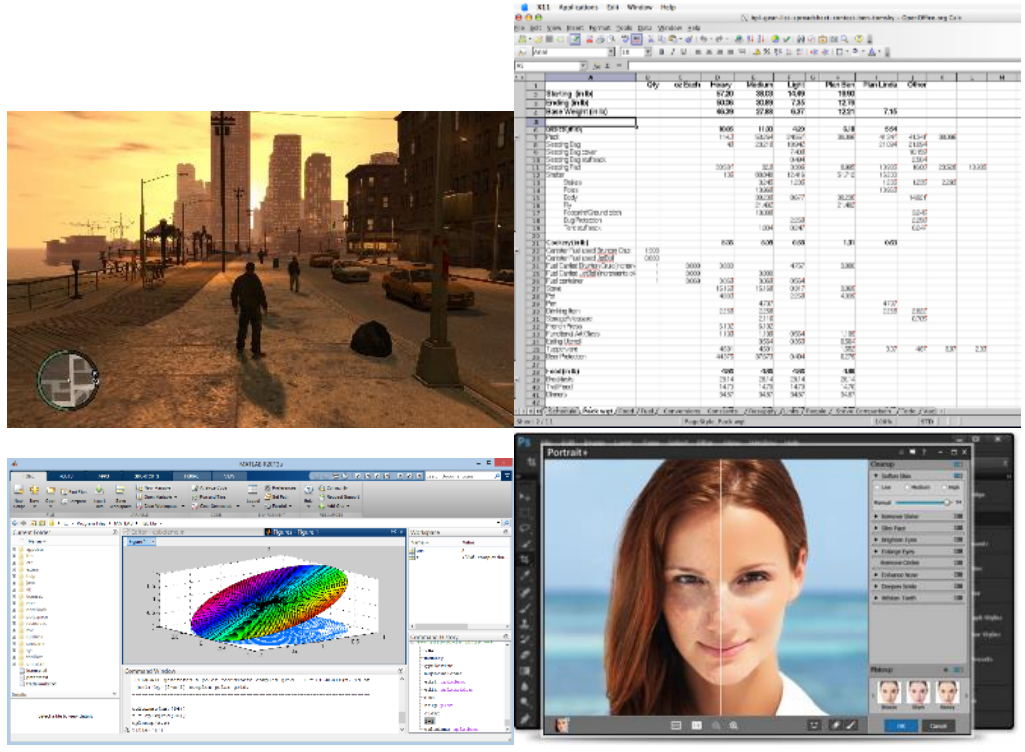
Question With your current knowledge of C, what strategies can be used to ensure variables are visible across different scopes?

Answer In previous units we mentioned the use of the `extern` keyword, which can help.

- Also you can declare global variables - usually considered bad practice!

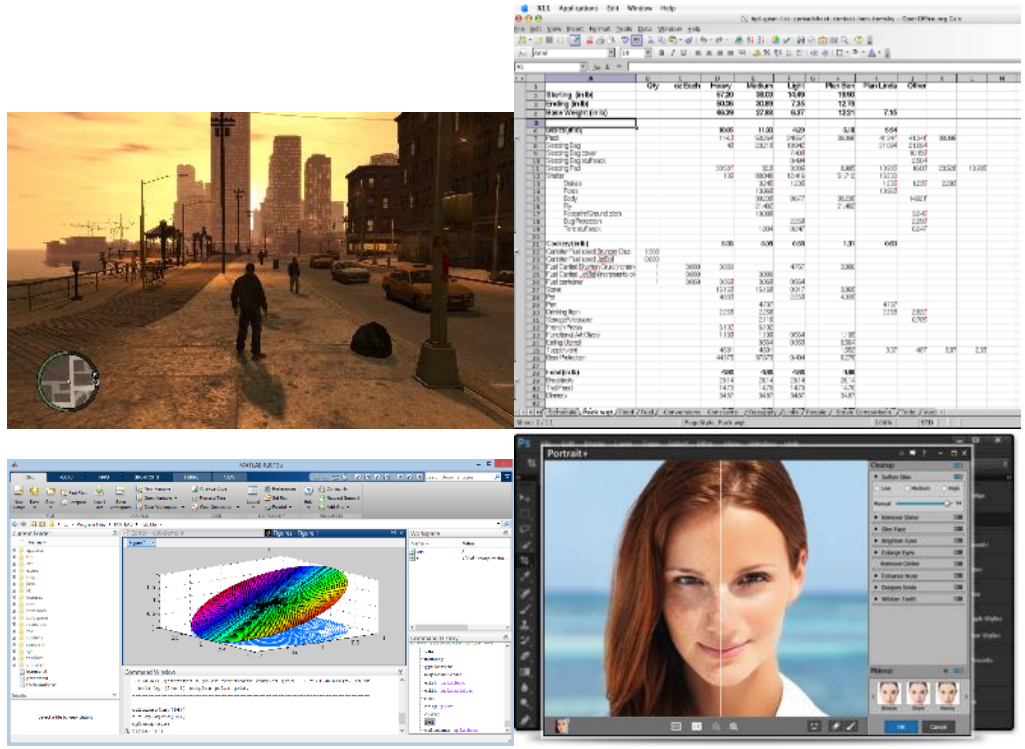
Another Issue

- The stack is very limited. How can you use a lot more available memory for heavier applications?



Another Issue

- The stack is very limited. How can you use a lot more available memory for heavier applications?
  - Hint - you may have done so using Java or C# in the past



The Heap

- Earlier in the module, your learned about the stack.

The Heap

- Earlier in the module, your learned about the stack.
  - The stack is a small amount of memory managing function calls and automatic variables.

The Heap

- Earlier in the module, your learned about the stack.
  - The stack is a small amount of memory managing function calls and automatic variables.
- The rest of the addressable memory is conventionally called the heap.

The Heap

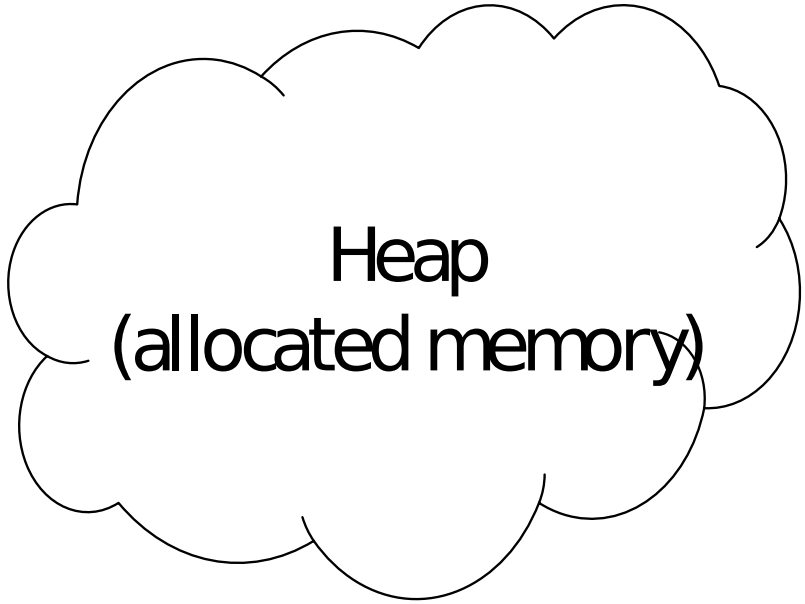
- Earlier in the module, your learned about the stack.
  - The stack is a small amount of memory managing function calls and automatic variables.
- The rest of the addressable memory is conventionally called the heap.
  - Memory can be allocated and deallocated dynamically, when required.

The Heap

- Earlier in the module, your learned about the stack.
  - The stack is a small amount of memory managing function calls and automatic variables.
- The rest of the addressable memory is conventionally called the heap.
  - Memory can be allocated and deallocated dynamically, when required.
- It is usually limited to how much virtual memory your system can handle (32 or 64 bits).

Stack and Heap

Stack (working memory)



Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.
    - *The default* stack size in a Windows is 1 MB

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.
  - *The default* stack size in a Windows is 1 MB
  - With Linux it is a bit dependant, but 8 MB is a considered norm

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.
  - *The default* stack size in a Windows is 1 MB
  - With Linux it is a bit dependant, but 8 MB is a considered norm
  - You can set it yourself when building the application

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.
  - *The default* stack size in a Windows is 1 MB
  - With Linux it is a bit dependant, but 8 MB is a considered norm
  - You can set it yourself when building the application
- To access data allocated in the heap, we need a reference in the stack. In C and C++, it is a pointer.

Principles

- Data in the stack has very a limited size and a limited scope. It is considered working memory.
- The size limit of data in the heap depends on the platform. The programmer retains control on its lifecycle and scope. Its use is for larger data and longer term.
  - *The default* stack size in a Windows is 1 MB
  - With Linux it is a bit dependant, but 8 MB is a considered norm
  - You can set it yourself when building the application
- To access data allocated in the heap, we need a reference in the stack. In C and C++, it is a pointer.
  - In C# and Java memory is also allocated on the heap using the `new` keyword

Questions?

Outline

Dynamic Allocation

- Using the heap is commonly called *dynamic allocation*. It is called so because:

Dynamic Allocation

- Using the heap is commonly called *dynamic allocation*. It is called so because:
  - The memory is allocated at runtime

Dynamic Allocation

- Using the heap is commonly called *dynamic allocation*. It is called so because:
  - The memory is allocated at runtime
  - The size of data allocated can vary (e.g. photos, sound, collections, etc.).

Dynamic Allocation

- Using the heap is commonly called *dynamic allocation*. It is called so because:
  - The memory is allocated at runtime
  - The size of data allocated can vary (e.g. photos, sound, collections, etc.).
- Compiled languages provide means to dynamically allocate data, using different keywords.

Dynamic Allocation - Single Value

Allocating Memory in C

```
// Needed for malloc
#include <stdlib.h>

int *number = (int*)malloc(sizeof(int));
*number = 5;

// Using the malloc(size) function where size is a
    number of bytes
// Note the need to cast malloc
```

Dynamic Allocation - Single Value

Allocating Memory in C

```
// Needed for malloc
#include <stdlib.h>

int *number = (int*)malloc(sizeof(int));
*number = 5;

// Using the malloc(size) function where size is a
    number of bytes
// Note the need to cast malloc
```

Allocating Memory in C++

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

// C++ introduces the keyword 'new'
// Note we don't have to define the number of bytes
```

Dynamic Allocation - Array

Allocating Memory in C

```
#include <stdlib.h>
int *array= (int*)malloc(10*sizeof(int));

// The dereferenced pointer is the first index of the
    array.
```

Dynamic Allocation - Array

Allocating Memory in C

```c
#include <stdlib.h>
int *array= (int*)malloc(10*sizeof(int));

// The dereferenced pointer is the first index of the
    array.
```

Allocating Memory in C++

```c
int *array = new int[10];

// We can use the array syntax
```

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`
- In C++, we would use the following syntax:

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`
- In C++, we would use the following syntax:
  - `delete pointer_to_heap;`

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`
- In C++, we would use the following syntax:
  - `delete pointer_to_heap;`
  - `delete[] pointer_to_heap;`

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`
- In C++, we would use the following syntax:
  - `delete pointer_to_heap;`
  - `delete[] pointer_to_heap;`
- It is good practice to set your pointer to 0 or `NULL` (or `nullptr` in modern C++) afterwards.

Cleaning Up

- Unless we are using a *garbage-collected* language (like Java or C#), you need to deallocate memory manually too.
- In C, we would use the following syntax:
  - `free(pointer_to_heap);`
  - `free` is also part of `stdlib.h`
- In C++, we would use the following syntax:
  - `delete pointer_to_heap;`
  - `delete[] pointer_to_heap;`
- It is good practice to set your pointer to 0 or `NULL` (or `nullptr` in modern C++) afterwards.
- *Warning* - you need to make sure to keep a reference to the data alive until after you've cleaned up. More on next slide.

Question

1. You create a new scope for a function

Question

1. You create a new scope for a function
2. You create a pointer to the heap

Question

1. You create a new scope for a function
2. You create a pointer to the heap
3. You set the pointer to memory allocated in the heap

Question

1. You create a new scope for a function
2. You create a pointer to the heap
3. You set the pointer to memory allocated in the heap
4. You exit the function without deallocating.

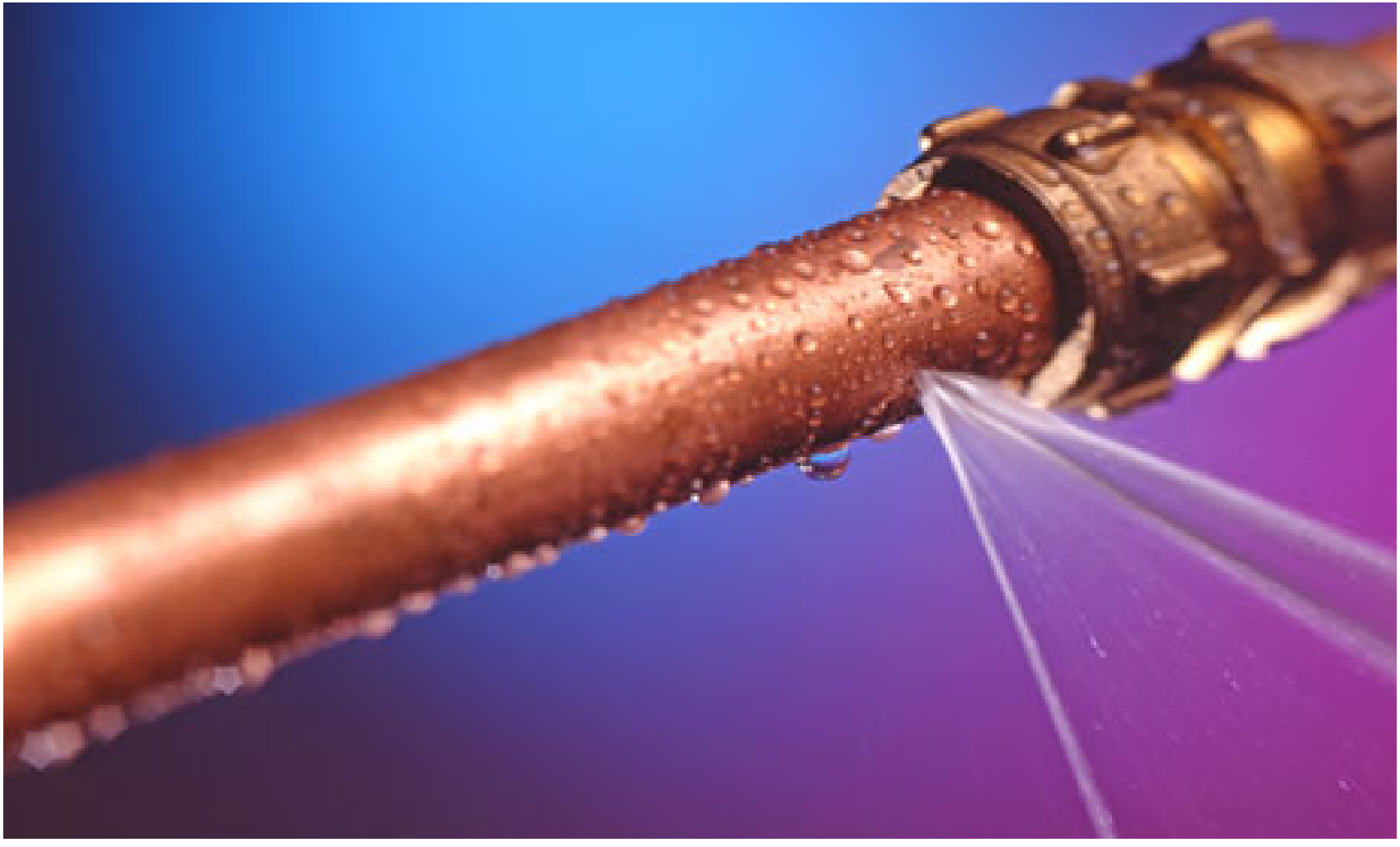
Question

1. You create a new scope for a function
2. You create a pointer to the heap
3. You set the pointer to memory allocated in the heap
4. You exit the function without deallocating.

- What just happened?

Answer

- Congratulations! You've created your first memory leak!



Memory Leaks

- Memory leaks happen when memory is allocated in the heap and the stack reference to the data in heap is lost.

Memory Leaks

- Memory leaks happen when memory is allocated in the heap and the stack reference to the data in heap is lost.
  - There is no way to retrieve the data and free it.

Memory Leaks

- Memory leaks happen when memory is allocated in the heap and the stack reference to the data in heap is lost.
  - There is no way to retrieve the data and free it.
- It can happen when you forget to keep the reference alive from function to function, or by forgetting to call `free`/`delete` when you stop using the data.

Memory Leaks

- Memory leaks happen when memory is allocated in the heap and the stack reference to the data in heap is lost.
  - There is no way to retrieve the data and free it.
- It can happen when you forget to keep the reference alive from function to function, or by forgetting to call `free`/`delete` when you stop using the data.
- This is a very common bug, still in many commercial applications.

Question

1. You use a pointer to allocate memory dynamically

Question

1. You use a pointer to allocate memory dynamically
2. Later in the scope, you deallocate the memory using `free`/`delete` and set your pointer to `NULL`.

Question

1. You use a pointer to allocate memory dynamically
2. Later in the scope, you deallocate the memory using `free`/`delete` and set your pointer to `NULL`.
3. Still in the same scope, later on, you try to access the memory referenced by the pointer.

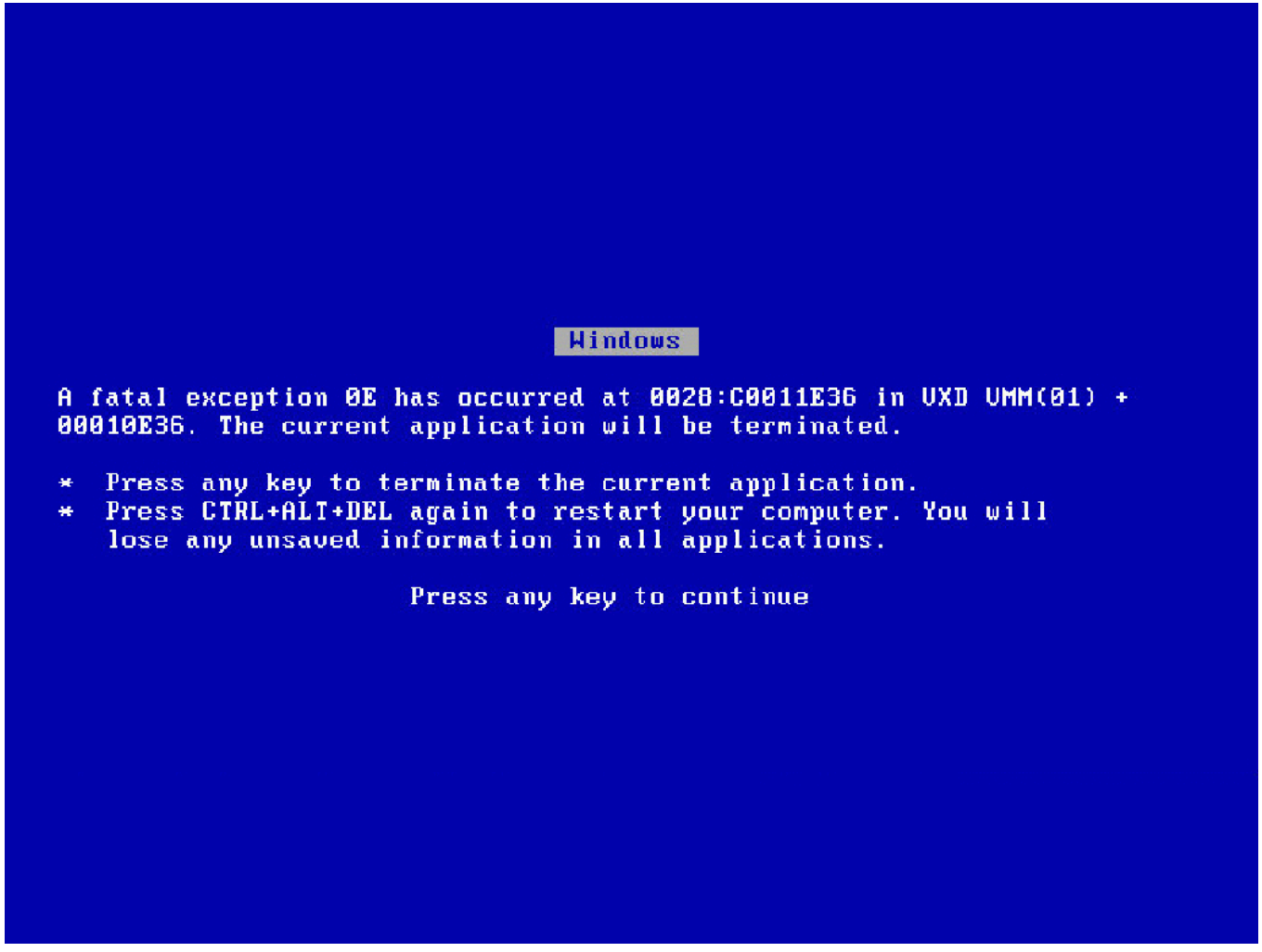
Question

1. You use a pointer to allocate memory dynamically
2. Later in the scope, you deallocate the memory using `free`/`delete` and set your pointer to `NULL`.
3. Still in the same scope, later on, you try to access the memory referenced by the pointer.

- What happens?

Answer

- Congratulations! You just crashed your system!



Segmentation Fault

- You just caused a segmentation fault by trying to access a null pointer, or memory in the heap that you are not allowed to access.

Segmentation Fault

- You just caused a segmentation fault by trying to access a null pointer, or memory in the heap that you are not allowed to access.
- This is actually a mechanism for protecting memory by the operating system.

Segmentation Fault

- You just caused a segmentation fault by trying to access a null pointer, or memory in the heap that you are not allowed to access.
- This is actually a mechanism for protecting memory by the operating system.
- This is also a common mistake for beginners. The compiler will not see them. You will need to be careful!

Questions?

Outline

Memory Management Elsewhere

- Manual memory management is still very much specific to C and C++. There are other methods adopted by other languages:

Memory Management Elsewhere

- Manual memory management is still very much specific to C and C++. There are other methods adopted by other languages:

  Garbage collection Java or C# have a background process called the garbage collector, it checks for memory which has no reference left and cleans it up.

Memory Management Elsewhere

- Manual memory management is still very much specific to C and C++. There are other methods adopted by other languages:

  Garbage collection Java or C# have a background process called the garbage collector, it checks for memory which has no reference left and cleans it up.

  - Downside: it has a performance cost. A process runs in the background performing garbage collection

Memory Management Elsewhere

- Manual memory management is still very much specific to C and C++. There are other methods adopted by other languages:

  Garbage collection Java or C# have a background process called the garbage collector, it checks for memory which has no reference left and cleans it up.

  - Downside: it has a performance cost. A process runs in the background performing garbage collection

  Reference counting Objective-C (Apple) has a reference counting system for each object. When it reaches zero, the object is deleted. The process is now automated at compile-time.

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

  `shared_ptr` A shared pointer (accessible in multiple scopes - deleted when no scopes access it).

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

  `shared_ptr` A shared pointer (accessible in multiple scopes - deleted when no scopes access it).

  `unique_ptr` A unique pointer (accessible in only one scope - deleted when no scope owns it).

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

  `shared_ptr` A shared pointer (accessible in multiple scopes - deleted when no scopes access it).

  `unique_ptr` A unique pointer (accessible in only one scope - deleted when no scope owns it).

  `auto` New keyword to enable easy use of smart pointers.

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

  `shared_ptr` A shared pointer (accessible in multiple scopes - deleted when no scopes access it).

  `unique_ptr` A unique pointer (accessible in only one scope - deleted when no scope owns it).

  `auto` New keyword to enable easy use of smart pointers.

- These new sort of smart pointers introduce the notion of reference counting to C++. Instead of allocating and deallocating memory.

- The newest C++ standard, C++ 11, has introduced new types of pointers, making memory management easier:

  `shared_ptr` A shared pointer (accessible in multiple scopes - deleted when no scopes access it).

  `unique_ptr` A unique pointer (accessible in only one scope - deleted when no scope owns it).

  `auto` New keyword to enable easy use of smart pointers.

- These new sort of smart pointers introduce the notion of reference counting to C++. Instead of allocating and deallocating memory.
  - Pointers responsible for allocating and releasing ownership of memory in the heap.

- The cleanup is automatic when all the shared pointers have released ownership.

- The cleanup is automatic when all the shared pointers have released ownership.
- ... or when a unique pointer has done so.

- The cleanup is automatic when all the shared pointers have released ownership.
- ... or when a unique pointer has done so.
- Examples of the new C++ 11 syntax and memory management will be practiced in the lab. They require a basic understanding of object-oriented syntax... that we will teach you!

Questions?

Outline

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap
  - Understanding when a value is on the stack or in the heap is important

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap
  - Understanding when a value is on the stack or in the heap is important
  - Also recall the discussion around pointers and references last time

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap
  - Understanding when a value is on the stack or in the heap is important
  - Also recall the discussion around pointers and references last time
- We have also looked at how we allocate and release memory

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap
  - Understanding when a value is on the stack or in the heap is important
  - Also recall the discussion around pointers and references last time
- We have also looked at how we allocate and release memory
  - `malloc` and `free` in C

Summary

- We have spent a lot of time discussing scope, and what it means for a value to be in scope
  - Scope is an important idea to wrap your head around - *spend the time on it!*
- We have also talked a lot about the stack and heap
  - Understanding when a value is on the stack or in the heap is important
  - Also recall the discussion around pointers and references last time
- We have also looked at how we allocate and release memory
  - `malloc` and `free` in C
  - `new` and `delete` in C++

To do...

- In the lab: memory management, including C++ 11 and a case study, building a tree.

To do...

- In the lab: memory management, including C++ 11 and a case study, building a tree.
- Next week - no lecture. You should spend the week focusing on your coursework, so we are introducing nothing new. Ensure you attend the labs and get help if you need it. It is what the lab is there for. If you don't ask for help, we cannot provide help. There are no silly questions - we are here to help you learn the material.