Dynamic Memory Management

COSC1076
Semester 1 2019

Week 03

RMIT UNIVERSITY

Admin

- Assignment 1
  - Discussed today
  - Due, end of week 5
  - Search the forum to see if your questions have already been asked
- Extra-Help Session
  - 1-hour sessions, BOYD
  - Times, announced on course forum
  - Register interest
    - If not enough attendance, will be cancelled

Typedef

Typedef

- A ***typedef*** is a user-defined type that is a ***synonym*** for another type
- Generally, typedefs are used for:
  - Program Clarity
  - Abstraction
  - Truncation

New Type name

```c
typedef int* IntPtr;
```

Existing Type

Pointers & References

The most important topic in the course!

RMIT UNIVERSITY

Computer Memory

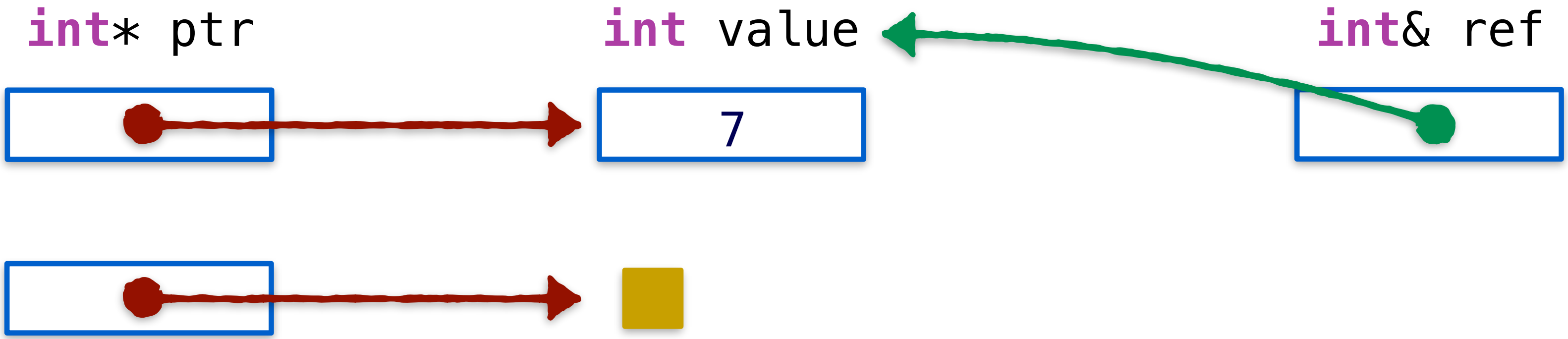
- Each ***byte*** has a unique ***address***
  - This is how a computer can find a piece of memory
  - Addresses are stored in hex, and adjacent memory locations are sequential

Pointer	Name	Memory			
0x0004fca4	integer:	00000000	00000000	00000000	00000111
0x0004fca8	character:	01100100			
0x0004fca9	decimal:				

Pointers & References

- A reference type is denoted in syntax using a ‘&’

```
int  value = 7;
int* ptr = &value;
int& ref = value;
ptr = NULL;
```



Classes

Classes Declaration

- C++ Class ***Declaration***

Public, Protected, & Private Scopes

```cpp
class Example {
public:
    Example(int value);

    void publicMethod();

protected:

    int protectedVariable;
    int protectedMethod(double param);

private:

    double privateArray[LENGTH];
    void privateMethod(int* ptr, double& ref);

};
```

Class Name

Constructor

Method

Field

Fields can be arrays or other classes

Don't forget ';'

Class Method Definitions

- C++ Class method definitions provide the implementation of each method
  - Definitions provided individually
  - Scope is not relevant to the definition
  - The Class name creates a namespace!

Return Type

Class Name

Method Name

Parameters

```
int Example::protectedMethod(double param) {
    return 0;
}
```

Namespace separator

Class Initialisation

- Objects (variables of a given class) can be created like any other variable
  - Does not need to be “new’ed”
- The constructor is called when defining the variable
  - Use bracket notation to provide the parameters to a class object

No return type

Constructor

Parameter

```
Example::Example(int value) {
   protectedVariable = value;
}
```

Object definition

```
Example ex(10);
```

Class is a type

variable/object name

Constructor Parameters

Access Class Members

- Class members (variables and methods) are accessed using dot '.' Syntax

```
Example ex(10);
ex.publicMethod();
```

- For pointers to object, arrow syntax '->' is a shortcut for dereferencing

```
Example* ptrEx = &ex;
(*ex).publicMethod();
ex->publicMethod();
```

- Class members can only be accessed from the correct scope
  - Public members are always accessible
  - Private members are only accessible only from within the class
  - Protected members can be accessed from this class and all children

Class & Functions

- Pass classes to functions either by
  - Pointer
  - Reference
- Passing the class directly:
  - Is possible
  - BUT!
    - Requires a special constructor (called a copy constructor)
    - We will cover this in future week(s)

Arrays are …. pointers?

Arrays in Memory

- Unsurprisingly in memory, an array is a ***sequence of adjacent memory cells***
  - The address of each array location is based on the arrays type

```
int array[10]
```

Address						
0x0004fca4	array[0]:	00000000	00000000	00000000	00000111
0x0004fca8	array[1]:	00000000	00000000	00010101	00010101
0x0004fcab	array[2]:	00000000	00000000	00000000	1001110
...	...	...	...	...	...

Arrays in Memory

- How does an array actually work?
  - That is, how does a compiler or program access the correct cell in memory?
  - Static access:

```c
int array[10] = {1};
array[0] = 2;
```

  - Dynamic access:

```c
int i = 2;
array[i] = -1;
```

Arrays as Pointers

- An array is actually an ***abstraction for using pointers***!
- The actual array “variable” is a pointer to the first element of the array
- The square-bracket lookup notation is short-hand for:
  - Go to the memory location of the array
  - Go to the ‘ith’ memory location from this
  - Dereference that memory address

```
array[3]
```

0x004a	0x004a	0x004a	0x004a	0x004a	0x004a
h	e	l	l	o	!

Arrays as Pointers

- In C/C++, a pointer type is another way to represent an array

```
int  array[10]
int* array;
```

  - BUT!
  - The first version actually ***set's aside the memory*** for the array
  - The second ***can be interpreted*** as an array, but only sets aside memory for a single pointer

- The "pointer form" of an array is often used in functions and methods

Arrays with Functions

- Arrays being a pointer is the reason why arrays are passed-by-reference
  - They are actually "pass-by-value on the pointer to the start of the array"

```
void foo(int array[]);
```

- The "pointer form" of an array is often used in functions and methods

```
void foo(int* array);
```

  - Both of these function prototypes achieve the same functionality

Multi-Dimensional Arrays

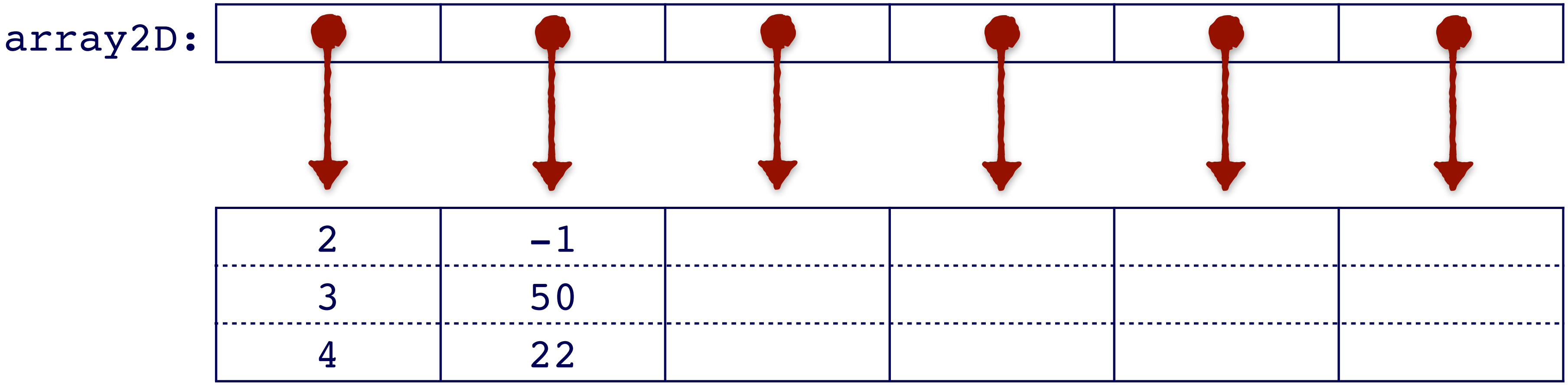
- Multi-dimensional arrays are ***“pointers-to-pointers”***
  - They use multiple star’s for short-hand

```
int array2D[ROWS][COLS];
int** array2D;
```

Multi-Dimensional Arrays

- The best way to think of them, is as an ***"array-of-arrays"***
  - At the first layer, is an array
  - Each element of this is another array

```
int array2D[ROWS][COLS];
```



Multi-File Programs

RMIT UNIVERSITY

File Types

- C/C++ has two types of files:
  - Header files (.h / .hpp)
  - Source files (c.pp)
-

Header Files

- Header files contain definitions, such as
  - Function prototypes
  - Class descriptions
  - Typedef's
  - Common #define's
- Header files do not contain any code implementation
- The purpose of the header files is to describe to source files all of the information that is required in order to implement some part of the code

Source Files

- Code files have source code definitions / implementations
  - In a single combined program, every function or class method may only have a single implementation
- To successfully provide implementations, the code must be given all necessary declarations to fully describe all types and classes that are being used
  - Definitions in header files are included in the code file

```
#include “header.h”
```

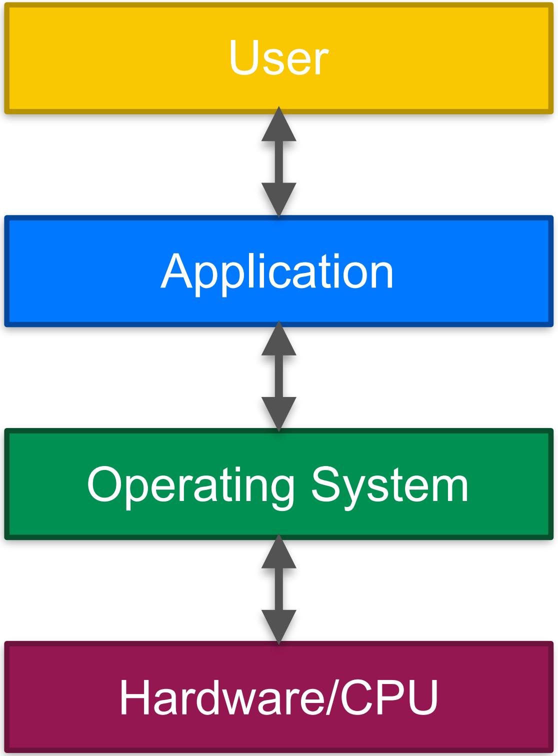
  - For local header files, use double-quotes
  - Use *relative-path* to the header file from the code file

Program Memory Management

How does a program manage memory?

- We have informally discussed things like:
  - The program "setting aside memory"
  - Declaring a variable, "creates memory"
  - Seen array overflow, that is, "reading beyond the end of an array"
  - Mentioned that the "operating system does stuff"

- The question, is what is happening?



Application Structure

- There are two general components to an application/program that we are concerned with:
  - Program code loaded into computer memory
  - Allocated memory for storing program data, such as
    - Variables
    - Function parameters
    - Program control information

Program Code

- Each line of code is converted into assembly instructions
- An assembly instruction is a single operation that the CPU can execute
- To run a program, the assembly instructions are loaded into memory
  - Thus, every instruction has an associated memory address
  - The operating system uses an ***instruction pointer*** (memory address) to track the instruction in a program that it is up to

Instruction Pointer

0x004a	ADD x y
0x004b	SUB x b
0x004c	JUMP 0x004a
...	...

Memory Structure

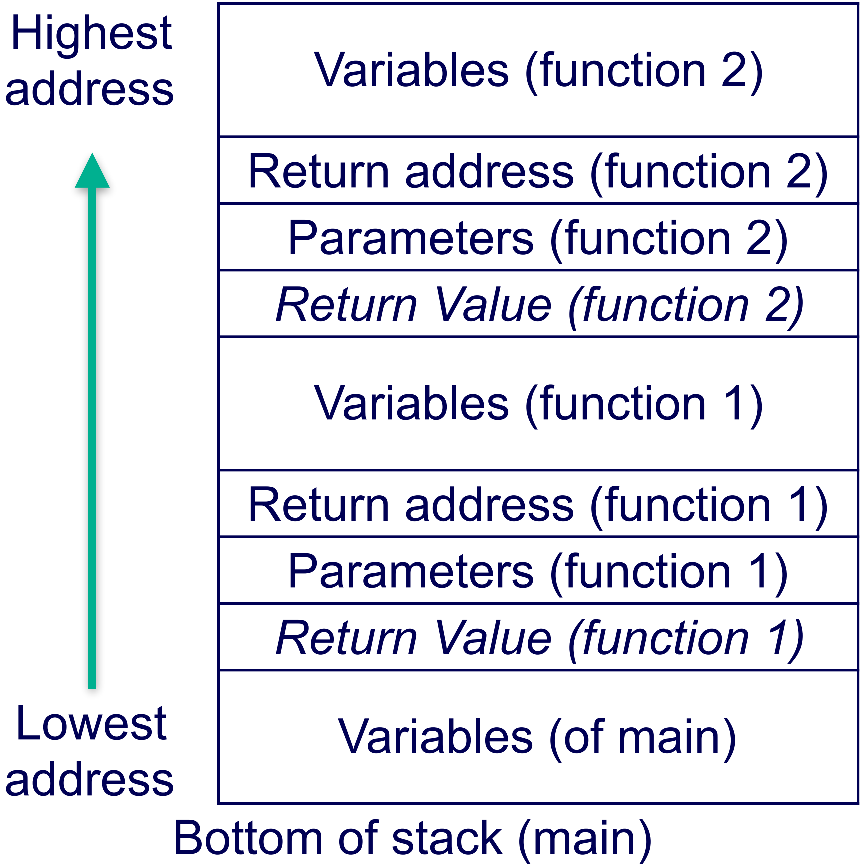
- In a typical program, memory is managed in two forms
  - Automatic memory allocation
  - Dynamic, programmer controlled, memory allocation
- Automatic memory allocation is managed through the programming language complier (C/C++) or interpreter (Java)
  - The typical method for this is the ***Program (or Call) Stack***
- Dynamic memory allocation, is maintained by the programmer through the programming language
  - This is typically done on the ***Heap*** (or ***Free Store***)
- Operating Systems and CPUs provide

Program/Call Stack

- The C++ compiler automatically handles allocating and de-allocating of memory for variables and function calls.
  - The compiler generates CPU instructions for memory management
  - A data structure called a ***stack***
- In the stack
  - As memory is required, a block (of the correct size) is allocated by being ***pushed*** onto the stack
  - Once memory is no-longer required, blocks are de-allocated by being ***popped*** off the stack
  - This forms an ordered structure, using FILO (first-in, last-out)
  - That is, the first allocated block is the last to be de-allocated

Program/Call Stack

- A stack is used, because of how memory is allocated
- Memory allocation has two components
  1. Variable allocation
  2. Function call



Program/Call Stack - Local Variables

```c
int main (void) {
    int a = 7;
    int b = 8;
    int c = 9;
    int d = 10;
    int array[2] = {11, 12};

    int* ptr = &a;

    return EXIT_SUCCESS;
}
```

ptr	0x00….. (&a)
array[0]	11
array[1]	12
d	10
c	9
b	8
a	7

Bottom of stack (main)

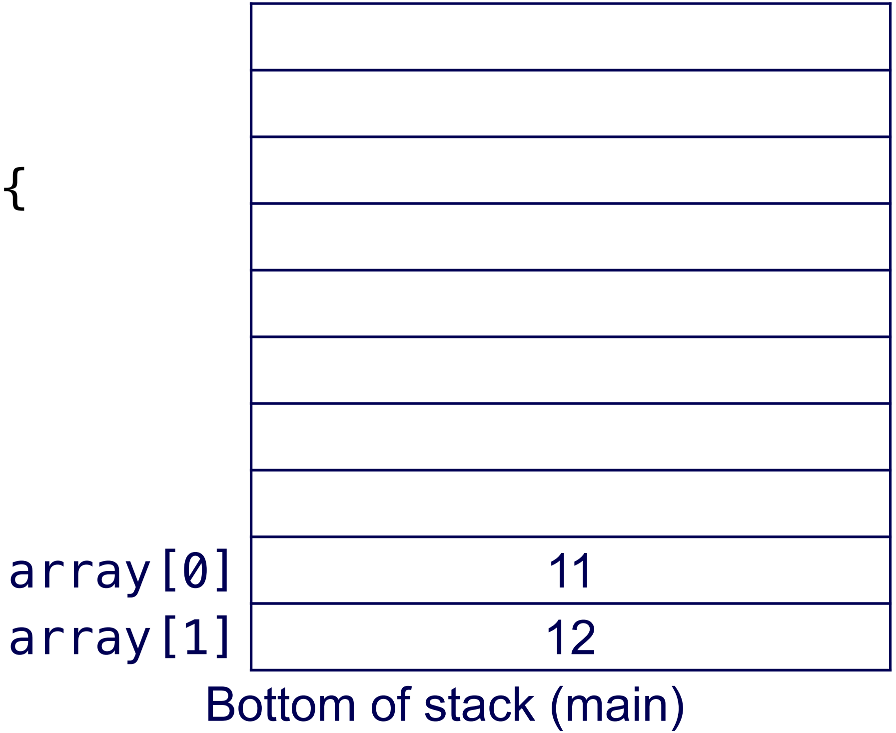
Program/Call Stack - Scoping

```
int main (void) {
   int array[2] = {11, 12};

   for(int i = 0; i < 2; ++i) {
      cout << array[i] << endl;
   }

   int a = 10;

   return EXIT_SUCCESS;
}
```



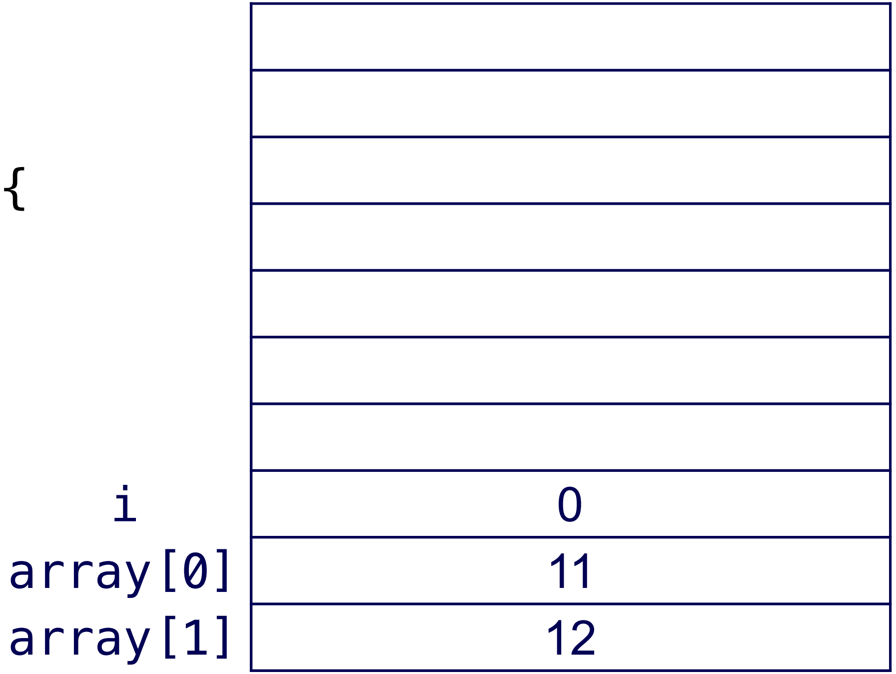
Program/Call Stack - Scoping

```
int main (void) {
   int array[2] = {11, 12};

   for(int i = 0; i < 2; ++i) {
      cout << array[i] << endl;
   }

   int a = 10;

   return EXIT_SUCCESS;
}
```



Bottom of stack (main)

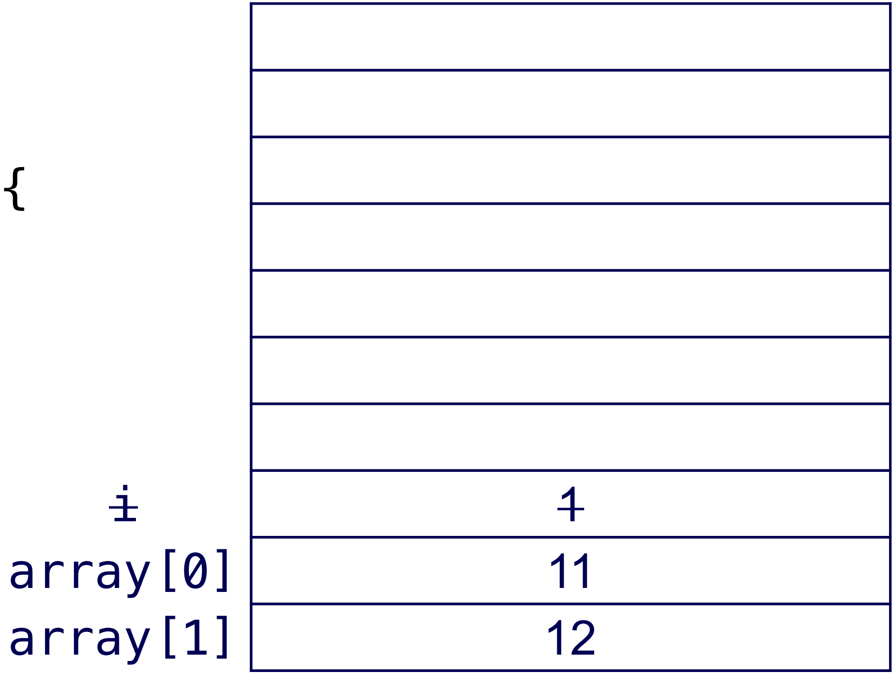
Program/Call Stack - Scoping

```
int main (void) {
   int array[2] = {11, 12};

   for(int i = 0; i < 2; ++i) {
      cout << array[i] << endl;
   }

   int a = 10;

   return EXIT_SUCCESS;
}
```



Bottom of stack (main)

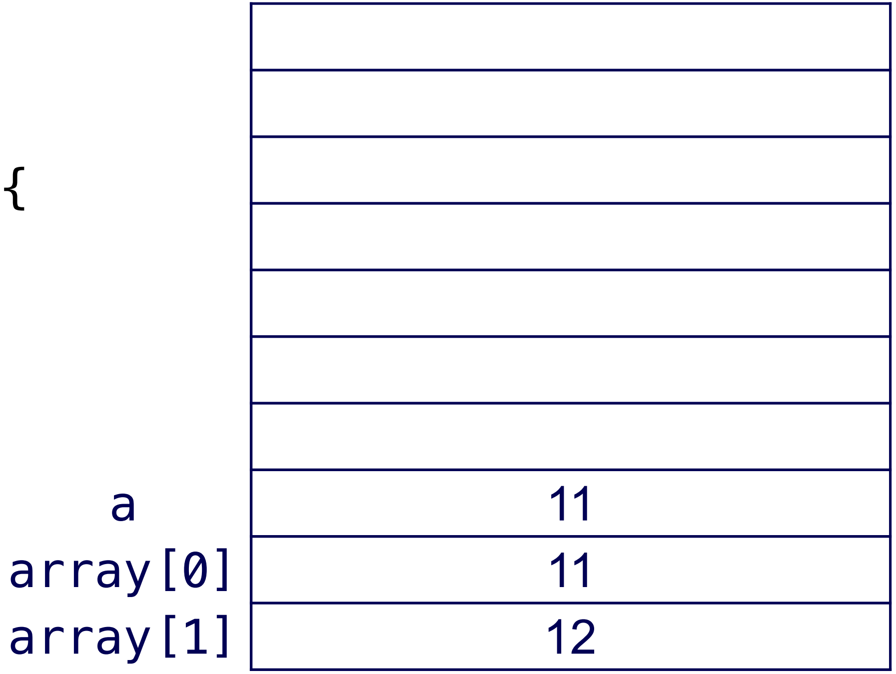
Program/Call Stack - Scoping

```
int main (void) {
   int array[2] = {11, 12};

   for(int i = 0; i < 2; ++i) {
      cout << array[i] << endl;
   }

   int a = 10;

   return EXIT_SUCCESS;
}
```



Bottom of stack (main)

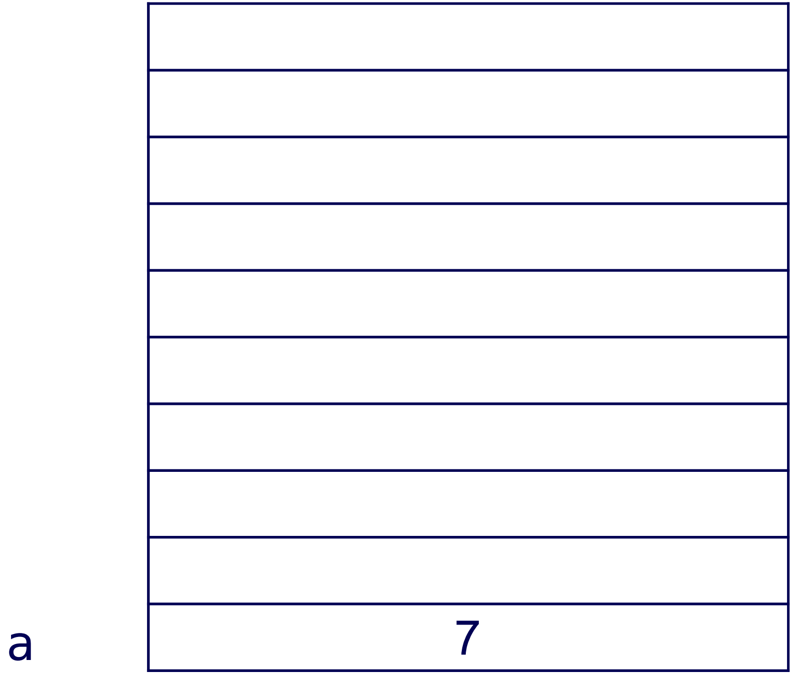
Program/Call Stack - Function Call

```
int main (void) {
    int a = 7;

    int b = 0;
    b = foo(a);

    return EXIT_SUCCESS;
}

int foo(int x) {
    int y = 3;
    return x + y;
}
```



Bottom of stack (main)

Program/Call Stack - Function Call

```c
int main (void) {
    int a = 7;

    int b = 0;
    b = foo(a);

    return EXIT_SUCCESS;
}

int foo(int x) {
    int y = 3;
    return x + y;
}
```

y	3
ret addr	0x001a… (return to main)
x	7
ret val	?
b	0
a	7

Bottom of stack (main)

Program/Call Stack - Function Call

```c
int main (void) {
    int a = 7;

    int b = 0;
    b = foo(a);

    return EXIT_SUCCESS;
}

int foo(int x) {
    int y = 3;
    return x + y;
}
```

y	3
ret addr	0x001a… (return to main)
x	7
ret val	10
b	0
a	7

Bottom of stack (main)

Program/Call Stack - Function Call

```c
int main (void) {
    int a = 7;

    int b = 0;
    b = foo(a);

    return EXIT_SUCCESS;
}

int foo(int x) {
    int y = 3;
    return x + y;
}
```

~~y~~	~~3~~
~~ret addr~~	~~0x001a… (return to main)~~
~~x~~	~~7~~
ret val	10
b	0
a	7

Bottom of stack (main)

Program/Call Stack - Function Call

```
int main (void) {
    int a = 7;

    int b = 0;
    b = foo(a);

    return EXIT_SUCCESS;
}

int foo(int x) {
    int y = 3;
    return x + y;
}
```

~~y~~	~~3~~
~~ret addr~~	~~0x001a… (return to main)~~
~~x~~	~~7~~
~~ret val~~	~~10~~
b	10
a	7

Bottom of stack (main)

Program/Call Stack

- The operating system determines and manages the location of the program stack in physical memory. The OS may
  - Limit the total size of the stack
  - Randomly position the stack
  - Clear (set to zero) all stack memory locations

Heap (Free Store)

- The programmer manages the allocation and de-allocation of memory on the ***heap***
- In Java, objects are allocated on the heap

```
Object obj = new Object();
```

  - The "new" keyword creates the object
  - The object is stored on the heap

Heap - De-allocation

- Any memory that is allocated, should be ***de-allocated***
  - Also called ***"freeing"*** or ***"deleting"*** memory
  - By de-allocating memory, a program can re-use it for another purpose
  - If memory is not "cleaned-up", the Operating System is not aware that memory is no longer needed
    - Any new memory that a program requires, must be allocated elsewhere
- Java has automated garbage collection, so the programmer does not have to be concerned with de-allocating the memory
  - In C++, ***You (the programmer) MUST*** de-allocate all memory
- Only when a program is terminated, will the operating system reclaim all memory that was not de-allocated

Call Stack vs Heap

- The operating system determines the size and location in physical memory of:
  - Program Store
  - Call Stack
  - Heap
- Generally, the heap is significantly larger than the call stack
  - The call stack should be used to store small, short-lived, scoped data
  - The heap is good for storing data that is not bounded within the scope of a single function or method
- In Java:
  - Objects are placed on the Heap
  - Local variable are placed on the stack
- In C++, we will operate with a similar model

Dynamic Memory Management

RMIT UNIVERSITY

Dynamic Memory Management

- Dynamic Memory Management is about managing memory on the heap

- The key principle: ***Everything you create, you must delete***

Allocating Memory on the Heap

- In C/C++ memory is allocated on the heap using the ‘new’ keyword

```
int* a = new int;
```

Returns a pointer

“new” memory

Type of memory to generate

- The “new” returns a pointer to the allocated memory
- The allocated memory is not initialised
  - This can be done separately
- The compiler/OS will set aside enough memory based on the type

Allocating Memory on the Heap

- The allocated memory may also be initialised inline, using “bracket” notation

```
int* a = new int(7);
```

Initialise

- Use of this is dependent on the type
  - Some types do not support inline allocation and initialisation.
- As with all other memory, the allocated memory must initialised!

Deleting Memory on the Heap

- Memory is de-allocated using the "delete" keyword

```
delete a;
```

"delete" memory   delete on pointer

- Delete is called on the pointer (not dereference)
- Once a pointer has been deleted, that memory location cannot be used
- Best practice is to set any deleted pointer to NULL

Issues with Memory Management

- Memory Leaks
  - If you loose a pointer to any allocated memory, you can’t get it back
  - That memory is lost, and cannot be re-allocated
  - The only way to get the memory back is to terminate the program
  - This is the largest cause of programs using up too much memory
- Double delete
  - Deleting the same location twice, results in a memory error
- Delete on NULL
  - Not possible
  - May cause an error on some operating systems - so be careful!

Primitive Types

- All primitive types can be allocated using the type keyword.

```
int* a = new int;
double* d = new double;
char* c = new char;
```

- Primitive types can be initialised inline.

```
int* a = new int(7);
double* d = new double(7.5);
char* c = new char('a');
```

- Delete operates as already shown

```
delete a;
```

Pointers to other Types

- Pointers to types can be allocated

```
int** a = new int*;
double** d = new double*;
char** c = new char*;
```

  - Do not forget the use double reference
- Pointers can be initialised inline.
- Delete operators as per usual

```
delete a;
```

Arrays

- Arrays can be allocated using square brackets

```
int* array = new int[LENGTH];
```

- Arrays cannot be initialised inline
- A special delete operator is required

```
delete[] array;
```

  - This does not need to be given the length
    - This is because of how "`new[]`" is implemented in C++
    - Strictly this is an operator

Multi-dimensional Arrays

- A Multi-Dimensional array cannot be allocated in a single statement
  - This is a limitation of C++ and the way in which arrays are represented
- Requires a multi-step allocation process:
  1. Allocate the first array dimension, as pointers

```
int** array2d = new int*[LENGTH];
for (int i = 0; i != LENGTH; ++i) {
   array2d[i] = new int[LENGTH/2];
}
```

- Delete requires a similar multi-step process

Classes - allocating

- Allocating memory for a Class ***creates an object*** of that class

```
Example* ex(10);
```

- Creating an object calls a constructor of the class
  - A constructor must always be called
  - Even if that constructor is empty (takes no parameters)

```
Example* ex();
```

  - If a class defines no constructors, C++ will generate a default empty constructor

Classes - deallocating

- When an object is deleted, a special method is called
  - This method is the ***deconstructor***
  - The deconstructor is denoted with a tilde (~)

- The deconstructor must clean up all memory, objects, or entities that are used by the object
  - If the object has allocated it’s own memory, this should be deleted
  - An object failing to clean-up after itself is one of the most common causes of memory leaks!

Classes - deallocating

- When an object is deleted, a special method is called
  - This method is the ***deconstructor***
    - The deconstructor is denoted with a tilde (~)
    - It always takes no parameters

```
class Example {
public:
    Example(int value);
    ~Example();
};
```

Deconstructor

```
Example* ex(10);
delete ex;
```

Calls deconstructor

```
Example::~Example() {
    // cleanup
}
```

Deconstructor Implementation

Assignment 1

RMIT UNIVERSITY