Cache and Memory Exercises

Create a Solution

Create a new Blank Solution (on the desktop) to contain all of the projects for this course. For the various exercises add new projects to this solution.

Cache Exercise

Create a new project to define a constant cache which will illustrate the principle by 'storing' a large number of squares. However, these values will be calculated when required.

- Create a new Console Project called CacheExercise.
- Add a source file with main function (if necessary).
- Add a class call 'SquareCache' with a data member for an array of 10000 integers.
- Add a constructor to initialise the array values with -1.
- Add a 'const' member function 'getSquare' taking an integer index. Within this member function index into the array (member) to return a square. If the value is not currently with the array, calculate the square (store with the array) and return the square.
- Within the main function make use of the 'SquareCache' to request a number of squares and output.
- Define a function to output a number of squares, passing the 'SquareCache' as a constant reference. Call this function from within main.

Build and test.

Memory Exercise

Create a new project to allocate objects using class specific new and delete.

- Create a new Console Project called MemoryExercise.
- Add a source file with main function (if necessary).
- Add a class call ‘DataClass’ holding two ‘int’ data members. Add a constructor to initialise these data members.
- Add member operator functions new and delete to allocate and free ‘DataClass’ objects.
- Within the main function dynamically allocate a number of ‘DataClass’ objects and ensure these are deleted. Check using the debugger that the operators are called.

Build and test.

Memory Pool

This optional exercise is to provide a memory pool to allocate memory for the ‘square’ values.

- Create a new class called MemoryPool.
- To the MemoryPool add data members for a pointer to character (_data) and a pointer to integer (_next).
- Within the constructor allocate memory and assign to the pointer to character. Initialise the _next to _data.
- Add a member function ‘getNextInt’ returning the address of the next available integer.
- Create a MemoryPool for the SquareCache. Use this MemoryPool to allocate an integer for each square (using placement new).

Build and Test.

Templates (Type Deduction)

Create a new project to investigate template type deduction and templates functions.

- Create a new project called ‘TemplateDemos’. Add a source file.

- To the source file add a simple template function taking a template parameter (T).
- Investigate type deduction by calling this function with a variable of type ‘int’, ‘int&’ and ‘const int&’.

- What is the type passed into the function?

- To the source file add a simple template function taking a template parameter (T&).

- Investigate type deduction by calling this function with a variable of type ‘int’, ‘int&’ and ‘const int&’. What is the type passed into the function?

- What is the type passed into the function?

- To the source file add a simple template function taking a template parameter (const T&).
- Investigate type deduction by calling this function with a variable of type ‘int’, ‘int&’ and ‘const int&’.

- What is the type passed into the function?

Template Function Specialisation

- Add a template function to calculate the maximum of two values (myMax).

- Call the myMax function with a number two values of various different type (int, double, etc.).

- Try calling the myMax function with string literals. Does the myMax function work?

- Implement a specialisation function taking ‘const char*’

Build and test.

Option: Template Stack

Stack Class

Create a console project to create and use a stack of ‘int’. Add a ‘stack’ class to store ‘int’ on a last in/first out basis. Implement push and pop member functions. Use a ‘vector’ to provide the underlying implementation. How should the pop member function behave with an empty stack? Should an ‘isEmpty’ member function be provided? Should a ‘peek’ member function be provided (reads value without popping)? Should the pop actually return a value? Should any of the member functions throw exceptions?
The above questions are essentially aspects of design and should be decided before implementation and then implement consistently with the expected design. When implementing the stack ensure ‘const’ correctness.
To the main function add code to push and pop ‘int’ onto the stack.

Stack Template

Add a console project to use a template stack. The previous implementation of the stack stored ‘int’. Create a template version of the stack which can store the type defined by the ‘parameter type’.
To the main function add code to push and pop objects onto various instantiations of the stack.

Template Conditional Selection

Create a template structure to conditionally (on bool) select between two template parameter types.

- Use a type aliasing to set this type (using TheType<true, int, double>::Value).

Build and test.

Variadic Template Functions

Create a new console project to experiment with variadic templates.

- Create a project called VariadicDemos.
- Add a source file.
- Define a variadic template function to sum an arbitrary number of values.

Build and test.

Dynamic Allocation

Change the cache to use dynamic memory allocation.

- Modify the array to be an array of pointers to integer.
- Initialise these pointers to ‘nullptr’.
- Within the ‘getSquare’ method instatiate an integer to hold the square value;

Build and test.

Are there any problems with this code? Memory leak? What can be done to resolve any problems?

Smart Pointer

Change the cache to use smart pointers rather than raw pointers.

- Modify the array to be an array of shared pointers to int (shared_ptr).
- Within the ‘getSquare’ method create a shared pointer the square.

Build and test.

Smart Pointer Exercise (Cyclic Dependencies!)

Create a new project to use a number of smart pointer types.

- Create a new Console Project called SmartPointerExercise.
- Add a source file with main function (if necessary).
- Add a class called ‘SomeData’ with a data member pointer to a class called ‘MoreData’ (user forward declaration for ‘MoreData’). Add a constructor to ‘SomeData’ to initialise the ‘MoreData’ pointer member.
- Add a class called ‘MoreData’ with a data member pointer to the class ‘SomeData’. Add a member function ‘setSomeData’ to initialise the pointer member.
- Within the main function dynamically allocate a ‘MoreData’ object. Dynamically allocate a ‘SomeData’ object initialising it with the ‘MoreData’ pointer. Use the ‘setSomeData’ to set the pointer of the ‘MoreData’ object.

Build and test.

Are there any problems with this code? Memory leaks?

- Wrap the pointers using ‘shared_ptr’ (#include <memory>).

Build and test.
Are there any problems with this code? Memory leaks?

- Use a ‘weak_ptr’ instead of ‘shared_ptr’ for wrapping the ‘MoreData’ pointer with the ‘SomeData’ class.

Build and test.

Confirm this is working correctly by adding output to destructors of both the ‘SomeData’ and ‘MoreData’ types.

Range Based For / Containers and Iterators

Create a new project called ForContainerIterator.

- Create a class to act as a container ‘MyContainer’ with a member of type integer pointer.
- Add member functions to the ‘MyContainer’ called ‘begin’ and ‘end’ to return an iterator (MyIterator).
- Within the ‘main’ method declare a MyContainer object and use a range based for loop to iterate over this container. Output to the console values from the container.
- Create a class to act as an iterator ‘MyIterator’ with a member of type reference to a ‘MyContainer’.
- For the iterator implement the operators !=, ++ and *.

Build the application.

Implement any remaining functionality to output value to the console.

Build and test the application.

