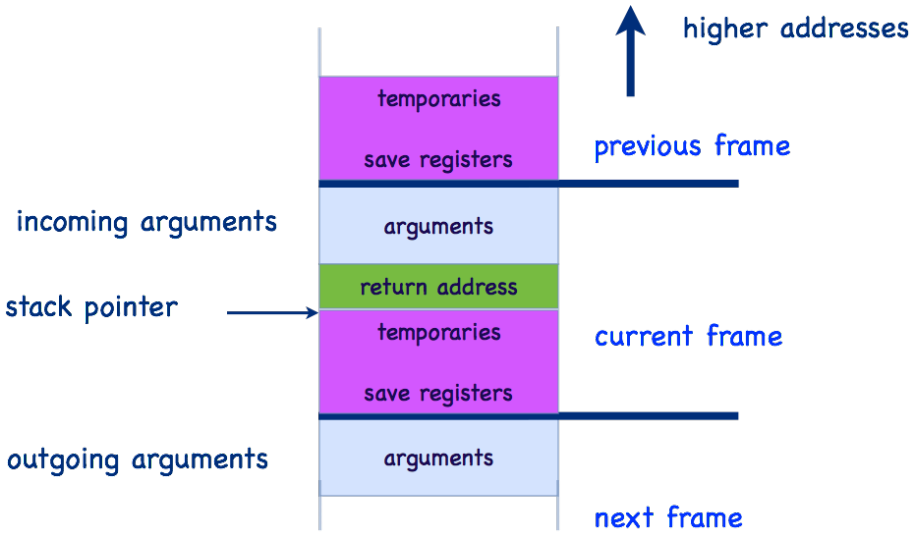
Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	9, ADT & Copy Control
Lecturer:	Ms. Siti Hawa

Any sufficiently advanced technology is indistinguishable from magic.
Arthur C. Clarke



Lab 9: ADT & Copy Control

In this tutorial, we study the complete implementation of `Stack`, an abstract data type with proper copy control. A stack automatically adjusts its size to the number of elements it has to store. The size of a stack can change in two ways: it can increase and it can decrease. We use a standard heuristic for the changes: we double the size when we increase the stack size and we half the size when we decrease the stack size. This allows the amortized costs of all stack operations to remain O(1).

To control the size changes, we employ a load factor. Initially, an empty stack has load factor 0 (the stack contains one free slot). If a stack is full, then its load factor is 1. We double the stack size in this case. This results in a stack with a load factor ½. When removing elements from the stack, the load factor may eventually decrease to ½ also. However, shrinking the stack size in this instance is not wise. This could result in an undesired effect. We half the stack size when decreasing its size. If the load factor is ½ initially, then after halving its size the load factor is 1. The next operation can be a push, which requires an immediate increase of the size by doubling it. This process can repeat. All benefits of controlling expansion and contraction are lost. Hence, rather than contracting the space at ½ we let the load factor decrease to ¼. A halving of the space at load factor ¼ results in a new load factor of ½. This is the same value as for expansion. Furthermore, there is sufficient space in the stack to accommodate more elements before it has to be expanded again.

In addition to the standard stack operations, we also define "emplace", a technique that constructs stack elements at their location in the stack. This operation frees us form the need to construct elements before they are copied unto the stack. Emplace uses perfect forwarding to identify the most suitable constructor for the object we wish to store on the stack. The interface of emplace can be confusing as it takes constructor arguments.

Finally, we define proper copy control for our abstract data type. Copy control guarantees the proper management of heap memory and avoids any memory leaks. Copy control requires a destructor, copy constructor, copy assignment operator, move constructor, move assignment operator, and a swap function. The compiler can synthesize some or all of those features, but we can rarely on the synthesized features, if at all, when our objects maintain heap memory. Even though copy control is quite involved, there exist standard techniques to define it.

Template Class Stack

We follow the example shown in class:

```
template<typename T>
class Stack
{
private:
  T* fElements;
  size_t fStackPointer;
  size_t fCurrentSize;

  void resize( size_t aNewSize );
  void ensure_capacity();
  void adjust_capacity();

public:

  Stack();
  ~Stack();

  Stack( const Stack& aOther );
  Stack& operator=( const Stack<T>& aOther );

  Stack( Stack<T>&& aOther ) noexcept;
  Stack& operator=( Stack<T>&& aOther ) noexcept;

  void swap( Stack& aOther ) noexcept;

  size_t size() const noexcept;

  std::optional<T> top() noexcept;
  void push( const T& aValue );

  template<typename... Args>
  void emplace( Args&&… args );

  void pop();
};
```

Implement `Stack` in four stages:

1. Basic operations
    a. Stack()
    b. ~Stack()
    c. size()
    d. top()
    e. push()
    f. pop()
    g. resize(), ensure_capacity(), and adjust_capacity()
2. Emplace
3. Copy Semantics
4. Move Semantics

The test driver provided for this tutorial task makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

The test driver (i.e., main.cpp) uses `P1`, `P2`, `P3`, and `P4` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
// #define P3
// #define P4
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX` … **`#endif`** are grayed out, if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX` … **`#endif`** enables conditional compilation. XCode does not use this color coding scheme.

The tests push 30 elements and pop them. Every test has to end with "success". Copy and move tests have two success criteria.