CHAPTER 3

STACKS
A stack is a version of a list that is particularly useful in applications involving reversing, such as the problem will be given later on. In a stack data structure, all insertions and deletions of entries are made at one end, called the top of the stack.

A helpful analogy is to think of a stack of trays or of plates sitting on the counter in a busy cafeteria. Throughout the lunch hour, customers take trays off the top of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The bottom tray is the first one put on, and the last one to be used.

When we add an item to a stack, we say that we push it onto the stack, and when we remove an item, we say that we pop it from the stack.

Note that the last item pushed onto a stack is always the first that will be popped from the stack. This property is called last in, first out, or LIFO for short.

A stack is particularly useful in applications involving reversing such as the problem at the end of this chapter.

Shown in the following figure are the effects of push and pop operations on the stack.

![Stack operations](image)

Indicates the stack top position

Figure: Stack operations.

The STL Stack Implementation
/* Program stackSTL.cpp
Pre: The user supplies an integer n and n decimal numbers.
Post: The numbers are printed in reverse order.
Uses: The STL class stack and its methods
*/
#include <iostream>
#include <stdio.h>
using namespace std;
#include <stack>
int main() {
    stack <int> numbers;
    cout << numbers.empty() << endl;
    numbers.push(10);
    numbers.push(20);
    numbers.push(30);

    cout << numbers.empty() << endl;
    cout << endl;

    while (!numbers.empty()) {
        cout << numbers.top() << endl;
        numbers.pop();
    }
    getchar();
}

Output:
1
0
30
20
10

In the program stackSTL.cpp the empty() member function returns a true if the stack is empty. Otherwise it returns a false.

User-Defined Implementation of a Stack - Creating a Stack in C++

//Program stackUserDefined.cpp
#include <stdio.h>
#include <iostream>
using namespace std;
#define DEFAULT_SIZE 10

class Stack{
private:
    int size;
    int top;
    int* values;
public:
    Stack(int size = DEFAULT_SIZE);
    virtual ~Stack();
    bool isFull();
    bool isEmpty();
    void push(int);
    int pop();
}
Stack::Stack(int size) { //constructor
    this->size = size;
    values = new int[size];
    top = -1;
}

Stack::~Stack() { //destructor
    delete[] values;
}

bool Stack::isFull()
{
    if(top < size-1)
    
        return false;
    
    else
    
        return true;
}

bool Stack::isEmpty()
{
    if(top == -1)
    
        return true;
    
    else
    
        return false;
}

void Stack::push(int x)
{
    if(!isFull())
    
        top++;
    
        values[top] = x;
}

int Stack::pop()
{
    int retVal = 0;
    if(!isEmpty())
    
        retVal = values[top];
    
        top--; 
    
        return retVal;
}

int main()
{
    Stack *stack = new Stack;
You can create a stack in C++ by defining a stack class and declaring an instance of the class. The Stack class requires three attributes and several member functions. You begin by defining a basic stack class that has only the components needed to create the stack.

The class is called Stack, but you can call it any name you wish. This class definition is divided into a private access specifier section and a public access specifier section. The private access specifier section has attributes and member functions (although not in this example) that are accessible only by a member function defined in this class. The public access specifier section has attributes (although not in this example) and member functions that are accessible by using an instance of the class.

The private access specifier section of the Stack class defines three attributes: size, top, and values, all of which are integers. The size attribute stores the number of elements in the stack, the top attribute stores the index of the top element of the stack, and the values attribute is a pointer to the stack, which is an array. The stack in this example is a stack of integer values, but you can use an array of any data type, depending on the nature of your program.

The Stack constructor receives an integer as an argument that is passed when the instance of the Stack class is declared. The integer determines the number of elements in the stack and is assigned to the size variable.

The first statement might look a bit confusing. It appears that the value of the size variable from the argument list is being assigned to itself, but that’s not the case. Actually, the size variable from the argument list is local to the Stack member function. The this->size combination refers to the size attribute of the Stack class, as shown here:

```
this->size = size;
```
Programmers use the this pointer within a member function to refer to the current instance of the class. In this example, the this pointer uses the pointer reference (->) to tell the computer to use the size attribute of the class. As you’ll remember from your C++ programming class, the pointer reference is used when indirectly working with a class member, and the dot operator is used when you are directly working with a class member.

This allows the compiler to distinguish between a class variable and local variable that have the same name. This means that the value of the size variable that is passed as an argument to the Stack member function is assigned to the size attribute, making the value available to other members of the Stack class.

You can see how the size attribute is used in the next statement. This statement does two things. First, it allocates memory for the stack by using the new operator (new int[size]). The new operator returns a pointer to the reserved memory location. The size is the size attribute of the class and determines the size of the array. The array is an array of integers.

Next, the pointer to the array of integers is assigned to the values attribute of the class. The values attribute is a pointer variable that is defined in the private attribute section of the Stack class.

The last statement in the Stack constructor assigns the value –1 to the top attribute. The value of the top attribute is the index of the top element of the stack and -1 for the top attribute means that that stack doesn’t have any elements (the stack is empty).

Now that you’ve seen how to define a class that creates a stack and now you define additional member functions that enable the class to push values onto the stack. Pushing a value onto the stack is a two-step process. First, you must determine if there is room on the stack for another value. If there is, you push the value onto the stack; otherwise, you don’t.

The member function isFull() determines if there is room on the stack. The isFull() member function is simple. It compares the value of the top attribute with the one less than the value of the size attribute.

The value of the top attribute is –1 when the instance of the stack is declared. Suppose the value of size is 10. The condition expression in the if statement of the isFull() member function determines if the value of top, which is –1, is 1 less than the value of size. Since the value of size is 10, the condition expression compares –1 < 9. If top is greater than or equal to 9, then a true is returned; otherwise, a false is returned.

Why do you subtract 1 from the size of the stack? The value of the top attribute is an index of an array element. Remember that the index begins with zero. In contrast, the size is actually the number of array elements in the stack. Therefore, the tenth array element on the stack has an index of 9.
The push() member function pushes a value onto the stack. The value being pushed onto the stack is passed as an argument to the push() member function and is assigned to the variable \( x \).

We still need a way to remove values from the stack. To do this, we need to define two additional member functions, isEmpty() and pop(). The isEmpty() member function determines if there are any values on the stack. The pop() member function removes the value from the top of the stack.

The isEmpty() member function contains an if statement. The condition expression of the if statement compares the value of the top attribute to –1. Remember that –1 is the initial value of the top attribute when the instance of a stack is declared. If the top attribute is equal to –1, then a true is returned because the stack is empty; otherwise, a false is returned.

The pop() member function of the Stack class has the job of changing the index that is at the top of the stack and returning the value of the corresponding array to the statement that calls the pop() member function.

**Tip** From your programming classes, you learned to always build error-trapping routines into your program to properly handle errors should they occur. Always include such routines in your stack program. Three common errors to trap are problems allocating memory for the stack, reacting to a full stack, and reacting to an empty stack.

The next member function is ~Stack() and is the destructor of the class. A destructor is the last member function that is called when the instance of the class goes out of scope and dies. A destructor must always be the same name as the class and begin with a tilde (~). By definition, destructors cannot accept any arguments. The purpose of the destructor is to free memory that is used by the stack or do any other sort of cleanup that’s required.

The ~Stack() member function frees memory used by the stack. It does this by using the delete operator and referencing the name of the array used for the stack. In this example, values is the name of the array.

To avoid memory leaks, freeing memory is important whenever memory is dynamically allocated. The square brackets ([ ]) are used with delete because the object being removed from memory was dynamically created.

The stack.cpp is compiled as you would compile any source code. The result is an object file that is joined together with the compiled stackDemo.cpp source code file by the linker to create an executable program called a load module.

Finally, we come to the function main() that creates the instance of the Stack class. The first statement creates the stack in a three-step process. The first step is to use the new operator to allocate space in memory for the Stack class by calling the constructor of the class. The new operator returns the memory location of the stack. The second step is to
declare a pointer called stack. The last step is to assign the memory location returned by the new operator to the stack pointer.

In this example, we used the default size for the stack, which is 10 elements. We can pass the Stack() constructor an integer to change the size of the stack.

The push() member function is called three times. Each time a different value is placed on the stack. Notice that the -> pointer is used instead of the dot operator. You must do this because stack is a pointer to an instance of the class and not the instance itself.

The last portion of main() calls the pop() member three times. Each time a value is removed from the top of the stack and displayed on the screen.

Problem
Write a complete C++ program that implements a stack to solve the following problem:
Read an integer $n$, then read a list of $n$ numbers, and print the list in reverse order.

Solution
First Version:
/* Program stackNumbersSTL.cpp
Pre: The user supplies an integer $n$ and $n$ decimal numbers.
Post: The numbers are printed in reverse order.
Uses: The STL class stack and its methods
*/
#include <iostream>
#include <stdio.h>
using namespace std;
#include <stack>
int main(){
  int n;
  double item;
  stack <double> numbers; // declares and initializes a stack of numbers
  cout << " Type in an integer $n$ followed by $n$ decimal numbers." << endl
       << " The numbers will be printed in reverse order." << endl;
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> item;
    numbers.push(item);
  }
  cout << endl << endl;
  while (!numbers.empty()) {
    cout << numbers.top() << " ";
    numbers.pop();
  }
  getchar();
  getchar();
}
Sample Output:
Type in an integer n followed by n integers.
The entered integers will be printed in reverse order.
5 21 43 65 78 49

49
78
65
43
21

Second Version:
/* Program stackNumbersUserDefined.cpp
 Pre : The user supplies an integer n and n decimal numbers.
 Post: The numbers are printed in reverse order.
 Uses: The user-defined class stack and its methods
 */
#include <stdio.h>
#include <iostream>
using namespace std;
#define DEFAULT_SIZE 10
class Stack{
  private:
    int size;
    int top;
    int* values;
  public:
    Stack(int size = DEFAULT_SIZE);
    virtual ~Stack();
    bool isFull();
    bool isEmpty();
    void push(int);
    int pop();
};

Stack::Stack(int size){
    this->size = size;
    values = new int[size];
    top = -1;
}

Stack::~Stack(){
    delete[] values;
}
bool Stack::isFull()
{
    if(top < size-1)
    {
        return false;
    }
    else{
        return true;
    }
}

bool Stack::isEmpty()
{
    if(top == -1)
    {
        return true;
    }
    else{
        return false;
    }
}

void Stack::push(int x){
    if(!isFull()){
        top++;
        values[top] = x;
    }
}

int Stack::pop(){
    int retVal = 0;
    if(!isEmpty()){
        retVal = values[top];
        top--;
    }
    return retVal;
}

int main(){
    int n;
    int item;
    Stack *numbers = new Stack();

    cout " Type in an integer n followed by n integers." " The entered integers will be printed in reverse order." " cin >> n;

    for (int i = 0; i < n; i++) {
        cin >> item;
        numbers->push(item);
    }
}
cout << endl;

for (int i = 0; i < n; i++) {
    cout << numbers->pop() << endl;
}
getchar();
getchar();
}

Example A complete C program to implement a stack using an array appears here:

```c
#include <stdio.h>
#define MAX 10 /* The maximum size of the stack */
#include <stdlib.h>

void push(int stack[], int *top, int value)
{
    if(*top < MAX){
        *top = *top + 1;
        stack[*top] = value;
    } else{
        printf("The stack is full can not push a value\n");
        exit(0);
    }
}

void pop(int stack[], int *top, int *value)
{
    if(*top >= 0){
        *value = stack[*top];
        *top = *top - 1;
    } else{
        printf("The stack is empty can not pop a value\n");
        exit(0);
    }
}

int main()
{
    int stack[MAX];
    int top = -1;
    int n,value;
    do{
        do{
            printf("Enter the element to be pushed\n");
            scanf("%d",&value);
            push(stack,&top,value);
            printf("Enter 1 to continue\n");
            scanf("%d",&n);
        }while(n == 1);
        printf("Enter 1 to pop an element\n");
        scanf("%d",&n);
        while(n == 1){
            pop(stack,&top,&value);
            printf("The value poped is %d\n",value);
            printf("Enter 1 to pop an element\n");
            scanf("%d",&n);
        }
        printf("Enter 1 to continue\n");
        scanf("%d",&n);
    }while(n == 1);
}
```
Quiz

1. What is a stack?
2. What is the purpose of the push() member method?
3. What is the purpose of the pop() member method?
4. What is the purpose of the isFull() member method?
5. What is the purpose of the isEmpty() member method?
6. What kind of value is assigned to the top attribute?
7. Why is the top attribute initialized to –1?
8. What is the purpose of the keyword private?
9. What is the purpose of the keyword public?
10. What is the difference between a constructor and a destructor?

Answers

1. A stack is the way you groups things together by placing one thing on top of another and then removing them one at a time from the top of the stack.
2. The push() member method places a value onto the top of a stack.
3. The pop() member method removes the value from the top of a stack, which is then returned by the pop() member method to the statement that calls the pop() member method.
4. The isFull() member method determines if there is room for one more value on the stack.
5. The isEmpty() member method determines if a value is at the top of the stack and is called before an attempt is made to remove the value.
6. The value at the top attribute is an index.
7. The top attribute is initialized to –1 because when the attribute is incremented by the push() member method, the new value of the top attribute is zero, which is the index of the first element of the array used to create the stack.
8. The keyword private means that the attribute or member method is accessible only by a member method. The instance of the class cannot directly access a private member of the class.
9. The keyword public means that the attribute or member method is accessible to member methods and from the instance of the class.
10. A constructor is a member method of a class that is called when an instance of the class is declared. A destructor is a member method of a class that is called when the instance of the class falls out of scope.

Homework
Reimplement the program “stackUserDefined.cpp” using templates.
Implementing a Stack in C++ using a Vector
Let’s examine a program, Stack.cpp, that implements a stack using a class called StackV.

```cpp
#include <iostream>
#include <vector>
using namespace std;

class StackX{
private:
    int maxSize; //size of stack vector
    vector<double> stackVect; //stack vector
    int top; //top of stack
public:
    StackX(int s) : maxSize(s), top(-1) //constructor
    {
        stackVect.reserve(maxSize); //size the vector
    }

    void push(double j) //put item on top
    {
        stackVect[++top] = j; //increment top,
    } //insert item

    double pop() //take item from top
    {
        return stackVect[top--]; //access item,
    } //decrement top

    double peek() //peek at top of stack
    {
        return stackVect[top];
    }

    bool isEmpty() //true if stack is empty
    {
        return (top == -1);
    }

    bool isFull() //true if stack is full
    {
        return (top == maxSize-1);
    }
}; //end class StackX
```
```c
int main(){
    StackX theStack(10); //make new stack, size 10
    theStack.push(20); //push items onto stack
    theStack.push(40);
    theStack.push(60);
    theStack.push(80);

    while( !theStack.isEmpty() ) //until it’s empty,
    //delete item from stack
    double value = theStack.pop();
    cout << value << " "; //display it
    getche();
    return 0;
} //end main()
```

**Output**
The main() function creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it’s empty.

Here’s the output:
80 60 40 20

Notice how the order of the data is reversed. Because the last item pushed is the first one popped, the 80 appears first in the output.

This version of the StackX class holds data elements of type double. You can change this to any other type, including object types.

**StackX Class Member Functions**
As in previous programs, the data storage mechanism within the class is a vector. Here it’s called stackVect.

The constructor creates a new stack of a size specified in its argument. The data members of the stack are a variable to hold its maximum size (the size of the vector), the vector itself, and a variable, top, which stores the index of the item on the top of the stack.

The push() member function increments top so it points to the space just above the previous top, and stores a data item there. Notice that top is incremented before the item is inserted.

The pop() member function returns the value at top and then decrements top. This effectively removes the item from the stack; it’s inaccessible, although the value remains in the vector (until another item is pushed into the cell).

The peek() member function simply returns the value at top, without changing the stack.
The `isEmpty()` and `isFull()` member functions return true if the stack is empty or full, respectively. The `top` variable is at -1 if the stack is empty and `maxSize-1` if the stack is full. Following figure shows how `push()` and `pop()` work.

**Error Handling**

There are different philosophies about how to handle stack errors. What happens if you try to push an item onto a stack that’s already full? Or pop an item from a stack that’s empty?

In `stackVector.cpp` we’ve left the responsibility for handling such errors up to the class user.
The user should always check to be sure the stack is not full before pushing a new item:

```cpp
if( !theStack.isFull() )
    theStack.push(item);
else
    cout << "Can’t insert, stack is full";
```

In the interest of simplicity, we’ve left this code out of the main() routine (and anyway, in this simple program, we know the stack isn’t full because it has just been initialized).

We do include the check for an empty stack when main() calls pop().

Many stack classes check for these errors internally, in the push() and pop() member functions. This is the preferred approach. In C++, a good solution for a stack class that discovers such errors is to throw an exception, which can then be caught and processed by the class user.

**Exercise 1: Reversing a Word**
Write a program that reverses a word: When you run the program, it asks you to type in a word. When you press Enter, it displays the word with the letters in reverse order.

**Hint**
A stack is used to reverse the letters. First the characters are extracted one by one from the input string and pushed onto the stack. Then they’re popped off the stack and displayed. Because of its last-in-first-out characteristic, the stack reverses the order of the characters.

**Exercise 2: Demonstrating Recursion and Stacks with Triangular Numbers**
As an example of recursion, let’s examine two approaches to an ancient problem. It’s said that the Pythagoreans, a band of mathematicians in ancient Greece who worked under Pythagoras (of Pythagorean theorem fame), felt a mystical connection with the series of numbers 1, 3, 6, 10, 15, 21, … (where the … means the series continues indefinitely).

Can you find the next member of this series?
The $n^{th}$ term in the series is obtained by adding $n$ to the previous term. Thus the second term is found by adding 2 to the first term (which is 1), giving 3. The third term is 3 added to the second term (which is 3) giving 6, and so on. The numbers in this series are called *triangular numbers* because they can be visualized as a triangular arrangement of objects, shown as little squares in the following figure:
Finding the $n^{th}$ Term Using a Loop

Suppose you wanted to find the value of some arbitrary $n^{th}$ term in the series; say the fourth term (whose value is 10). How would you calculate it? Looking at Figure 11.2, you might decide that the value of any term could be obtained by adding up all the vertical columns of squares.

In the fourth term, the first column has four little squares, the second column has three, and so on. Adding $4+3+2+1$ gives 10.

**Figure:** Triangular number as columns.
The following `triangle()` function uses this column-based technique to find a triangular number. It sums all the columns, from a height of `n` to a height of 1.

```c
int triangle(int n) {
    int total = 0;
    while(n > 0) // until n is 1
    {
        total = total + n; // add n (column height) to total
        --n; // decrement column height
    }
    return total;
}
```

The function cycles around the loop `n` times, adding `n` to total the first time, `n-1` the second time, and so on down to 1, quitting the loop when `n` becomes 0.

**Finding the \(n\)th Term Using Recursion**

The loop approach might seem straightforward, but there’s another way to look at this problem. The value of the \(n\)th term can be thought of as the sum of only two things, instead of a whole series. These are

1. The first (tallest) column, which has the value `n`.
2. The sum of all the remaining columns.

This is shown in the following figure:

```
int triangle(int n) {
    if(n == 1) // base case
        return 1;
    else
        return( n + triangle(n-1) );
}
```

The condition that leads to a recursive function returning without making another recursive call is referred to as the **base case**. It’s critical that every recursive function has a base case to prevent infinite recursion and the consequent demise of the program.
The triangle.cpp Program

Does recursion actually work? If you run the triangle.cpp program, you'll see that it does. This program uses recursion to calculate triangular numbers. Enter a value for the term number, n, and the program will display the value of the corresponding triangular number.

```cpp
#include <iostream>
using namespace std;

//-------------------------------------------------------------
int main(){
    int theNumber;
    int triangle(int);
    cout << “Enter a number: “;
    cin >> theNumber;
    int theAnswer = triangle(theNumber);
    cout << “Triangle=” << theAnswer << endl;
    return 0;
} // end main()
//-------------------------------------------------------------
int triangle(int n){
    if(n==1)
        return 1; \base case
    else
        return ( n + triangle(n-1) );
} // end recursive function triangle()
```

The main() routine prompts the user for a value for n, calls triangle(), and displays the return value. The triangle() function calls itself repeatedly to do all the work.

Here's some sample output:

Enter a number: 1000
Triangle = 500500

If you're skeptical of the results returned from triangle(), you can check them by using the following formula:

\[
\text{n}^{th} \text{ triangular number} = \frac{(n^2+n)}{2}
\]

What the triangle() Function Is Really Doing

Let's modify the triangle() function to provide an insight into what's happening when it executes. We'll insert some output statements to keep track of the arguments and return values:
int triangle(int n){
    cout << “Entering: n=” << n << endl;
    if(n==1)
    {
        cout << “Returning 1” << endl;
        return 1;
    }
    else{
        int temp = n + triangle(n-1);
        cout << “Returning “ << temp << endl;
        return temp;
    }
}

Here’s the interaction when this function is substituted for the earlier triangle() function and the user enters 5:

Enter a number: 5
Entering: n=5
Entering: n=4
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15
Triangle = 15

**Analysis:** Each time the triangle() function calls itself, its argument, which starts at 5, is reduced by 1. The function calls itself again and again until its argument is reduced to 1. Then it returns. This causes an entire series of returns. Each time it returns, it adds the value of n it was called with to the return value from the function it called.

The return values replay the series of triangular numbers, until the answer is returned to main(). Following figure shows how each invocation of the triangle() function can be imagined as being “inside” the previous one.
Recursion Versus Stacks

Often an algorithm is easy to conceptualize as a recursive function, but in practice the recursive approach proves to be inefficient. In such cases, it might be useful to transform the recursive approach into a nonrecursive approach. Such a transformation can often make use of a stack.

There is a close relationship between recursion and stacks. In fact, most compilers implement recursion by using stacks. As we noted, when a function is called, the compiler pushes the function arguments and the return address (where control will go when the function returns) on the stack, and then transfers control to the function. When the function returns, the compiler pops these values off the stack. The arguments
disappear, and control returns to the return address. *In theory, any algorithm that uses recursion can be systematically transformed into one that uses a stack.* In practice, however, it's usually more practical to rethink the algorithm from the ground up, using a stack-based approach instead of a recursive approach.

Following program shows what happens when we do that with the triangle() member function:

```cpp
//Program stackTriangle.cpp
//evaluates triangular numbers, stack replaces recursion
#include <stdio.h>
#include <iostream>
#include <vector>
using namespace std;

class StackX{
private:
    int maxSize; //size of stack array
    vector<int>(stackVect); //stack vector
    int top; //top of stack
public:
    StackX(int s) : maxSize(s), top(-1) //constructor
    {
        stackVect.resize(maxSize);
    }
    void push(int p) //put item on top of stack
    {
        stackVect[++top] = p;
    }
    int pop() //take item from top of stack
    {
        return stackVect[top--];
    }
    int peek() //peek at top of stack
    {
        return stackVect[top];
    }
    bool isEmpty() //true if stack is empty
    {
        return (top == -1);
    }
}; //end class StackX

to main(){
    int theNumber;
    int theAnswer;
    int stackTriangle(int);
    cout << "Enter a number: ";
    cin >> theNumber;
    theAnswer = stackTriangle(theNumber);
    cout << "Triangle= " << theAnswer << endl;
    getchar();
    getchar();
    return 0;
} //end main()
```
int stackTriangle(int number) {
    StackX theStack(10000); // make a big stack
    int answer = 0; // initialize answer
    while (number > 0) // until n is 1,
    {
        theStack.push(number); // push value
        --number; // decrement value
    }
    while ( !theStack.isEmpty() ) // until stack empty,
    {
        int newN = theStack.pop(); // pop value,
        answer += newN; // add to answer
    }
    return answer;
}

Analysis
In this program we use a stack class called StackX. In main() there are two loops. In the first, the numbers from n down to 1 are pushed onto the stack. In the second, they’re removed from the stack and summed. The result is the triangular number for n. Of course, in this program you can see by inspection that you can eliminate the stack entirely and use a simple loop. However, in more complicated algorithms the stack must remain.

Often you’ll need to experiment to see whether a recursive function, a stack-based approach, or a simple loop is the most efficient (or practical) way to handle a particular situation.

Summary
- A **recursive function** calls itself repeatedly, with a different argument value each time.
- Some value of its arguments causes a recursive function to return without calling itself. This is called the **base case**.
- A **triangular number** is the sum of itself and all numbers smaller than itself. (Number means positive integer in this context.) For example, the triangular number of 4 is 10, because $4+3+2+1 = 10$.
- Triangular numbers can be calculated using a recursive function, a simple loop, or a stack-based approach.
- **Any operation that can be carried out with recursion can be carried out with a stack.**
- A recursive approach might be inefficient. If so, it can sometimes be replaced with a simple loop or a stack-based approach.
Quiz
1. Complete the sentence: A recursive function is one that...
2. The value of the eighth triangular number is...
3. True or false: Recursion is used because it is more efficient.
4. What is a base case?
5. True or false: A recursive approach can be replaced with a stack-based approach.

Answers
1. Complete the sentence: A recursive function is one that...
   calls itself.
2. The value of the eighth triangular number is 36
3. True or false: Recursion is used because it is more efficient.
   False. It's usually used because it simplifies a problem conceptually.
4. What is a base case?
   The situation in which a recursive function returns rather than calls itself.
5. True or false: A recursive approach can be replaced with a stack-based approach.
   True.