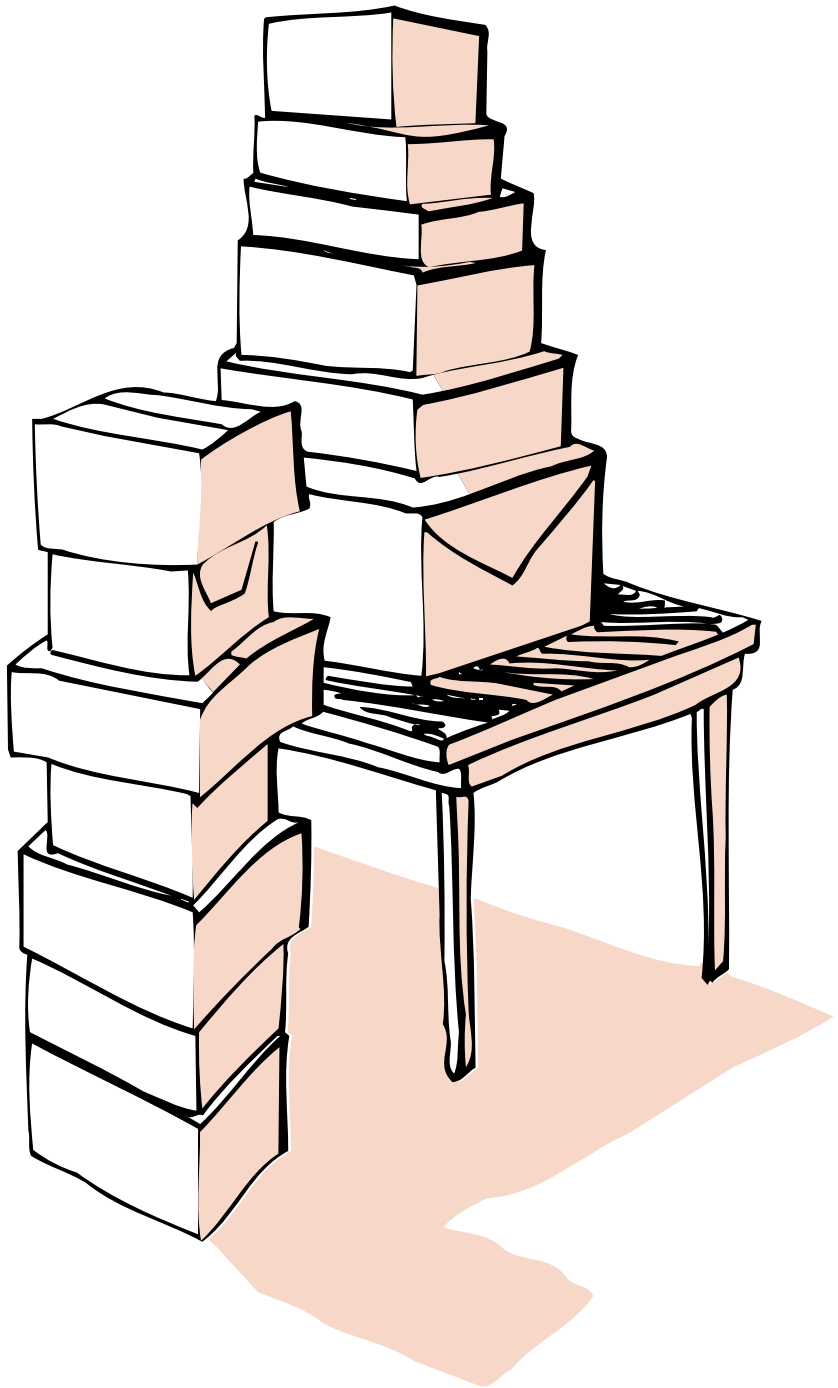
Chapter 2

INTRODUCTION TO STACKS

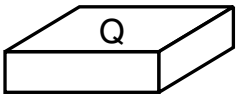
1. Stack Specifications
2. Implementation of Stacks
3. Application: A Desk Calculator
4. Application: Bracket Matching
5. Abstract Data Types and Their Implementations

Stacks

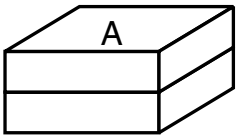
A ***stack*** is a data structure in which all insertions and deletions of entries are made at one end, called the ***top*** of the stack. The last entry which is inserted is the first one that will be removed.



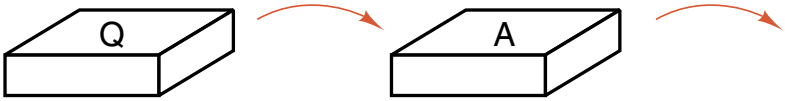
Push box Q onto empty stack:



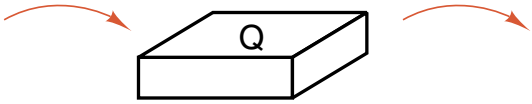
Push box A onto stack:



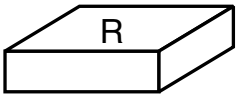
Pop a box from stack:



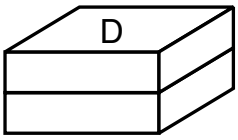
Pop a box from stack: (empty)



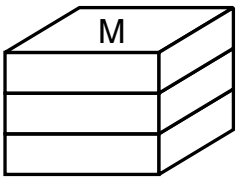
Push box R onto stack:



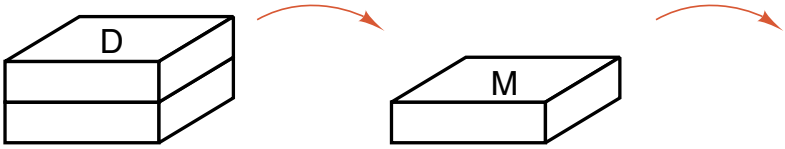
Push box D onto stack:



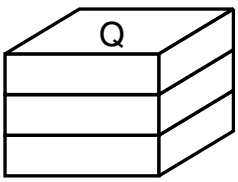
Push box M onto stack:



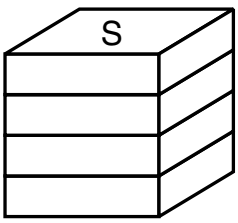
Pop a box from stack:



Push box Q onto stack:



Push box S onto stack:



Standard Template Library (STL)

- The ***standard template library*** (abbreviated ***STL***) in C++ contains much useful information, many functions, and many classes.
- The STL provides convenient implementations for many common data structures, including almost all the data structures we shall study in this book.
- We can include the STL stack implementation into our programs with the directive

```
#include <stack>
```

  and then we can define initially empty stack objects and apply methods called push, pop, top, and empty.
- In C++, a ***template*** construction allows us to create data structures whose entries have different types. Example:

```
stack<double> numbers      and      stack<int> numbers
```

- The STL provides several implementations of various data structures such as stacks.
- The *code* in a client program should not depend on a particular choice of implementation, but the *performance* of the final program may very much depend on the choice of implementation.
- A second, optional template parameter selects the implementation.

First Example: Reversing a List

```
#include <stack>

int main( )
/* 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  */

{
   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( );
   }
   cout << endl;
}
```

Constructor

The ***constructor*** is a function with the same name as the corresponding class and no return type. It is invoked automatically whenever an object of the class is declared.

```
Stack :: Stack( );
```
Pre: None.
Post: The Stack exists and is initialized to be empty.

Entry Types, Generics

For generality, we use Stack_entry for the type of entries in a Stack. A client can specify this type with a definition such as

```
typedef int Stack_entry;
```

or

```
typedef char Stack_entry;
```

The ability to use the same underlying data structure and operations for different entry types is called ***generics***. **typedef** provides simple generics. Starting in Chapter 6, we shall use C++ ***templates*** for greater generality.

Error Processing

We shall use a single enumerated type called Error_code to report errors from all of our programs and functions.

The enumerated type Error_code is part of the utility package in Appendix C. Stacks use three values of an Error_code:

success, overflow, underflow

Later, we shall use several further values of an Error_code.

Programming Precept

After a client uses a class method,
it should decide whether to check the resulting error status.
Classes should be designed to allow clients to decide
how to respond to errors.

C++ also provides more sophisticated error processing called ***exception handling***. The standard library implementations of classes use exception handling, but we shall opt for the simplicity of returning error codes in all our implementations.

Specification for Methods

```
Error_code Stack :: pop( );
```

Pre: None.

Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is empty, an Error_code of underflow is returned and the Stack is left unchanged.

```
Error_code Stack :: push(const Stack_entry &item);
```

Pre: None.

Post: If the Stack is not full, item is added to the top of the Stack. If the Stack is full, an Error_code of overflow is returned and the Stack is left unchanged.

```
Error_code Stack :: top(Stack_entry &item) const;
```

Pre: None.

Post: The top of a nonempty Stack is copied to item. A code of fail is returned if the Stack is empty.

```
bool Stack :: empty( ) const;
```

Pre: None.

Post: A result of **true** is returned if the Stack is empty, otherwise **false** is returned.

Class Specification, Contiguous Stack

We set up an array to hold the entries in the stack and a counter to indicate how many entries there are.

```
const int maxstack = 10;          //  small value for testing

class Stack {
public:
   Stack( );
   bool empty( ) const;
   Error_code pop( );
   Error_code top(Stack_entry &item) const;
   Error_code push(const Stack_entry &item);

private:
   int count;
   Stack_entry entry[maxstack];
};
```

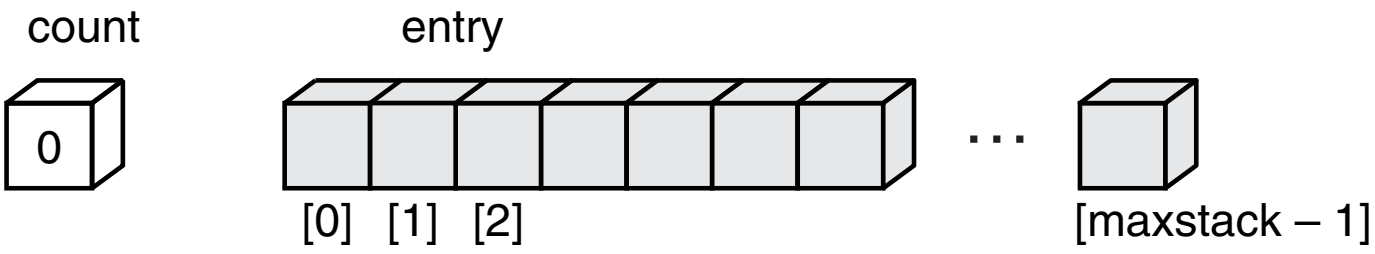
The declaration of **private** visibility for the data makes it is impossible for a client to access the data stored in a Stack except by using the methods push( ), pop( ), and top( ).

Important result: A Stack can never contain illegal or corrupted data. In general, data is said to be ***encapsulated*** if it can only be accessed by a controlled set of functions.

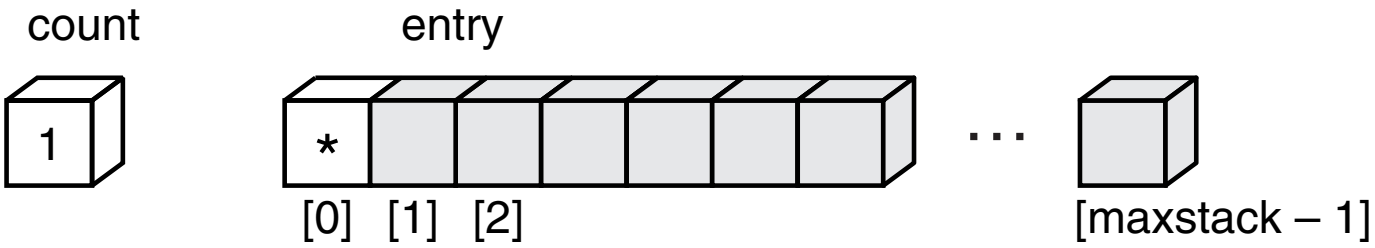
Programming Precept

The public methods for a data structure
should be implemented without preconditions.
The data members should be kept private.

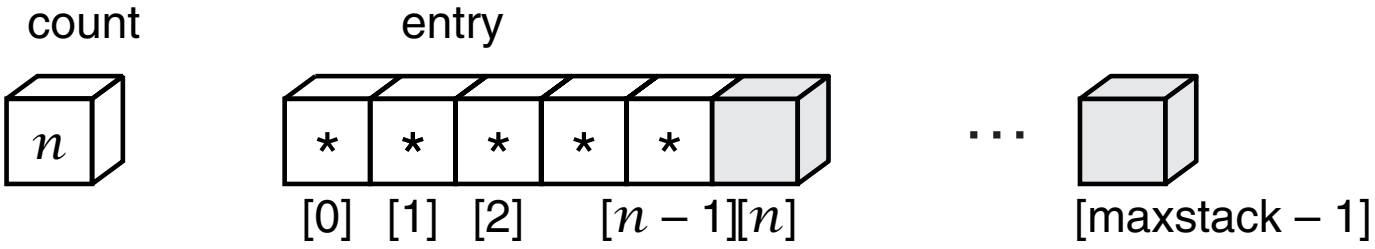
Representation of Data in a Contiguous Stack



(a) Stack is empty.



(b) Push the first entry.



(c) n items on the stack

Stack Methods

```
Error_code Stack::push(const Stack_entry &item)
/* Pre:   None.
   Post:  If the Stack is not full, item is added to the top of the Stack.  If the Stack is
          full, an Error_code of overflow is returned and the Stack is left unchanged.
          */
{
   Error_code outcome = success;
   if (count >= maxstack)
      outcome = overflow;
   else
      entry[count++] = item;
   return outcome;
}
```

```
Error_code Stack::pop( )
/* Pre:   None.
   Post:  If the Stack is not empty, the top of the Stack is removed.  If the Stack is
          empty, an Error_code of underflow is returned.  */
{
   Error_code outcome = success;
   if (count ==  0)
      outcome = underflow;
   else --count;
   return outcome;
}
```

Further Stack Methods

```
Error_code Stack :: top(Stack_entry &item) const
/* Pre:   None.
   Post:  If the Stack is not empty, the top of the Stack is returned in item. If the Stack
          is empty an Error_code of underflow is returned.  */
{
   Error_code outcome = success;
   if (count ==  0)
      outcome = underflow;
   else
      item = entry[count – 1];
   return outcome;
}

bool Stack :: empty( ) const
/* Pre:   None.
   Post:  If the Stack is empty, true is returned. Otherwise false is returned.  */
{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}

Stack :: Stack( )
/* Pre:   None.
   Post:  The stack is initialized to be empty.  */
{
   count = 0;
}
```

Reverse Polish Calculator

In a ***Reverse Polish calculator***, the operands (numbers, usually) are entered *before* an operation (like addition, subtraction, multiplication, or division) is specified. The operands are *pushed* onto a stack. When an operation is performed, it *pops* its operands from the stack and *pushes* its result back onto the stack.

```
typedef double Stack_entry;
#include "stack.h"

int main( )
/* Post:  The program has executed simple arithmetic commands entered by the user.
   Uses:  The class Stack and functions introduction, instructions, do_command,
          and get_command.  */
{
   Stack stored_numbers;
   introduction( );
   instructions( );
   while (do_command(get_command( ), stored_numbers));
}
```

Obtaining a Command

The auxiliary function get_command obtains a command from the user, checking that it is valid and converting it to lowercase by using the string function tolower( ) that is declared in the standard header file `cctype`.

```
char get_command( )
{
   char command;
   bool waiting = true;
   cout << "Select command and press < Enter > :";

   while (waiting) {
      cin >> command;
      command = tolower(command);
      if (command == '?'  || command == '='  || command == '+'  ||
          command == '-' || command == '*'  || command == '/'  ||
          command == 'q') waiting = false;

      else {
         cout  << "Please enter a valid command:"  << endl
               << "[?]push to stack   [=]print top"      << endl
               << "[+] [-] [*] [/]   are arithmetic operations" << endl
               << "[Q]uit." << endl;
      }
   }
   return command;
}
```

```
bool do_command(char command, Stack &numbers)
/* Pre:   The first parameter specifies a valid calculator command.
   Post:  The command specified by the first parameter has been applied to the Stack
          of numbers given by the second parameter.  A result of true is returned unless
          command == 'q'.
   Uses:  The class Stack.  */
{ double p, q;
  switch (command) {
  case '?':
    cout << "Enter a real number: " << flush;  cin >> p;
    if (numbers.push(p) == overflow)
      cout << "Warning: Stack full, lost number" << endl;  break;
  case '=':
    if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
    else cout << p << endl;  break;
  case '+':
    if (numbers.top(p) == underflow) cout << "Stack empty" << endl;
    else {
      numbers.pop( );
      if (numbers.top(q) == underflow) {
        cout << "Stack has just one entry" << endl;
        numbers.push(p);
      }
      else {
        numbers.pop( );
        if (numbers.push(q + p) == overflow)
          cout << "Warning: Stack full, lost result" << endl;
      }
    } break;
  //  Add options for further user commands.
  case 'q':  cout << "Calculation finished.\n";  return false;
  } return true;  }
```

Application: Bracket Matching

- We develop a program to check that brackets are correctly matched in an input text file.
- We limit our attention to the brackets `{`, `}`, `(`, `)`, `[`, and `]`.
- We read a single line of characters and ignore all input other than bracket characters.
- Algorithm: Read the file character by character. Each opening bracket `(`, `[`, or `{` that is encountered is considered as unmatched and is stored until a matching bracket can be found. Any closing bracket `)`, `]`, or `}` must correspond, in bracket style, to the last unmatched opening bracket, which should now be retrieved and removed from storage. Finally, at the end of the program, we must check that no unmatched opening brackets are left over.
- A program to test the matching of brackets needs to process its input file character by character, and, as it works its way through the input, it needs some way to remember any currently unmatched brackets.
- These brackets must be retrieved in the exact reverse of their input order, and therefore a Stack provides an attractive option for their storage.
- Our program need only loop over the input characters, until either a bracketing error is detected or the input file ends. Whenever a bracket is found, an appropriate Stack operation is applied.

Bracket Matching Program

```
int main( )
/* Post:  The program has notified the user of any bracket mismatch in the standard input
          file.
   Uses:  The class Stack.  */
{ Stack openings;
  char symbol;
  bool is_matched = true;
  while (is_matched && (symbol = cin.get( )) != '\n') {
    if (symbol == '{' || symbol == '(' || symbol == '[')
      openings.push(symbol);
    if (symbol == '}' || symbol == ')' || symbol == ']') {
      if (openings.empty( )) {
        cout << "Unmatched closing bracket " << symbol
             << " detected." << endl;
        is_matched = false;
      }
      else {
        char match;
        openings.top(match);
        openings.pop( );
        is_matched = (symbol == '}' && match == '{')
                  || (symbol == ')' && match == '(')
                  || (symbol == ']' && match == '[');
        if (!is_matched)
          cout << "Bad match " << match << symbol << endl;
      }
    }
  }
  if (!openings.empty( ))
    cout << "Unmatched opening bracket(s) detected." << endl;
}
```

Abstract Data Types

DEFINITION A ***type*** is a set, and the elements of the set are called the ***values*** of the type.

Types such as **int**, **float**, and **char** are called ***atomic*** types because we think of their values as single entities only, not something we wish to subdivide.

Computer languages like C++ provide tools such as arrays, classes, and pointers with which we can build new types, called ***structured*** types. A single value of a structured type (that is, a single element of its set) is a structured object such as a contiguous stack. A value of a structured type has two ingredients: It is made up of ***component*** elements, and there is a ***structure***, a set of rules for putting the components together.

DEFINITION A ***sequence of length*** 0 is empty. A ***sequence of length*** $n \geq 1$ of elements from a set T is an ordered pair (S_{n-1}, t) where S_{n-1} is a sequence of length $n-1$ of elements from T, and t is an element of T.

We shall always draw a careful distinction between the word ***sequential***, meaning that the elements form a sequence, and the word ***contiguous***, which we take to mean that the elements have adjacent addresses in memory. Hence we shall be able to speak of a *sequential* list in a *contiguous* implementation.

Abstract Stacks

The definition of a finite sequence allows us to define a ***list*** of items of a type T as a finite sequence of elements of the set T.

Next we wish to define a stack. But there is nothing regarding the sequence of items to distinguish a stack from a list. The primary characteristic of a stack is the set of *operations* or *methods* by which changes or accesses can be made. Including a statement of these operations with the structural rules defining a finite sequence, we obtain:

DEFINITION A ***stack*** of elements of type T is a finite sequence of elements of T, together with the following operations:

1. *Create* the stack, leaving it empty.
2. Test whether the stack is *Empty*.
3. *Push* a new entry onto the top of the stack, provided the stack is not full.
4. *Pop* the entry off the top of the stack, provided the stack is not empty.
5. Retrieve the *Top* entry from the stack, provided the stack is not empty.

Refinement of Data Specification

1. On the *abstract* level we decide how the data are related to each other and what operations are needed, but we decide nothing concerning how the data will actually be stored or how the operations will actually be done.
2. On the *data structures* level we specify enough detail so that we can analyze the behavior of the methods and make appropriate choices as dictated by our problem.
3. On the *implementation* level we decide the details of how the data structures will be represented in computer memory.
4. On the *application* level we settle all details required for our particular application.

The first two levels are often called ***conceptual*** because at these levels we are more concerned with problem solving than with programming. The middle two levels can be called ***algorithmic*** because they concern precise methods for representing data and operating with it. The last two levels are specifically concerned with ***programming***.

Programming Precept

Let your data structure your program.
Refine your algorithms and data structures at the same time.

Programming Precept

Once your data are fully structured,
your algorithms should almost write themselves.

Pointers and Pitfalls

1. Use data structures to clarify the logic of your programs.

2. Practice information hiding and encapsulation in implementing data structures: Use functions to access your data structures, and keep these in classes separate from your application program.

3. Postpone decisions on the details of implementing your data structures as long as you can.

4. Stacks are among the simplest kind of data structures; use stacks when possible.

5. In any problem that requires a reversal of data, consider using a stack to store the data.

6. Avoid tricky ways of storing your data; tricks usually will not generalize to new situations.

7. Be sure to initialize your data structures.

8. In designing algorithms, always be careful about the extreme cases and handle them gracefully. Trace through your algorithm to determine what happens in extreme cases, particularly when a data structure is empty or full.

9. Before choosing implementations, be sure that all the data structures and their associated operations are fully specified on the abstract level.

Major References

For many topics concerning data structures, such as stacks, the best source for additional information, historical notes, and mathematical analysis is the following series of books, which can be regarded almost like an encyclopædia for the aspects of computing science that they discuss:

> Donald E. Knuth, *The Art of Computer Programming,* published by Addison-Wesley, Reading, Mass.

Three volumes have appeared to date:

1. *Fundamental Algorithms*, second edition, 1973, 634 pages.
2. *Seminumerical Algorithms*, second edition, 1980, 700 pages.
3. *Sorting and Searching*, 1973, 722 pages.

In future chapters we shall often give references to this series of books, and for convenience we shall do so by specifying only the name Knuth together with the volume and page numbers. The algorithms are written both in English and in an assembler language, where Knuth calculates detailed counts of operations to compare various algorithms.

A detailed description of the standard library in C++ occupies a large part of the following important reference, which gives the "official" description of C++ as written by its inventor:

> Bjarne Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, Mass., 1997.