A Portable C++ Library for Coroutine Sequencing

Keld Helsgaun
E-mail: keld@ruc.dk

Department of Computer Science
Roskilde University
DK-4000 Roskilde, Denmark

Abstract

This report describes a portable C++ library for coroutine sequencing. The facilities of the library are based on the coroutine primitives provided by the programming language SIMULA. The implementation of the library is described and examples of its use are given. One of the examples is a library for process-oriented discrete event simulation.

Keywords: coroutine, simulation, backtrack programming, C++, SIMULA, control extension.

1. Introduction

Coroutines can be used to describe the solutions of algorithmic problems that are otherwise hard to describe [1]. Coroutines provide the means to organize the execution of a program as several sequential processes.

A *coroutine* is an object that has its own stack of procedure activations. A coroutine may temporarily suspend its execution and another coroutine may be executed. A suspended coroutine may later be resumed at the point where it was suspended.

This form of sequencing is called *alternation.* Figure 1.1 shows a simple example of alternation between two coroutines.

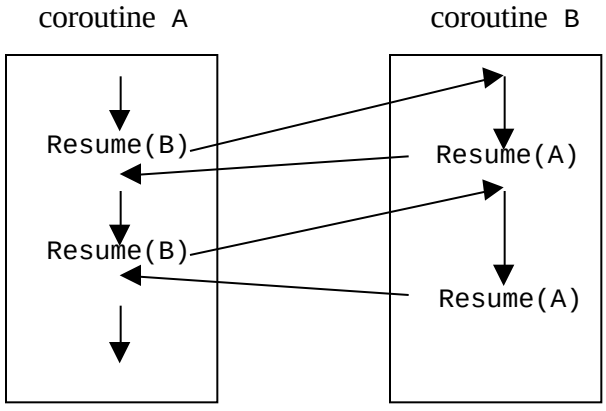

Figure 1.1 Alternation between two coroutines

Coroutines have demonstrated their usefulness through many years of use in SIMULA, for example. An illustrative example is a syntax analyzer calling a lexical analyzer for the next token. Here, the execution of the analyzers may be interleaved by implementing each of them as a coroutine. Another example is discrete event simulation. Here, coroutines can be used to model processes, i.e., objects that may undergo active and interactive phases during their lifetimes.

This report describes a library for coroutine sequencing in C++. The implementation does not use any platform-specific features and will run unmodified on most platforms. The facilities of the library are based on the coroutine primitives provided by the programming language SIMULA [2, 3]. Both symmetric and semi-symmetric sequencing is supported.

The rest of this paper is organized as follows. Section 2 gives an overview of the library. Some examples of its use are given in Section 3. Its implementation is described in Section 4.

Section 5 describes a library for discrete event simulation. The library is an implementation in C++ of SIMULA's built-in facilities for process-oriented discrete event simulation. An example of use of the library is given in Section 6.

The combination of coroutine sequencing and backtrack programming is explored in Section 7. Finally, some conclusions are made in Section 8.

2. The coroutine library

This section gives a brief description of the coroutine library from the user's point of view.

A coroutine program is composed of a collection of coroutines, which run in quasi-parallel with one another. Each coroutine is an object with its own execution-state, so that it may be suspended and resumed. A coroutine object provides the execution context for a function, called `Routine`, which describes the actions of the coroutine.

The library provides class `Coroutine` for writing coroutine programs. `Coroutine` is an abstract class. Objects can be created as instances of `Coroutine`-derived classes that implement the pure virtual function `Routine`. As a consequence of creation, the current execution location of the coroutine is initialized at the start point of `Routine`.

The interface of the coroutine library (`coroutine.h`) is sketched out below.

```
#ifndef Sequencing
#define Sequencing(S) { ...; S; }

class Coroutine {
protected:
    virtual void Routine() = 0;
};

void Resume(Coroutine *C);
void Call(Coroutine *C);
void Detach();

Coroutine *CurrentCoroutine();
Coroutine *MainCoroutine();

#endif
```

Control can be transferred to a coroutine `C` by one of two operations:

```
Resume(C)
Call(C)
```

Coroutine `C` resumes its execution from its current execution location, which normally coincides with the point where it last left off. The current coroutine is suspended within the `Resume` or `Call` operation, which is only completed at the subsequent resumption.

The `Call` operation establishes the current coroutine as `C`'s caller. A subordinate relationship exists between caller and called coroutines. `C` is said to be *attached* to its caller.

The current coroutine can relinquish control to its caller by means of the operation

```
Detach()
```

The caller resumes its execution from the point where it last left off. The current coroutine is suspended within the `Detach` operation, which is only completed at the subsequent resumption.

The function `CurrentCoroutine` may be used to get a pointer to the currently executing coroutine.

A coroutine corresponding to the main program exists at the beginning of program execution. A pointer to this coroutine is provided through the function `MainCoroutine`.

Below is shown a complete coroutine program. The program shows the use of the `Resume` function for coroutine alternation as illustrated in Figure 1.1. The macro `Sequencing` is used in the last line to make the main program behave as a coroutine.

```
#include "coroutine.h"
#include <iostream.h>

Coroutine *A, *B;

class CoA : public Coroutine {
    void Routine() {
        cout << "A1 ";
        Resume(B);
        cout << "A2 ";
        Resume(B);
        cout << "A3 ";
    }
};

class CoB : public Coroutine {
    void Routine() {
        cout << "B1 ";
        Resume(A);
        cout << "B2 ";
        Resume(A);
        cout << "B3 ";
    }
};

void MainProgram() {
    A = new CoA;
    B = new CoB;
    Resume(A);
    cout << "STOP ";
}

int main() Sequencing(MainProgram())
```

Execution of this program produces the following (correct) output:

```
A1 B1 A2 B2 A3 STOP
```

A coroutine may be in one of four states of execution at any time: *attached*, *detached*, *resumed* or *terminated*. Figure 2.1 shows the possible state transitions of a coroutine.

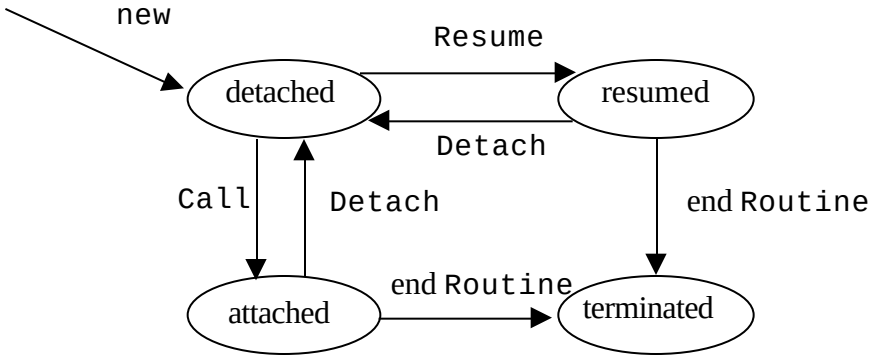

Figure 2.1 Execution states of a coroutine

A coroutine program consists of *components*. Each component is a chain of coroutines. The head of the component is a detached or resumed coroutine. The other coroutines are attached to the head, either directly or through other coroutines.

The main program corresponds to a detached coroutine, and as such it is the head of a component. This component is called the *main component*.

Exactly one component is operative at any time. Any non-operative component has an associated *reactivation point*, which identifies the program point where execution will continue if and when the component is activated (by `Resume` or `Call`).

When calling `Detach` there are two cases:

1) The coroutine is attached. In this case, the coroutine is detached, its execution is suspended, and execution continues at the reactivation point of the component to which the coroutine was attached.

2) The coroutine is resumed. In this case, its execution is suspended, and execution continues at the reactivation point of the main component.

Termination of a coroutine's `Routine` function has the same effect as a `Detach` call, except that the coroutine is terminated, not detached. As a consequence, it attains no reactivation point and it loses its status as a component head.

A call `Resume(C)` causes the execution of the current operative component to be suspended and execution to be continued at the reactivation point of `C`. The call constitutes an error in the following cases:

`C` is `NULL`
`C` is attached
`C` is terminated

A call `Call(C)` causes the execution of the current operative component to be suspended and execution to be continued at the reactivation point of `C`. In addition, `C` becomes attached to the calling component. The call constitutes an error in the following cases:

`C` is `NULL`
`C` is attached
`C` is resumed
`C` is terminated

A coroutine program using only `Resume` and `Detach` is said to use *symmetric* coroutine sequencing. If only `Call` and `Detach` are used, the program is said to use *semi-symmetric* coroutine sequencing. In the latter case, the coroutines are called *semi-coroutines.*

3. Examples

3. 1 A simple dice game

The following program simulates four people playing a simple dice game. The players, represented as coroutines, take turns at throwing a die. The first player to accumulate 100 pips wins and prints his identification.

The `Player`-objects are kept in a circular list. When a `Player`-object becomes active, it throws the die by selecting a random integer between 1 and 6. If the `Player`-object has not won, it resumes the next `Player`-object in the circle. Otherwise, it terminates, causing the main program to be resumed.

```
#include "coroutine.h"
#include <stdlib.h>
#include <iostream.h>

class Player : public Coroutine {
public:
    int Id;
    Player *Next;
    Player(int id) : Id(id) {}

    void Routine() {
        int Sum = 0;
        while ((Sum += rand()%6+1) < 100)
            Resume(Next);
        cout << "The winner is player " << Id << endl;
    }
};

void DiceGame(int Players) {
    Player *FirstPlayer = new Player(1),
           *p = FirstPlayer;
    for (int i = 2; i <= Players; i++, p = p->Next)
        p->Next = new Player(i);
    p->Next = FirstPlayer;
    Resume(FirstPlayer);
}

int main() Sequencing(DiceGame(4))
```

The program above uses symmetric coroutines. Alternatively, semi-coroutines could have been used. In the program below, the main program acts as master, while the players act as slaves. The main program uses the primitive `Call` to make the players throw the die in turn. After throwing the die, a player transfers control to the main program by calling `Detach`.

```
#include "coroutine.h"
#include <stdlib.h>
#include <iostream.h>

class Player : public Coroutine {
public:
    int Sum, Id;
    Player *Next;
    Player(int id) : Id(id) { Sum = 0; }

    void Routine() {
        for (;;) {
            Sum += rand()%6+1;
            Detach();
        }
    }
};

void DiceGame(int Players) {
    Player *FirstPlayer = new Player(1),
           *p = FirstPlayer;
    for (int i = 2; i <= Players; i++, p = p->Next)
        p->Next = new Player(i);
    p->Next = FirstPlayer;
    for (p = FirstPlayer; p->Sum < 100; p = p->Next)
        Call(p);
    cout << "The winner is player " << p->Id << endl;
}

int main() Sequencing(DiceGame(4))
```

3.2 Generation of permutations

Class `Permuter` shown below can be used to generate all permutations of integers between 1 and n, where n is a parameter to the constructor of the class. An object of the class acts as a semi-coroutine. Each time the object is `Call`ed, it makes the next permutation available in the integer array `p`. The elements `p[1]`, `p[2]`, ..., `p[n]` contain the permutation. When all permutations have been generated, the integer member `More` (initially 1) is set to zero. The permutations are generated using the recursive function `Permute`.

```
class Permuter : public Coroutine {
public:
    int N, *p, More;
    Permuter(int n) : N(n) { p = new int[N+1]; }

    void Permute(int k) {
        if (k == 1)
            Detach();
        else {
            Permute(k-1);
            for (int i = 1; i < k; i++) {
                int q = p[i]; p[i] = p[k]; p[k] = q;
                Permute(k-1);
                q = p[i]; p[i] = p[k]; p[k] = q;
            }
        }
    }

    void Routine() {
        for (int i = 1; i <= N; i++)
            p[i] = i;
        More = 1;
        Permute(N);
        More = 0;
    }
};
```

The following program uses class `Permuter` to print all permutations of the integers between 1 and 5.

```
void PrintPermutations(int n) {
    Permuter *P = new Permuter(n);
    Call(P);
    while (P->More) {
        for (int i = 1; i <= n; i++)
            cout << P->p[i] << " ";
        cout << endl;
        Call(P);
    }
}

int main() Sequencing(PrintPermutations(5))
```

3.3 Text transformation

Consider the following problem [4, 5]. A text is to be read from cards and printed on a line printer. Each card contains 80 characters, but the line printer prints 125 characters on each line. We want to pack as many characters as possible on each output line, marking the transition from one card to the next merely by the insertion of an extra space. In the text a consecutive pair of asterisks is to be replaced by a '^'. The end of the text is marked by the special character '•'.

The program given solves this problem by means of the following five coroutines:

`Reader`	fills, on each resumption the array `Card` with 80 characters from the next card.
`Disassembler`	takes the characters from the array `Card` and delivers them one by one to the squasher (through the global character variable `c1`).
`Squasher`	performs the transformation on pairs of asterisks, and outputs the characters one by one to the assembler (through the global character variable `c2`).
`Assembler`	groups the characters delivered by the squasher into lines, and delivers each line in the array `Line` to the printer.
`Printer`	will, on each resumption, print the characters from the array `Line` on the next line on paper.

In the present implementation the operations of `Reader` and `Printer` are simulated using the standard I/O streams in C++, `cin` and `cout`.

```
#include "coroutine.h"
#include <iostream.h>

const int CardLength = 80, LineLength = 125;
char Card[CardLength], Line[LineLength], c1, c2;
Coroutine *theReader, *theDisassembler, *theSquasher,
          *theAssembler, *thePrinter;

class Reader : public Coroutine {
    void Routine() {
        for (;;) {
            for (int i = 0; i < CardLength; i++)
                cin >> Card[i];
            Resume(theDisassembler);
        }
    }
};

class Disassembler : public Coroutine {
    void Routine() {
         for (;;) {
            Resume(theReader);
            for (int i = 0; i < CardLength; i++) {
                c1 = Card[i]; Resume(theSquasher);
            }
            c1 = ' ';
            Resume(theSquasher);
        }
    }
};

class Squasher : public Coroutine {
     void Routine() {
        for (;;) {
            if (c1 == '*') {
                Resume(theDisassembler);
                if (c1 == '*') {
                    c2 = '^'; Resume(theAssembler);
                }
                else {
                    c2 = '*'; Resume(theAssembler);
                    c2 = c1;
                }
            }
            else
                c2 = c1;
            Resume(theAssembler);
            Resume(theDisassembler);
        }
    }
};
```

```
class Assembler : public Coroutine {
     void Routine() {
        for (;;) {
            for (int i = 0; i < LineLength; i++) {
                Line[i] = c2;
                if (c2 == '•') {
                    while (++i < LineLength)
                        Line[i] = ' ';
                    Resume(thePrinter);
                    Detach();  // back to main program
                }
                Resume(theSquasher);
            }
            Resume(thePrinter);
        }
    }
};

class Printer : public Coroutine {
    void Routine() {
        for (;;) {
            for (int i = 0; i < LineLength; i++)
                cout << Line[i];
            cout << endl;
            Resume(theAssembler);
        }
    }
};

void TextTransformation() {
    theReader = new Reader();
    theDisassembler = new Disassembler();
    theSquasher = new Squasher();
    theAssembler = new Assembler();
    thePrinter = new Printer();
    Resume(theDisassembler);
}

int main() Sequencing(TextTransformation())
```

3.4 Two simple generators

A semi-coroutine may be employed when a generator is needed. The purpose of a generator is to produce a sequence of outputs. Each time the generator is called, it produces the next output.

3.4.1 A random number generator

Class `Random` shown below can be used to generate a sequence of pseudo-random integers in the interval from 0 to 32767. The generator produces the same sequence of numbers as the standard C library function `rand`. The next random number is obtained as a result of calling the public member function `Next`. The generator seed may be given as a parameter to the constructor.

```
class Rand : public Coroutine {
public:
    Rand(int Seed = 1) : U(Seed) {}
    unsigned int Next() {
        Call(this);
        return (unsigned int) (U/65536) % 32768;
    }
private:
    void Routine() {
        for (;;) {
            U = U*1103515245 + 12345;
            Detach();
        }
    }
    unsigned long U;
};
```

A small test program that prints a sequence of 20 pseudo-random numbers is shown below.

```
void RandTest() {
     Rand *R = new Rand;
     for (int i = 1; i <= 20; i++)
        cout << R->Next() << endl;
}

int main() Sequencing(RandTest())
```

3.4.2 A Fibonacci number generator

The Fibonacci numbers are integers defined by the following recurrence relation

$$F_n = F_{n-1} + F_{n-2}, \qquad \text{for } n \geq 2 \text{ with } F_0 = F_1 = 1.$$

This defines the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Class `Fibonacci` shown below can be used as a generator for the Fibonacci numbers.

```
class Fibonacci : public Coroutine {
public:
    int Next() {
        Call(this);
        return F2;
    }
private:
    void Routine() {
        F1 = 0; F2 = 1;
        for (;;) {
            Detach();
            F2 += F1; F1 = F2 - F1;
        }
    }
    int F1, F2;
};
```

A small test program that prints the first 20 Fibonacci numbers is shown below.

```
void FibonacciTest() {
    Fibonacci *F = new Fibonacci;
    for (int i = 1; i <= 20; i++)
        cout << F->Next() << endl;
}

int main() Sequencing(FibonacciTest())
```

3.5 Merging two sorted arrays

The following program merges two sorted arrays, `A` and `B`, into a single sorted array, `C`. A coroutine is associated with each of the two arrays to be merged. At any moment the coroutine proceeds, which inspects the smallest data item.

```
#include "coroutine.h"
#include <iostream.h>

int A[] = {1,5,6,8,10,12,15,17};
int B[] = {2,4,7,9,11,13,14,18,20,30};
int *C;
int m = 8, n = 10, CIndex;

class Traverser : public Coroutine {
public:
    Traverser(int A[], int L) :
             Array(A), Limit(L), Index(0) {}
    int *Array, Limit, Index;
    Traverser *Partner;

    void Routine() {
        while (Index < Limit) {
            if (Partner->Array[Partner->Index] <
                Array[Index])
                Resume(Partner);
            C[CIndex++] = Array[Index++];
        }
        while (CIndex < m+n)
            C[CIndex++] =
                Partner->Array[Partner->Index++];
    }
};

void MergeArrays() {
    Traverser *X = new Traverser(A, m);
    Traverser *Y = new Traverser(B, n);
    X->Partner = Y; Y->Partner = X;
    C = new int[m+n];
    CIndex = 0;
    Resume(X);
    for (int j = 0; j < m+n; j++)
        cout << C[j] << " ";
    cout << endl;
}

int main() Sequencing(MergeArrays())
```

3.6 Merging binary search trees

Class `Traverser` shown below is intended to be used for scanning the values in a binary search tree of integers in ascending order.

This operation is implemented by a semi-coroutine, which on each call assigns its integer member `Current` the next higher value of the node of the tree. When the whole tree has been traversed, `Current` is assigned the maximum integer value (`INT_MAX`).

The scanning is accomplished by a local recursive function, which calls `Detach` each time a node is visited.

```
class Tree {
public:
    Tree(int V, Tree *L, Tree *R) :
         Value(V), Left(L), Right(R) {}
    int Value;
    Tree *Left, *Right;
};

class Traverser : public Coroutine {
public:
    Traverser(Tree *T) : MyTree(T) {}
    int Current;
private:
    Tree *MyTree;

    void Routine() {
        Traverse(MyTree);
        Current = INT_MAX;
    }

    void Traverse(Tree *T) {
        if (T != NULL) {
            Traverse(T->Left);
            Current = T->Value;
            Detach();
            Traverse(T->Right);
        }
    }
};
```

An example of use is given in the test program below, which merges the values of two binary search trees and output the values in ascending order.

```
void MergeTrees() {
     Tree *Tree1 =
         new Tree(8,
             new Tree(5,
                 new Tree(1, NULL, NULL),
                 new Tree(6, NULL, NULL)),
         new Tree(10,
             NULL,
             new Tree(12,
                 NULL,
                 new Tree(15,
                     NULL,
                    new Tree(17, NULL, NULL)))));
    Tree *Tree2 =
        new Tree(13,
            new Tree(4,
                new Tree(2, NULL, NULL),
                new Tree(9,
                    new Tree(7, NULL, NULL),
                    new Tree(11, NULL, NULL))),
            new Tree(20,
                new Tree(14,
                    NULL,
                    new Tree(18, NULL, NULL)),
                new Tree(30, NULL, NULL)));
    Traverser *T1 = new Traverser(Tree1);
    Traverser *T2 = new Traverser(Tree2);
    Call(T1);
    Call(T2);
    for (;;) {
        int min = T1->Current;
        if (T2->Current < min) {
            min = T2->Current;
            Call(T2);
        } else {
            if (min == INT_MAX) break;
            Call(T1);
        }
        cout << min << " ";
    }
    cout << endl;
}

int main() Sequencing(MergeTrees())
```

3.7 Binary insertion sort

The coroutine `Tree` shown below may be used to sort integer values using the binary insertion method. Sorting is accomplished by building a binary search tree. Each node of the tree is a semi-coroutine [6].

If more than one integer is to be sorted, `Tree` sorts them by creating two more `Tree`-objects, `Left` and `Right`, and having each of them sort some of the integers.

`Left` sorts integers less than or equal to the value in the node, whereas `Right` sorts integers larger than the value in the node.

The end of the set of integers to be sorted is signaled with the value -1. When a coroutine receives a value of -1, it stops sorting and prepares to return the sorted integers one at a time.

```
int V;

class Tree : public Coroutine {
     int Value;
     Tree *Left, *Right;

    void Routine() {
        if (V == -1) {
            Detach();
            V = -1;
            return;
        }
        Value = V;
        Tree *Left = new Tree, *Right = new Tree;
        for (;;) {
            Detach();
            if (V == -1)
                break;
            if (V <= Value)
                Left->Send(V);
            else
                Right->Send(V);
        }
        Left->Send(-1); Right->Send(-1);
        Detach();
```

```
        for (;;) {
            V = Left->Receive();
            if (V == -1)
                break;
            Detach();
        }
        V = Value;
        Detach();
        for (;;) {
            V = Right->Receive();
            if (V == -1)
                break;
            Detach();
        }
        V = -1;
     }

public:
    void Send(int InputValue) {
        V = InputValue;
        Call(this);
    }

    int Receive() {
        Call(this);
        return V;
    }
};
```

A simple program that uses class `Tree` to sort 100 random integers between 0 and 100 is shown below.

```
void BinaryInsertionSort() {
    Tree *BST = new Tree;
    for (int i = 0; i < 100; i++)
        BST->Send(rand()%100);
    BST->Send(-1);
    int i;
    while ((i = BST->Receive()) != -1)
        cout << i << " ";
    cout << endl;
}

int main() Sequencing(BinaryInsertionSort())
```

3. 8 A cash dispenser

Coroutines may often be used to solve problems that are solvable by means of backtracking. Each incarnation of a recursive solution of the problem is replaced by a coroutine.

Consider the following problem [7]. A cash dispenser is able to make payments to a customer. Write a program that, given a wanted amount of money, computes the number of coins and notes to be paid out.

Such a program is shown below. Each kind of coin (or note) is represented as an instance of class `Coin`. The integer members `Denomination` and `Number` denote the denomination of this kind of coin, and the number of available coins of this denomination, respectively.

For each kind of coin the program computes the number of that coin to be used in a payment. The integer member `Used` contains this number.

Each `Coin`-object is a coroutine that is able to try all possible payments from its stock of coins. The objects are held in a two-way list in descending order of coin denominations. Each time a coroutine becomes active, it tries the next possible payment and resumes its successor (`Suc`). However, if it has no successor, or all possible payments have been tried, it resumes its predecessor (`Pred`). If it has no predecessor, i.e., it is the first object in the list, all possible combinations have been tried, and the given payment problem cannot be solved.

The two-way list is implemented by using the facilities of the library `simset` (described in Appendix D).

```
#include "coroutine.h"
#include "simset.h"
#include <iostream.h>
int Amount;
inline int min(int a, int b) { return a < b ? a : b; }

class Coin : public Link, public Coroutine {
public:
    Coin(int d, int n) : Denomination(d), Number(n) {}
    int Denomination, Number, Used;

    void PrintSolution() {
        if (Pred() != NULL)
            ((Coin*) Pred())->PrintSolution();
        if (Used > 0)
            cout << Used << " of " << Denomination
                 << endl;
    }

    void Routine() {
        for (;;) {
            for (Used =
                   min(Amount/Denomination, Number);
                 Used >= 0;
                 Used--) {
                Number -= Used;
                Amount -= Used*Denomination;
                if (Amount == 0) {
                    PrintSolution();
                    Detach();
                }
                if (Suc() != NULL)
                    Resume((Coin*) Suc());
                Amount += Used*Denomination;
                Number += Used;
            }
            if (Pred() == NULL) {
                cout << "No solution" << endl;
                Detach();
            }
            Resume((Coin*) Pred());
        }
    }
};
```

```
void ChangeDispensor() {
    cout << "Amount to be paid: "; cin >> Amount;
    Head *List = new Head;
    (new Coin(1000,19))->Into(List);
    (new Coin(500,9))->Into(List);
    (new Coin(100,11))->Into(List);
    (new Coin(50,10))->Into(List);
    (new Coin(20,32))->Into(List);
    (new Coin(10,0))->Into(List);
    (new Coin(5,1))->Into(List);
    (new Coin(1,7))->Into(List);
    Resume((Coin*) List->First());
}

int main() Sequencing(ChangeDispensor())
```

3. 9 A filter for telegrams

This problem has been taken from [7].

A telegram is a text without any punctuation characters. The word STOP is used instead of a period. A telegram ends with the word ZZZZ.

Assume we have a file consisting of a set of telegrams. The end of the file is signaled by two consecutive ZZZZ-words. Write a program that prints the contents of the file such that

1) the word STOP is replaced by a period
2) redundant spaces are removed
3) a maximum of 20 characters are printed on each line without hyphenation of any word
4) the individual telegrams are separated by two blank lines.

The program below solves this problem. The program contains the following three coroutines

`LetterProducer`	reads letters from the file and delivers them one by one to the word producer. Each new line character is replaced by a space.
`WordProducer`	assembles the letters into words. Extraneous spaces are removed, and the word STOP is replaced by a period. The words are handed over one-by-one to the printer.
`Printer`	receives words from the word producer and prints them on lines with a maximum of 20 characters.

```
#include "coroutine.h"
#include <iostream.h>
#include <fstream.h>
#include <string.h>

char Word[20], Letter;
ifstream *TelegramFile;
Coroutine *theLetterProducer,
          *theWordProducer,
          *thePrinter;

class LetterProducer : public Coroutine {
    void Routine() {
        for (;;) {
            TelegramFile->get(Letter);
            if (Letter == '\n')
                Letter = ' ';
            Resume(theWordProducer);
        }
    }
};

class WordProducer : public Coroutine {
    void Routine() {
        for (;;) {
            while (Letter == ' ')
                Resume(theLetterProducer);
            char NextWord[21];
            NextWord[0] = '\0';
            do {
                char NextLetter[1];
                NextLetter[0] = Letter;
                strncat(NextWord, NextLetter, 1);
                Resume(theLetterProducer);
            } while (Letter != ' ');
            if (!strcmp(NextWord, "STOP"))
                strcat(Word, ".");
            else {
                if (strlen(Word))
                    Resume(thePrinter);
                if (!strcmp(Word, "ZZZZ") &&
                    !strcmp(NextWord, "ZZZZ"))
                    Detach();
                strcpy(Word, NextWord);
            }
        }
    }
};
```

```
class Printer : public Coroutine {
    void Routine() {
        int LineLength = 0;
        for (;;) {
            while (strcmp(Word, "ZZZZ")) {
                if (LineLength + strlen(Word) > 20) {
                    cout << endl;
                    LineLength = 0;
                }
                cout << Word << " ";
                LineLength += strlen(Word) + 1;
                Resume(theWordProducer);
            }
            cout << endl << endl;
            LineLength = 0;
            Resume(theWordProducer);
        }
    }
};

void TelegramFilter() {
    do {
        cout << "Enter file name: ";
        char FileName[80];
        cin >> FileName;
        delete TelegramFile;
        TelegramFile = new ifstream(FileName,ios::in);
    } while (!TelegramFile->is_open());

    theLetterProducer = new LetterProducer;
    theWordProducer = new WordProducer;
    thePrinter = new Printer;
    Resume(theLetterProducer);
}

int main() Sequencing(TelegramFilter())
```

4. Implementation

A coroutine is characterized mainly by its execution state consisting of its current execution location and a stack of activation records. The bottom element of the stack is the `Routine` activation record. The remaining part of the stack contains activation records corresponding to function activations triggered by `Routine`.

When control is transferred to a coroutine (by means of `Resume`, `Call` or `Detach`), the coroutine must be able to carry on where it left off. Thus, its execution state must persist between successive occasions on which control enters it. Its execution state must be "frozen", so to speak.

When a coroutine transfers from one execution state to another, it is called a *context switch.* This implies the saving of the execution state of the suspending coroutine and its replacement with the execution state of the other coroutine.

The central issue when implementing coroutines is how to achieve such context switches. The goal is to implement the primitive `Enter(C)` with the following semantics [1]:

`Enter(C)`	The execution point for the currently executing coroutine is set to the next statement to be executed, after which this coroutine becomes suspended and the coroutine `C` (re-)commences execution at its execution point

Having implemented this primitive, it is easy to implement the primitives `Resume`, `Call` and `Detach` (or similar primitives).

In the following two implementations of `Enter` are presented. The first implementation is based on copying of stacks in and out of C++'s runtime stack. In the second implementation all stacks reside in the runtime stack (i.e., no stacks are copied).

Both implementations exploit the services of the C++ library functions `setjmp` and `longjmp`.

Each implementation has both advantages and drawbacks when compared with the other one. For this reason, both implementations are made available in two separate versions of the coroutine library.

No platform-specific features are used. Thus, both implementations will run without modifications on most platforms.

4.1 The copy-stack implementation

This implementation works in principle as follows.

> At any time, the stack of the currently operating coroutine is held in C++'s runtime stack.
>
> When a coroutine suspends, the runtime stack and the current execution location are copied to two buffers (`StackBuffer` and `Environment`) associated with the coroutine.
>
> A coroutine is resumed by copying the contents of its stack buffer to C++'s runtime stack and setting the program counter to the saved execution location.

The standard C++ functions `setjmp` and `longjmp` are used to implement the context switch.

> `setjmp` is used to save the current execution state in the buffer `Environment`. `Environment` contains a snapshot of the processor state (register values, including the program counter).
>
> `longjmp` is used to make the processor return to a saved state.

`setjmp` saves the current state in the buffer and returns 0. `longjmp` returns the processor to a previous state, as though `setjmp` had returned a value other than 0.

The following code shows the interface of class `Coroutine`.

```
class Coroutine {
friend void Resume(Coroutine *);
friend void Call(Coroutine *);
friend void Detach();
protected:
    Coroutine();
    ~Coroutine();
    virtual void Routine() = 0;
private:
    void Enter();
    void StoreStack();
    void RestoreStack();
    char *StackBuffer, *High, *Low;
    size_t BufferSize;
    jmp_buf Environment;
    Coroutine *Caller, *Callee;
};
```

The data members of the class have the following meaning.

`StackBuffer`	pointer to a buffer containing a copy of the runtime stack.
`High`, `Low`	address bounds of the runtime stack.
`BufferSize`	size (in bytes) of the stack buffer area.
`Environment`	array containing the information saved by `setjmp`.
`Caller, Callee`	attachment links.

The meaning of `StackBuffer`, `Low`, `High` and `BufferSize` is illustrated in Figure 4.1.

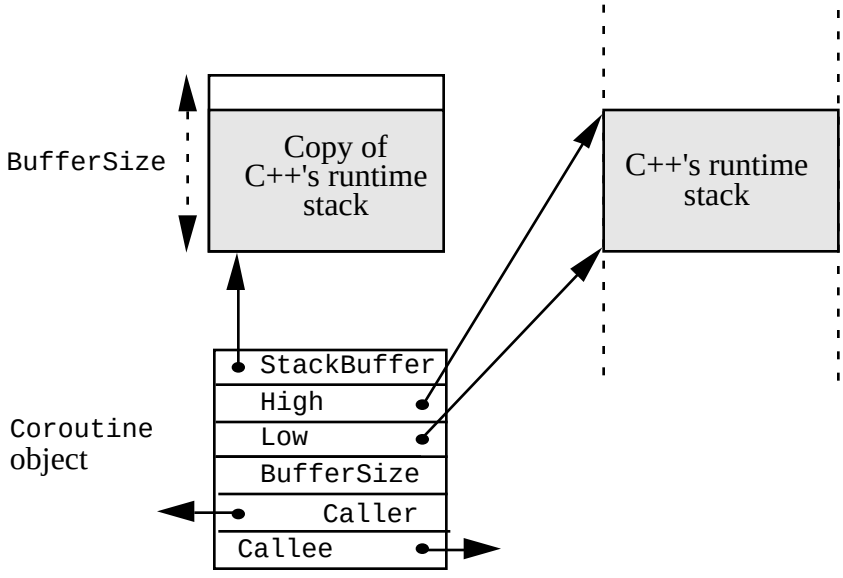

Figure 4.1 Data structures (copy-stack implementation)

A context switch is made by the member function `Enter`. A call `C->Enter()`, where `C` points to a `Coroutine` object, suspends the currently operating coroutine, `Current`, and resumes coroutine `C`. The runtime stack is saved in `Current`'s stack buffer, and `C`'s stack buffer is copied to the runtime stack.

The code of the function `Enter` is shown below.

```
void Coroutine::Enter() {
    if (!Terminated(Current)) {
        Current->StoreStack();
        if (setjmp(Current->Environment))
            return;
    }
    Current = this;
    if (StackBuffer) {
        Routine();
        delete Current->StackBuffer;
        Current->StackBuffer = 0;
        Detach();
    }
    RestoreStack();
}
```

The auxiliary function `StoreStack` is used to save the run time stack. Its implementation is shown below.

```
void Coroutine::StoreStack() {
     if (!Low) {
        if (!StackBottom)
            Error("StackBottom is not initialized");
        Low = High = StackBottom;
    }
    char X;
    if (&X > StackBottom)
        High = &X;
    else
        Low = &X;
    if (High - Low > BufferSize) {
        delete StackBuffer;
        BufferSize = High - Low;
        if (!(StackBuffer = new char[BufferSize]))
            Error("No more space available");
    }
    memcpy(StackBuffer, Low, High - Low);
}
```

First, the function computes the boundaries of the runtime stack, `Low` and `High`. It is assumed that the bottom of the runtime stack has already been initialized (by the macro `Sequencing`). Next, if necessary, it allocates a buffer, `StackBuffer`, to hold a copy of the run time stack. Finally, the run time stack is copied to this buffer.

Note that the function takes account of the fact that the runtime stack may grow up on some platforms and down on others.

In order to restore the state of a coroutine, the auxiliary function `RestoreStack` is used. The code of this function is shown below.

```
void Coroutine::RestoreStack() {
    char X;
    if (&X >= Low && &X <= High)
        RestoreStack();
    Current = this;
    memcpy(Low, StackBuffer, High - Low);
    longjmp(Current->Environment, 1);
}
```

The function copies the contents of the stack buffer to the runtime stack, and jumps to the execution location saved in the buffer `Environment`. First, however, the function calls itself recursively as long as the current top address is within the saved address bounds of the runtime stack. This prevents the restored stack from being destroyed by the subsequent call of `longjmp`.

Having implemented the function `Enter`, it is easy to implement the user functions `Resume`, `Call` and `Detach`. Their implementation, excluding error handling, is shown below.

```
Resume(Coroutine *Next) {
    while (Next->Callee)
        Next = Next->Callee;
    Next->Enter();
}

void Call(Coroutine *Next) {
    Current->Callee = Next;
    Next->Caller void = Current;
    while (Next->Callee)
        Next = Next->Callee;
    Next->Enter();
}

void Detach() {
    Coroutine *Next = Current->Caller;
    if (Next)
        Current->Caller = Next->Callee = 0;
    else {
        Next = &Main;
        while (Next->Callee)
            Next = Next->Callee;
    }
    Next->Enter();
}
```

The complete program code of this version of the coroutine library may be found in Appendix B.

The implementation is based on the same principles as was used by the author in his implementation of a library for backtrack programming in C [8]. Actually, the latter library may easily be implemented by means of the coroutine library (see Section 7). The same principles were used in [9] to extend C++ with control extensions similar to those described in this report.

4.2 The share-stack implementation

This implementation is more complex than the previous one. The basic idea, first time described by Kofoed [10], is to let all coroutine stacks share C++'s runtime stack.

The runtime stack is divided into contiguous areas of varying size. An area is either unused or contains a coroutine stack. Recursive function calls are used to wind down the stack and mark off allocated areas.

Each area contains a control block, called a *task*, which describes the properties of the area, for example its size and whether or not it is in use.

Class `Coroutine` has the following interface.

```
class Coroutine {
friend void Resume(Coroutine *);
friend void Call(Coroutine *);
friend void Detach();
friend void InitSequencing(size_t main_stack_size
                           = DEFAULT_STACK_SIZE);
protected:
    Coroutine(size_t stack_size = DEFAULT_STACK_SIZE);
    virtual void Routine() = 0;
private:
    void Enter();
    void Eat();
    Task *MyTask;
    size_t StackSize;
    int Ready, Terminated;
    Coroutine *Caller, *Callee;
};
```

The data members of the class have the following meanings.

`MyTask`	pointer to the control block.
`StackSize`	maximum area size (measured in bytes) for the stack.
`Ready`	signifies whether the coroutine is ready to run its `Routine`.
`Terminated`	signifies whether the coroutine has terminated its `Routine`.
`Caller`, `Callee`	attachment links.

The structure `Task` is shown below.

```
struct Task {
    Coroutine *MyCoroutine;
    jmp_buf jmpb;
    int used;
    size_t size;
    struct Task *pred, *suc;
    struct Task *prev, *next;
};
```

The members have the following meanings.

`MyCoroutine`	pointer to the owner coroutine
`jmpb`	the environment saved by `setjmp`
`used`	signifies whether the task is in use
`size`	the size (measured in bytes) of the associated area
`pred`, `suc`	predecessor and successor in a doubly linked list of unused tasks
`prev`, `next`	pointers to the two adjacent tasks

Figure 4.2 illustrates the data structures.

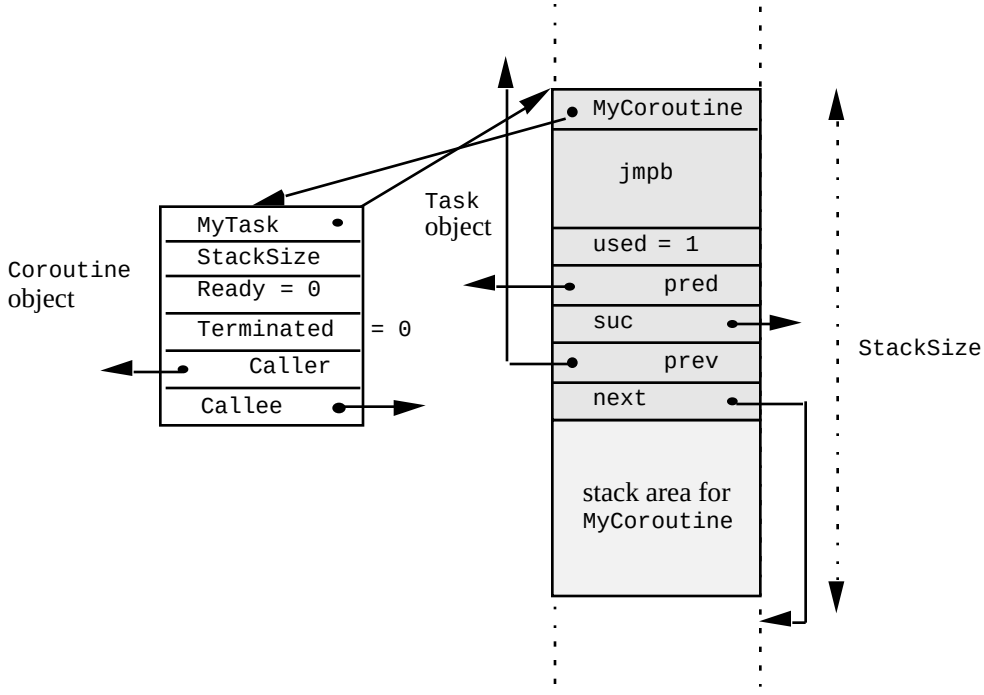

Figure 4.2 Data structures (share-stack implementation)

A task is a control block allocated in C++'s runtime stack.

A program is initialized by calling the function `InitSequencing`. This function saves the current state using `setjmp`, and calls a function, `Eat`, recursively until it has used enough of C++'s runtime stack to accommodate the stack corresponding to the main program. After that, the function saves the current state in a local control block together with the size of the remaining free area, marks the area as "free", and jumps to the previously saved state using `longjmp`. This control block serves as a potential starting point for allocating space for new coroutine stacks.

All control blocks are linked together with pointers (`pred` and `suc`) in a doubly linked list. When a new stack is to be allocated, the linked list is searched, using a first-fit algorithm, for a free area that is large enough. If the requested stack size is smaller than the area found, and the are is large enough to contain another stack of a predefined minimum size, the area is split using the previously mentioned recursive function `Eat`, creating a new free area. The original block is marked as "used" and is ready to be used for executing the actions of the coroutine in question.

When a stack is no longer needed (because the corresponding coroutine terminates), the control block is marked "free" and possibly merged with the preceding or following free block (referenced by `prev` and `next`, respectively).

The implementation closely follows the task library implementation made by Kofoed (see [10] for details). However, some adjustments have been made.

First, the coroutine library provides user-defined alternation between coroutines, whereas the task library provides alternation as determined by a pre-defined round-robin algorithm.

Second, in order to speed up the search for a possible merge of free blocks, the singly linked list of adjacent blocks has been replaced by a doubly linked list.

Third, at program initialization the user does not need to provide the size of the total stack area (only the size of the stack area required for the main program).

A complete listing of the share-stack implementation of the coroutine library may be found in Appendix C.

4.3 Comparison of the two implementations

Which of the two implementations is to be preferred? This question has no simple answer. Each implementation has both advantages and disadvantages.

Below the two implementations are evaluated in relation to a series of criteria.

4.3.1 Ease of use

Both implementations appear to be very easy to use.

An advantage of the copy-stack implementation is that the user does not need to bother about stack sizes for coroutines.

In contrast, the share-stack implementation requires the maximum stack size of each coroutine to be specified. This drawback has been somewhat reduced in the actual implementation of the copy-stack method. A default size (10,000 bytes) is used if the user omits the specification.

4.3.2 Efficiency

For some applications the copy-stack implementation may cause extensive copying.

In general, the time used for context switching is reduced when the copy-stack version of the library is replaced by the share-stack version. This gain in speed may be of importance in applications with many context switches and/or large stacks.

In Section 6 the implementations are compared with respect to their efficiency in a simulation application.

4.3.3 Restrictions in use

The copy-stack implementation gives rise to backtracking. At resumption of a coroutine, the stack is reestablished to its contents at the last time of suspension of the coroutine. This means that automatic variables (i.e., variables on the stack) have their values restored. Any changes between suspension and resumption are annulled. Furthermore, when an automatic object has been moved due to a control switch, pointers to that object are no longer valid.

Thus, shared variables should not be automatic in the copy-stack implementation. They should be global or static.

This restriction does not apply to the share-stack implementation.

However, the restriction is usually not important. Actually, as will be shown in Section 7 the backtracking property of the copy-stack implementation may be exploited for writing applications that combine the use of coroutine sequencing with backtracking.

4.3.4 Robustness

The share-stack implementation has no check on stack overflows. If the user has specified a stack size that is too small, the program will crash, or even worse, produce meaningless results without any notification.

4.3.5 Memory use

In the share-stack implementation the maximum stack size specified is allocated when a coroutine starts. If this upper bound is much too large, a lot of unnecessary memory is used. This is especially of importance in simulation applications where, usually, there are many simultaneous coroutines operating in quasi-parallel.

4.3.6 Maintenance

The copy-stack implementation seems to be the simplest to understand, and therefore also the simplest to maintain.

4.3.7 Portability

Both implementations are portable. At the time of writing, they have both been installed and tested successfully with compilers on the following machines: Macintosh, IBM PC and Sun SPARC.

5. The simulation library

Discrete event simulation is an important application area for coroutine sequencing.

Simulation is a technique for representing a dynamic system with a model and experimenting with the model in order to gain information about the system. The system could be a production line, a traffic system, a computer system, a post office, etc.

The following notions are common to such systems:

- *Processes* act in parallel, giving rise to *discrete events*
- *Queuing* arises when processes have to wait

Processes may have active as well as inactive phases. A process may be suspended temporarily and resumed later from where it left off. Thus, a process has the properties of a coroutine.

The coroutine library described in this report can be used to implement a library for simulation. Such a library is described in this section. The design of the library follows very closely the design of the built-in package for discrete event simulation in SIMULA, class simulation [3].

A simulation program is composed of a collection of processes that undergo scheduled and unscheduled phases. When a process is scheduled, it has an event time associated with it. This is the time at which its next active phase is scheduled to occur. When the active phase of a process ends, it may be rescheduled, or descheduled (either because all its actions have been executed, or the time of its next active phase is not known). In either case, the scheduled process with the smallest event time is resumed.

The currently active process always has the smallest event time associated with it. This time, the *simulation time*, moves in jumps to the event time of the next scheduled process.

Scheduled processes are contained in an *event list*. The processes are ordered in accordance with increasing event times. The process at the front of the event list is always the one, which is active. Processes not in the event list are either terminated or passive.

At any point in simulation time, a process can be in one (and only one) of the following states:

active: the process is at the front of the event list. Its actions are being executed

suspended: the process is in the event list, but not at the front

passive: the process is not in the event list and has further actions to execute

terminated: the process is not in the event list and has no further actions to execute.

The interface of the simulation library (`simulation.h`) is sketched below.

```
#ifndef Simulation
#define Simulation Sequencing

#include "coroutine.h"
#include "simset.h"
#include “random.h”

class Process : public Link, public Coroutine {
public:
    virtual void Actions() = 0;
    Process();
    int Idle() const;
    int Terminated() const;
    double EvTime() const;
    Process *NextEv() const;
};

Process *Main();
Process *Current();
double Time();

void Hold(double T);
void Passivate();
void Wait(Head *Q);
void Cancel(Process *P);

enum Haste   {at = 1, delay = 2};
enum Ranking {before = 3, after = 4};
enum Prior   {prior = 5};
void Activate(Process *P);
void Activate(Process *P, Haste H, double T);
void Activate(Process *P, Haste H, double T,
              Prior Prio);
void Activate(Process *P1, Ranking Rank, Process *P2);
```

```
void Reactivate(Process *P);
void Reactivate(Process *P, Haste H, double T);
void Reactivate(Process *P, Haste H, double T,
                Prior Prio);
void Reactivate(Process *P1, Ranking Rank,
                Process *P2);

void Accum(double &A, double &B, double &C, double D);

#endif
```

Processes can be created as instances of `Process`-derived classes that implement the pure virtual function `Actions`. The `Actions` function is used to describe the life cycle of a process.

Since class `Process` is a subclass of class `Link`, each process has the capability of being a member of a two-way list (see Appendix D). This is useful, for example, when processes must wait in a queue.

It is desirable to have the main program participating in the simulation as a process. This is achieved by an impersonating `Process`-object that can be manipulated like any other `Process`-object.

`Main()` returns a reference to this object.

`Current()` returns a reference to the `Process`-object at the front of the event list (the currently active process).

`Time()` returns the current simulation time.

`Hold(double T)` schedules `Current` for reactivation at `Time() + T`.

`Passivate()` removes `Current` from the event list and resumes the actions of the new `Current()`.

`Wait(Head *Q)` includes `Current` into the two-way list `Q`, and then calls `Passivate`.

`Cancel(Process *X)` removes the process `X` from the event list. If `X` is currently active or suspended, it becomes passive. If `X` is a passive or terminated process or `NULL`, the call has no effect.

There are seven ways to activate a currently *passive* process:

`Activate(Process *X)`: activates process `X` at the current simulation time.

`Activate(Process *X, before, Process *Y)`: positions process `X` in the event list *before* process `Y`, and gives it the same event time as `Y`.

`Activate(Process *Y, after, Process *X)`: positions process `Y` in the event list *after* process `X`, and gives it the same event time as `X`.

`Activate(Process *X, at, double T)`: the process `X` is inserted into the event list at the position corresponding to the event time specified by `T`. The process is inserted *after* any processes with the same event time which may already be present in the list.

`Activate(Process *X, at, double T, prior)`: the process `X` is inserted into the event list at the position corresponding to the event time specified by `T`. The process is inserted *before* any processes with the same event time which may already be present in the list.

`Activate(Process *X, delay, *T)`: the process `X` is activated after a specified delay, `T`. The process is inserted in the event list with the new event time, and *after* any processes with the same simulation time which may already be present in the list.

`Activate(Process *X, delay, *T, prior)`: the process `X` is activated after a specified delay, T. The process is inserted in the event list with the new event time, and *before* any processes with the same simulation time which may already be present in the list.

Correspondingly, there are seven `Reactivate` functions, which work on either *active*, *suspended* or *passive* processes. They have similar signatures to their `Activate` counterparts and work in the same way.

The following four public member functions are available in class `Process`:

> `Idle()` returns `1` if the process is not currently in the event list. Otherwise `0`.
>
> `Terminated()` returns `1` if the process has executed all its actions. Otherwise `0`.
>
> `EvTime()` returns the time at which the process is scheduled for activation.
>
> `NextEv()` returns a reference to the next process, if any, in the event list.

In addition, the simulation library provides the function `Accum` that can be used to compute the time integral of a variable.

> `Accum(double &A, double &B, double &C, double D)` accumulates the "time integral" of the variable `C`, interpreted as a step function of the simulated time. The integral is accumulated in the variable `A`. The variable `B` contains the event time at which the variables were last updated. The value of `D` is the current increment of the step function. The code is:
>
> ```
> A += C*(Time()-B);
> B = Time();
> C += D;
> ```

A listing of the complete source code of the simulation library can be found in Appendix E

In a simulation, it is often necessary to specify the distribution functions of various events (e.g., the time between car arrivals at a traffic light). For this purpose, the simulation library provides functions for random drawing. These functions, collected in a separate library described in Appendix F, have the same functionality as in SIMULA.

6. A simulation example

This example has been taken from [2]

A garage owner has installed an automatic car wash that services cars one at a time. Each service takes 10 minutes. When a car arrives, it goes straight into the car wash if this is idle, otherwise it must wait in a queue. As long as cars are waiting, the car wash is in continuous operation serving on a first-come, first-served basis. The average time between car arrivals has been estimated at 11 minutes.

The garage owner is interested in predicting the maximum queue length and average waiting time if he installs one more car wash.

A simulation program that solves this problem is presented below.

It is assumed that each car wash is manned by a car washer, and that the car washers start their day in a tearoom and return there each time they have no work to do. A car washer may described by the following subclass of `Process`:

```
class CarWasher : public Process {
    void Actions() {
        for (;;) {
            Out();
            while (!WaitingLine->Empty()) {
                Car *Served =
                        (Car*) WaitingLine->First();
                Served->Out();
                Hold(10);
                Activate(Served);
                delete Served;
            }
            Wait(TeaRoom);
        }
    }
};
```

The actions of a car washer are contained in an infinite loop (the length of the simulation is supposed to be determined by the main program). Each time a car washer is activated, he leaves the tearoom (by calling `Out`) and starts serving the cars in the waiting line. He takes the first car out of the waiting line, washes it for ten minutes (`Hold(10)`). The car washer will continue servicing, as long as there are cars waiting in the queue. If the waiting line becomes empty, he returns to the tearoom and waits.

A car may be described by the following subclass of `Process`:

```
class Car : public Process {
    void Actions() {
        double EntryTime = Time();
        Into(WaitingLine);
        int QLength = WaitingLine->Cardinal();
        if (MaxLength < QLength)
            MaxLength = QLength;
        if (!TeaRoom->Empty())
            Activate((Process*) TeaRoom->First());
        Passivate();
        NoOfCustomers++;
        ThroughTime += Time() - EntryTime;
    }
};
```

On arrival each car enters the waiting line and, if the tearoom is not empty, it activates the idle car washer in the room. The car washer then washes it. If, however, the tearoom is empty, the car waits until a car washer can service it.

When a car has been washed (and activated by the car washer), it leaves the garage.

The following subclass of `Process` is used to make cars arrive to the garage with an inter-arrival time of `P` minutes:

```
class CarGen : public Process {
    void Actions() {
        while (Time() < SimPeriod) {
            Activate(new Car);
            Hold(Negexp(1.0/P, U));
        }
    }
};
```

The main program, shown below, generates the two queues, `TeaRoom` and `WaitingLine`, and activates the two car washers and the car generator.

```
void CarWash() {
    P = 11.0; N = 2; SimPeriod = 200; U = 5;
    TeaRoom = new Head;
    WaitingLine = new Head;
    for (int i = 1; i <= N; i++)
        (new CarWasher)->Into(TeaRoom);
    Activate(new CarGen);
    Hold(SimPeriod+100000);
    Report();
}

int main() Simulation(CarWash())
```

`Simperiod` denotes the total opening time of the garage (200 minutes). All cars that have arrived before the garage closes down are washed. When all activity has stopped, function `Report`, shown below, prints the number of cars washed, the average elapsed time (wait time plus service time), and the maximum queue length.

```
void Report() {
    cout << N << " Car washer simulation\n";
    cout << "No.of cars through the system = "
         << NoOfCustomers << endl;
    cout << "Av.elapsed time = "
         << ThroughTime/NoOfCustomers << endl;
    cout << "Maximum queue length = " << MaxLength
         << endl;
}
```

A complete listing of the program can be found in Appendix G.

This program was used with `Simperiod` set to `1000000` to compare the efficiency of the two versions of the coroutine library.

The CPU time used to run the program on a 300 MHz PowerPC Macintosh was

copy-stack	1.9 seconds
share-stack	1.2 seconds
SIMULA	6.0 seconds

The last line shows the time needed to run a SIMULA version of the program (Lund SIMULA 4.07).

The CPU-time used to run the program on a SUN SPARC server was

copy-stack	5.5 seconds
share-stack	3.4 seconds
SIMULA	6.2 seconds

The last line shows the time needed to run a SIMULA version of the program (cim-2.8, a SIMULA compiler that produces C code).

As can be seen, the share-stack is more efficient than the copy-stack version.

Considering that both versions use a primitive implementation of the event list, it is interesting to note that they both result in a program that runs faster than an equivalent SIMULA program.

7. Combining coroutines and backtracking

Backtrack programming is a well-known technique for solving combinatorial search problems [11]. The search is organized as a multi-stage search process where, at each stage, a choice among a number of alternatives has to be made. Whenever it is found that the previous choices cannot possibly lead to a solution, the algorithm backtracks, that is to say, re-establishes its state exactly as is was at the most recent choice point and chooses the next untried alternative at this point. If all alternatives have been tried, the algorithm backtracks to the previous choice point.

Backtrack programming is often realized by recursion. A choice is made by calling a recursive procedure. A backtrack is made by returning from the procedure. When a return is made, the programmer must take care that the program's variables are restored to their values at the time of the call.

However, writing programs that explicitly handle their own backtracking can be difficult, tedious and error-prone. For this reason, a number of high-level languages have been supplemented with special facilities for backtrack programming.

A simple, but general tool for backtrack programming in C is described in [8]. The tool, called CBack, implements the two functions `Choice` and `Backtrack.`

`Choice` is used when a choice is to be made among a number of alternatives. `Choice(N)`, where `N` is a positive integer denoting the number of alternatives, returns successive integer. `Choice` first returns the value 1, and the program continues. The values 2 to N are returned by `Choice` through subsequent calls of `Backtrack.`

A call of `Backtrack` causes the program to *backtrack*, that is to say, return to the most recent call of `Choice`, which has not yet returned all its values. The state of the program is re-established exactly as it was when `Choice` was called, and the next untried value is returned. All *automatic* program variables, i.e. local variables and register variables, will be re-established. The remaining variables, the *static* variables, are not touched.

For a more comprehensive description of these facilities, see [8].

The implementation of CBack is based on the copy-stack method. At each `Choice`-call, a copy of the C's runtime stack is saved. When `Backtrack` is called, the program state is re-established from the saved copy. The copies are kept in a stack. A call of `Backtrack` re-establishes the state from the top element of the stack. When a `Choice`-call returns its last alternative, the corresponding copy is popped from the stack.

Given the copy-stack version of the coroutine library, it is a simple matter to implement CBack. Appendix H contains the source code of such an implementation. It should be noted that only the most essential parts of CBack have been included in this implementation.

The program below solves the classical 'eight-queens' problem using the CBack library. The task is to place eight queens on a chessboard so that no queen is under attack by another; that is, there is at most one queen on each row, column and diagonal.

```
#include "CBack.h"
#include <iostream.h>

int Q[9];

void EightQueensProblem() {
    for (int r = 1; r <= 8; r++) {
        int c = Choice(8);
        for (int i = 1; i < r; i++)
            if (c == Q[i] || abs(c - Q[i]) == r - i)
                Backtrack();
        Q[r] = c;
    }
    for (int r = 1; r <= 8; r++)
        cout << Q[r] << " ";
    cout << endl;
}

int main() Backtracking(EightQueensProblem())
```

It is interesting that the copy-stack technique allows for the combination of coroutine control and backtracking. Coroutine control over several backtracking subsystems can, for example, be used to achieve "planning" or cooperative solutions (e.g. working from both ends to solve a maze problem). Lindstrom [12] has shown that coroutines and backtracking can be combined in a coherent manner; in fact, Lindstrom has used this combination to implement a “non-forgetful” form of backtracking, in which it its possible to remember previously searched subgoals and to re-use the results of these searches.

Appendix I contains an implementation of CBack in which each coroutine controls its own backtracking subsystem. In addition to being a full implementation of CBack, it adds coroutines and the possibility of having simultaneous backtracking systems (one for each coroutine).

Depth-first is the default search strategy. If required, the user may obtain best-first strategy. In the present implementation the simple sorted list representation of the priority queue of states has been replaced by a pairing heap representation.

The conceptual and programmatic utility of the coroutine-backtracking control combination is now illustrated through two examples. Both examples have been taken from [12]. The description and the source code follow the paper closely.

7.1 Minimal node weight sum of two trees

Problem: Given two n-ary trees with non-negative integer node weights, determine which tree has the root to terminal path with the smallest node weight sum (if both trees possesses such a path, then either tree is an admissible answer).

This problem may be solved by pursuing minimum path searches on both trees at once, with control alternating between them on a "can you top this" basis.

A program following this strategy is shown below. The program consists of two independent search algorithms controlled as coroutines by the main program.

```
#include "CBack.h"
#include <iostream.h>
#include <limits.h>

const int NodeMax = 100, DegreeMax = 5;
unsigned int Deg[NodeMax];           // degree of each node
unsigned int Wt[NodeMax];            // weight of each node
unsigned int Desc[NodeMax][DegreeMax]; // immed. desc. of nodes
unsigned long Best = ULONG_MAX;      // current minimum

class Shortest : public Coroutine {
public:
    Shortest(unsigned int R) : Root(R) {}
private:
    unsigned int Root;

    void Routine() {
        unsigned long Sum;     // current path sum
        unsigned int NodeNow;  // current node pointer

        Sum = Wt[NodeNow = Root];
        while (Sum < Best) {
            if (Deg[NodeNow] > 0) {  // pick descendant node
                NodeNow = Desc[NodeNow][Choice(Deg[NodeNow])];
                Sum += Wt[NodeNow];
            }
            else {       // this node is terminal
                Best = Sum;
                Detach();      // let colleague try to beat it
                Backtrack();   // look for alternate path
            }
        }
        Backtrack();     // current path already too large
    }
};

void Treewalk() {
    // assume Desc, Deg, Wt, Root1 and Root2 already read in
    Shortest *T[] = {new Shortest(Root1), new Shortest(Root2)};
    unsigned long OldBest;
    int i = 0;           // let system 0 try first
    do {
        OldBest = Best;  // save current best path weight
        Call(T[i]);      // call current subsystem
        i = 1 - i; // switch to other subsystem
    } while (Best < OldBest);
    cout << "Tree " << i << " has minimum terminal path"
         << endl;
}

int main() Backtracking(Treewalk())
```

7.2 Context-free language intersection

Problem: Given two context-free grammars G_1 and G_2, find a minimal length string on $L(G_1) \cap L(G_2)$, i.e. the intersection of their languages. If no such string exists of length less than or equal to *lim*, report that fact and stop; otherwise, exhibit the string found and stop.

This problem may be solved by means of the following algorithm:

- Set k = 1.
- Generate all length k strings of $L(G_1)$ via a backtracking system. As each new character is produced (in left-to-right order) in a potential length k member of $L(G_1)$, pass that character to a G_2 parser, operating as a backtracking subsystem of the $L(G_1)$ generator.
- The G_2 parser attempts to accommodate the current string as extended by the new character. If the G_2 parser fails in this attempt, it signals failure to the G_1 generator, which then retracts that character. In either case, the G_1 generator continues its generation process as described in step 2.
- This process continues until either (a) an entire string of length k is produced and parsed as a member of $L(G_2)$, in which case a success is reported and the program terminates after printing the string; or (b) the G_1 generator has produced all length k members of $L(G_1)$.
- In the latter case k is incremented and compared against *lim*. If k > *lim*, a message is printed indicating the absence of the desired string and the program halts. Otherwise, the process begins anew at step 2.

A program that implements this algorithm is shown below. For details see [12].

```
#include "CBack.h"
#include <iostream.h>

const int strmax = 25;

enum type {alt, conc, term};   // rule forms
type ruletype[CHAR_MAX+1];     // rule types
char rule[CHAR_MAX+1][2];      // rules of grammars
char root1, root2;             // grammar root symbols
char str[strmax];              // string buffer
int lim;                       // length limit
int ok, strfound;              // success signals

class parseboss : public Coroutine {
friend class genboss;
public:
    parseboss(genboss *genref) : genref(genref) {}
    void Routine();
    void parse(char goal);
    int parseptr;              // parser's string pointer
    genboss *genref;           // reference to generator
};

class genboss : public Coroutine {
friend class parseboss;
public:
    genboss(int k) : k(k) { parseref = new parseboss(this); }
    ~genboss() { delete parseref; }
private:
    void Routine();
    void generate(char goal, int substrlength);
    int k;                     // string length
    int genptr;                // generator's string pointer
    parseboss *parseref;       // reference to parser
};

void parseboss::Routine() {
    Notify(parseptr);          // backtrace parseptr
    parseptr = 0;              // initialize string pointer
    parse(root2);              // start up parser
    if (parseptr < genref->genptr+1) // parsed only prefix of string
        Backtrack();           // so backup
    ok = 1;                    // signal success on last character
    strfound = 1;              // and overall success
    Detach();                  // await generator's pleasure
    strfound = 0;              // another character has arrived
    Backtrack();                     // so backup
}
```

```
void parseboss::parse(char goal) {
    // find string spanned by goal, starting at parseptr
    switch(ruletype[goal]) {
    case alt:
        parse(rule[goal][Choice(2)-1]);
        break;
    case conc:
        parse(rule[goal][0]);
        parse(rule[goal][1]);
        break;
    case term:
       if (parseptr > genref->genptr) { // need another character
            ok = 1;      // so far, so good
            Detach();    // if controls returns, we have it
        }
        if (str[parseptr] != rule[goal][0])
            Backtrack();
        parseptr++;
    }
}

void genboss::Routine() {
    Notify(genptr);      // backtrace genptr
    genptr = 0;          // initialize string pointer
    generate(root1,k);   // start up generation
    if (!strfound)       // generator finished, but not parser
        Backtrack();     // so back up
}

void genboss::generate(char goal, int k) {
    switch(ruletype[goal]) {
    case alt:
        generate(rule[goal][Choice(2)-1],k);
        break;
    case conc:
        int j = Choice(k-1);   // form all 2-partions of k
        generate(rule[goal][0],j);
        generate(rule[goal][1],k-j);
        break;
    case term:
        if (k > 1)
            Backtrack();
        str[genptr] = rule[goal][0];
        ok = 0;                // clear signal
        Call(parseref);        // give char. to parser
        if (!ok)               // parser could not oblige
            Backtrack();       // so back up
        genptr++;              // otherwise, onward
    }
}
```

```
void GrammarIntersection() {
    // assume rule, ruletype, root1, root2 and lim already read in
     BreadthFirst = 1;    // use breadth-first search
     strfound = 0;
     int k = 0;
     while (k < lim && !strfound) {
         // see if a common string of length k+1 exists
         k++;
         genboss *g = new genboss(k);
         Call(g);
         delete g;
         ClearAll();
     }
     if (strfound) {
         cout << "Success, string = ";
         for (int i = 0; i < k; i++)
             cout << str[i];
         cout << endl;
     }
     else
        cout << "No common string of length less than or equal to "
              << lim << endl;
}

int main() Backtracking(GrammarIntersection())
```

8. Conclusions

This report describes a portable C++ library for coroutine sequencing. The library has been implemented in two versions, the copy-stack version and the share-stack version. Both versions show quite good performance. Their adequacy has been demonstrated through the implementation of a library for process-oriented discrete event simulation.

It is not clear-cut which version is to be preferred. Each version has drawbacks that are not present in the other version. More user experience is required in other to judge which version should be recommended in general.

References

1. C. D. Marlin,
   Coroutines,
   Lecture Notes in Computer Science (1980).

2. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug & K. Nygaard,
   SIMULA BEGIN,
   Studentlitteratur (1974).

3. *Programspråk – SIMULA*,
   SIS, Svensk Standard SS 63 61 14 (1987).

4. M. E. Conway,
   Design of a Separable Transition-Diagram Compiler,
   Comm, A.C.M., **6**(7), pp. 396-408 (1963).

5. O.-J. Dahl & C. A. R. Hoare,
   Hierarchical Program Structures,
   Structured Programming, pp. 175-220,
   eds. O.-J. Dahl, E. W. Dijkstra & C. A. R. Hoare,
   Academic Press (1972).

6. P. A. Buhr, G. Ditchfield, A. Stroobosscher, B. M. Younger & C. R. Zarnke,
   μC++: Concurrency in the Object-oriented Language C++,
   Softw. prac. exp., **22**(2), pp. 137-172 (1992)

7. H. B. Hansen,
   SIMULA - et objektorienteret programmeringssprog,
   Kompendium, Roskilde Universitetscenter (1990).

8. K. Helsgaun,
   CBack: A Simple Tool for Backtrack Programming in C,
   Softw. prac. exp., **25**(8), pp. 905-934 (1995).

9. L. Nigro,
   Control extensions in C++,
   Object Oriented Programming, **6**(9), pp. 37-47 (1994)

10. S. Kofoed,
Portable Multitasking in C++,
Dr. Dobb's Journal, November (1995)

11. R. W. Floyd,
Nondeterministic algorithms.
Journal ACM ,**14**(4), 636-644 (1967).

12. G. Lindstrom,
Backtracking in a Generalized Control Setting,
A.C.M. Trans. on Programming Languages and Systems, **1**(1),
pp. 8-26 (1979).

Appendices

A. Installation of the coroutine library

B. Source code of the copy-stack version of the coroutine library

C. Source code of the share-stack version of the coroutine library

D. The simset library

E. Source code of the simulation library

F. The random drawing library

G. Source code of the car wash simulation program

H. A rudimentary implementation of CBack

I. A complete implementation of CBack (with coroutine sequencing)

A. Installation of the coroutine library

The source code of the coroutine library is provided in two files, a header file called `coroutine.h`, and a source file called `coroutine.cpp`. Two versions of the library are provided, the copy-stack version (described in Section 4.1) and the share-stack version (described in Section 4.2).

Both versions should run without modifications on most platforms. However, for the copy-stack version a small adjustment may be necessary. Some C++ systems do not always keep the runtime stack up-to-date but keep some of the variable values in registers. This is for example the case for C++ systems on Sun machines. If this is the case, the macro `Synchronize` must be used. The comment characters are simply removed from the macro definition (in the beginning of `coroutine.cpp` of the copy-stack version).

The library may now be compiled and tested with the program shown on the next page [7].

The program should produce the following output

```
m1a1m2b1m3a2c1a3b2c2
==>
```

and wait for input from the keyboard.

If the character `r` is typed, the program should print

```
b3a4m4c3m5
```

and stop. If the character `c` is typed, the program should print

```
b3a4c3m4c4m5
```

and stop. If any other character is typed, the program should print

```
m4c3m5
```

and stop.

```
#include "coroutine.h"
#include <iostream.h>
#include <stdlib.h>

class A : public Coroutine { void Routine(); };
class B : public Coroutine { void Routine(); };
class C : public Coroutine { void Routine(); };
Coroutine *a1, *b1, *c1;

void A::Routine() {
    cout << "a1"; Detach();
    cout << "a2"; Call(c1 = new C);
    cout << "a3"; Call(b1);
    cout << "a4"; Detach();
}

void B::Routine() {
    cout << "b1"; Detach();
    cout << "b2"; Resume(c1);
    cout << "b3";
};

void C::Routine() {
    cout << "c1"; Detach();
    cout << "c2" << endl << "==> "; flush(cout);
    char command;
    cin >> command;
    if (command == 'r')
        Resume(a1);
    else if (command == 'c')
        Call(a1);
    else
        Detach();
    cout << "c3"; Detach();
    cout << "c4";
}

void TestProgram() {
    cout << "m1"; Call(a1 = new A);
    cout << "m2"; Call(b1 = new B);
    cout << "m3"; Resume(a1);
    cout << "m4"; Resume(c1);
    cout << "m5" << endl;
}

int main() Sequencing(TestProgram())
```

B. Source code of the copy-stack version of the coroutine library

Header file: `coroutine.h`

```
#ifndef Sequencing
#define Sequencing(S) {char Dummy; StackBottom = &Dummy; S;}
#include <stddef.h>
#include <setjmp.h>

extern char *StackBottom;

class Coroutine {
friend void Resume(Coroutine *);
friend void Call(Coroutine *);
friend void Detach();
friend class Process;
friend unsigned long Choice(long);
friend void Backtrack();
protected:
    Coroutine(size_t Dummy = 0);
    ~Coroutine();
    virtual void Routine() = 0;
private:
    void Enter();
    void StoreStack();
    void RestoreStack();
    char *StackBuffer, *Low, *High;
    size_t BufferSize;
    jmp_buf Environment;
    Coroutine *Caller, *Callee;
    static Coroutine *ToBeResumed;
};

void Resume(Coroutine *);
void Call(Coroutine *);
void Detach();

Coroutine *CurrentCoroutine();
Coroutine *MainCoroutine();

#define DEFAULT_STACK_SIZE 0

#endif
```

Source file: coroutine.cpp

```
#define Synchronize // {jmp_buf E; if (!setjmp(E))
longjmp(E,1);}
#include "coroutine.h"
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

char *StackBottom;

#define Terminated(C) (!(C)->StackBuffer && (C)->BufferSize)
static Coroutine *Current = 0, *Next;

static void Error(const char *Message) {
    cerr << "Error: " << Message << endl;
    exit(0);
}

Coroutine *Coroutine::ToBeResumed = 0;

static class MainCoroutine : public Coroutine {
public:
    MainCoroutine() { Current = this; }
    void Routine() {}
} Main;

Coroutine::Coroutine(size_t Dummy = 0) {
    char X;
    if (StackBottom)
        if (&X < StackBottom ?
            &X <= (char*) this && (char*) this <= StackBottom :
            &X >= (char*) this && (char*) this >= StackBottom)
          Error("Attempt to allocate a Coroutine on the stack");
    StackBuffer = 0; Low = High = 0; BufferSize = Dummy = 0;
    Callee = Caller = 0;
}

Coroutine::~Coroutine() {
    delete StackBuffer; StackBuffer = 0;
}

inline void Coroutine::RestoreStack() {
    Synchronize;
    char X;
    if (&X >= Low && &X <= High) RestoreStack();
    Current = this;
    memcpy(Low, StackBuffer, High - Low);
    longjmp(Current->Environment, 1);
}
```

```
inline void Coroutine::StoreStack() {
    if (!Low) {
        if (!StackBottom)
            Error("StackBottom is not initialized");
        Low = High = StackBottom;
    }
    char X;
    if (&X > StackBottom)
        High = &X;
    else
        Low = &X;
    if (High - Low > BufferSize) {
        delete StackBuffer;
        BufferSize = High - Low;
        if (!(StackBuffer = new char[BufferSize]))
            Error("No more space available");
    }
    Synchronize;
    memcpy(StackBuffer, Low, High - Low);
}

inline void Coroutine::Enter() {
    if (!Terminated(Current)) {
        Current->StoreStack();
        if (setjmp(Current->Environment))
            return;
    }
    Current = this;
    if (!StackBuffer) {
        Routine();
        delete Current->StackBuffer;
        Current->StackBuffer = 0;
        if (ToBeResumed) {
            Next = ToBeResumed;
            ToBeResumed = 0;
            Resume(Next);
        }
        Detach();
    }
    RestoreStack();
}
```

```
void Resume(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Resume a non-existing Coroutine");
    if (Next == Current)
        return;
    if (Terminated(Next))
        Error("Attempt to Resume a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Resume an attached Coroutine");
    while (Next->Callee)
        Next = Next->Callee;
    Next->Enter();
}

void Call(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Call a non-existing Coroutine");
    if (Terminated(Next))
        Error("Attempt to Call a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Call an attached Coroutine");
    Current->Callee = Next;
    Next->Caller = Current;
    while (Next->Callee)
        Next = Next->Callee;
    if (Next == Current)
        Error("Attempt to Call an operating Coroutine");
    Next->Enter();
}

void Detach() {
    Next = Current->Caller;
    if (Next)
        Current->Caller = Next->Callee = 0;
    else {
        Next = &Main;
        while (Next->Callee)
            Next = Next->Callee;
    }
    Next->Enter();
}

Coroutine *CurrentCoroutine() { return Current; }

Coroutine *MainCoroutine() { return &Main; }
```

C. Source code of the share-stack version of the coroutine library

Header file: `coroutine.h`

```
#ifndef Sequencing
#include <stddef.h>
#include <setjmp.h>
#define DEFAULT_STACK_SIZE 10000
#define Sequencing(S) { InitSequencing(DEFAULT_STACK_SIZE); S; }
class Task;

class Coroutine {
friend void Resume(Coroutine *);
friend void Call(Coroutine *);
friend void Detach();
friend class Process;
friend void InitSequencing(size_t main_StackSize);
protected:
    Coroutine(size_t StackSize = DEFAULT_STACK_SIZE);
    virtual void Routine() = 0;
private:
    void Enter();
    void Eat();
    Task *MyTask;
    size_t StackSize;
    int Ready, Terminated;
    Coroutine *Caller, *Callee;
    static Coroutine *ToBeResumed;
};

void Resume(Coroutine *);
void Call(Coroutine *);
void Detach();

Coroutine *CurrentCoroutine();
Coroutine *MainCoroutine();

void InitSequencing(size_t main_StackSize);
```

```
struct Task {
    Coroutine *MyCoroutine;
    jmp_buf jmpb;
    int used;
    size_t size;
    struct Task *pred, *suc;
    struct Task *prev, *next;
};

#endif
```

Source file: coroutine.cpp

```
#include "coroutine.h"
#include <iostream.h>
#include <stdlib.h>
#include <limits.h>
#define MIN_STACK_SIZE 500          // minimum stack size

static Task main_task;              // the main task
static jmp_buf tmp_jmpb;            // temporary jump buffer
static Coroutine *Current = 0;      // current coroutine

static void Error(const char *Msg) {
    cerr << Msg << endl; exit(0);
}

Coroutine *Coroutine::ToBeResumed = 0;

class Main_Coroutine : public Coroutine {
friend class Process;
public:
    void Routine() {}
} Main;

Coroutine::Coroutine(size_t s = DEFAULT_STACK_SIZE) {
    Caller = Callee = 0; Ready = 1; Terminated = 0;
    StackSize = s;
}
```

```
void Coroutine::Enter() {
    if (!Current)
        Error("InitSequencing has not been called");
    if (Ready) {                          // find free block
        for (MyTask = main_task.suc;
             MyTask != &main_task;
             MyTask = MyTask->suc)
            if (MyTask->size >= StackSize + MIN_STACK_SIZE)
                break;
            if (MyTask == &main_task)
                Error("No more space available\n");
            MyTask->MyCoroutine = this;
            if (!setjmp(tmp_jmpb))
                longjmp(MyTask->jmpb, 1);
            Ready = 0;
    }
    if (!setjmp(Current->MyTask->jmpb)) { // activate control block
        Current = this;
        longjmp(MyTask->jmpb, 1);
    }
}

void Coroutine::Eat() {
    static size_t d;
    static Task *p;
    Task t;

    // eat stack
    if ((d = labs((char *) &t - (char *) MyTask)) <
        StackSize)
        Eat();
    t.size = MyTask->size - d;      //set size
    MyTask->size = d;
    t.used = 0;
    t.suc = main_task.suc;
    t.pred = &main_task;
    t.suc->pred = main_task.suc = &t;
    if (MyTask->next != &t) {
        t.next = MyTask->next;      // set link pointers
        MyTask->next = &t;
        t.prev = MyTask;
        if (t.next)
            t.next->prev = &t;
    }
    if (!setjmp(t.jmpb))            // wait
        longjmp(MyTask->jmpb, 1);
```

```
    for (;;) {
        // test size
        if (StackSize + MIN_STACK_SIZE < t.size &&
            !setjmp(t.jmpb))
            t.MyCoroutine->Eat();   // split block
        t.used = 1;                 // mark as used
        t.pred->suc = t.suc;
        t.suc->pred = t.pred;
        if (!setjmp(t.jmpb))        // wait
            longjmp(tmp_jmpb, 1);
        t.MyCoroutine->Routine();   // execute Routine
        t.MyCoroutine->Terminated = 1;
        t.used = 0;                 // mark as free
        p = t.next;
        if (p && !p->used) {  // merge with following block
            t.size += p->size;
            t.next = p->next;
             if (t.next)
                 t.next->prev = &t;
             p->pred->suc = p->suc;
             p->suc->pred = p->pred;
        }
        p = t.prev;
        if (!p->used) {       // merge with preceding block
            p->size += t.size;
            p->next = t.next;
            if (t.next)
                t.next->prev = p;
        }
        else {
            t.suc = main_task.suc;
            t.pred = &main_task;
            t.suc->pred = main_task.suc = &t;
        }
        if (!setjmp(t.jmpb)) {      // save state
            if (ToBeResumed) {
                static Coroutine *Next;
                Next = ToBeResumed;
                ToBeResumed = 0;
                Resume(Next);
            }
            else
                Detach();
        }
    }
}
```

```
void Resume(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Resume a non-existing Coroutine");
    if (Next == Current)
        return;
    if (Next->Terminated)
        Error("Attempt to Resume a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Resume an attached Coroutine");
    while (Next->Callee)
        Next = Next->Callee;
    Next->Enter();
}

void Call(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Call a nonexisting Coroutine");
    if (Next->Terminated)
        Error("Attempt to Call a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Call an attached Coroutine");
    Current->Callee = Next;
    Next->Caller = Current;
    while (Next->Callee)
        Next = Next->Callee;
    if (Next == Current)
        Error("Attempt to Call an operating Coroutine");
    Next->Enter();
}

void Detach() {
    Coroutine *Next = Current->Caller;
    if (Next)
        Current->Caller = Next->Callee = 0;
    else {
        Next = &Main;
        while (Next->Callee)
            Next = Next->Callee;
    }
    Next->Enter();
}

Coroutine *CurrentCoroutine() { return Current; }

Coroutine *MainCoroutine() { return &Main; }
```

```
void InitSequencing(size_t main_StackSize) {
    Task tmp;
    tmp.size = ULONG_MAX;
    Main.StackSize = main_StackSize;
    tmp.next = 0;
    Main.MyTask = &tmp;
    main_task.pred = main_task.suc = &main_task;
    tmp.MyCoroutine = Current = &Main;
    if (!setjmp(tmp.jmpb))
        Main.Eat();
    tmp.pred = main_task.pred;
    tmp.suc = main_task.suc;
    main_task = tmp;
    main_task.next->prev = &main_task;
    Main.MyTask = &main_task;
    main_task.used = 1;
    Main.Ready = 0;
}
```

D. The simset library

This library contains facilities for the manipulation of two-way linked lists. Its functionality corresponds closely to SIMULA's built-in class `simset`.

List members are objects of subclasses of the class `Link`.

An object of the class `Head` is used to represent a list.

The class `Linkage` is a common base class for list heads and list members.

The three classes are described below by means of the following variables:

```
Head *HD;
Link *LK;
Linkage *LG;
```

Class `Linkage`

```
class Linkage {
public:
    Link *Pred();
    Link *Suc();
    Linkage *Prev();
};
```

`LK.Suc()` returns a reference to the list member that is the successor of `LK` if `LK` is a list member and is not the last member of the list; otherwise 0.

`HD.Suc()` returns a reference to the fist member of the list HD, if the list is not empty; otherwise 0.

`LK.Pred()` returns a reference to the list element that is the predecessor of `LK` if `LK` is a list member and is not the first member of the list; otherwise 0.

`HD.Pred()` returns a reference to the last member of the list `HD`, if the list is not empty; otherwise 0.

`LK.Prev()` returns 0 if `LK` is not a list member, a reference to the list head, if `LK` is the first member of a list; otherwise a reference to `LK`'s predecessor in the list.

`HD.Prev()` returns a reference to `HD` if `HD` is empty; otherwise a reference to the last member of the list.

Class `Head`

```
class Head : public Linkage {
public:
    Link *First();
    Link *Last();
    int Empty(void) const;
    int Cardinal(void) const;
    void Clear(void);
};
```

`HD.First()` returns a reference to the first member of the list (0, if the list is empty).

`HD.Last()` returns a reference to the last member of the list (0, if the list is empty).

`HD.Cardinal()` returns the number of members in the list (0, if the list is empty).

`HD.Empty()` returns 1 if the list `HD` has no members; otherwise 0.

`HD.Clear()` removes all members from the list.

Class `Link`

```
class Link : public Linkage {
public:
    void Out(void);
    void Into(Head *H);
    void Precede(Linkage *L);
    void Follow(Linkage *L);
};
```

`LK.Out()` removes `LK` from the list (if any) of which it is a member. The call has no effect if `LK` has no membership.

`LK.Into(HD)` removes `LK` from the list (if any) of which it is a member and inserts `LK` as the last member of the list `HD`.

`LK.Precede(LG)` removes `LK` from the list (if any) of which it is a member and inserts `LK` before `LG`. The effect is the same as `LK.Out()` if `LG` is 0, or it has no membership and is not a list head.

`LK.Follow(LG)` removes `LK` from the list (if any) of which it is a member and inserts `LK` after `LG`. The effect is the same as `LK.Out()` if `LG` is 0, or it has no membership and is not a list head.

Header file: `simset.h`

```
#ifndef SIMSET_H
#define SIMSET_H

class Linkage {
friend class Link;
friend class Head;
public:
    Linkage();
    Link *Pred() const;
    Link *Suc() const;
    Linkage *Prev() const;
private:
    Linkage *PRED, *SUC;
    virtual Link *LINK() = 0;
};

class Head : public Linkage {
public:
    Head();
    Link *First() const;
    Link *Last() const;
    int Empty(void) const;
    int Cardinal(void) const;
    void Clear(void);
private:
    Link *LINK();
};

class Link : public Linkage {
public:
    void Out(void);
    void Into(Head *H);
    void Precede(Linkage *L);
    void Follow(Linkage *L);
private:
    Link *LINK();
};

#endif
```

Source file: simset.cpp

```
#include "simset.h"

Linkage::Linkage() { SUC = PRED = 0; }

Link* Linkage::Pred() const { return PRED ? PRED->LINK() : 0; }

Link* Linkage::Suc() const { return SUC ? SUC->LINK() : 0; }

Linkage* Linkage::Prev() const { return PRED; }

Head::Head() { SUC = PRED = this; }

Link* Head::First() const { return Suc(); }

Link* Head::Last() const { return Pred(); }

int Head::Empty() const { return SUC == (*Linkage) this; }

int Head::Cardinal() const {
    int i = 0;
    for (Link *L = First(); L; L = L->Suc())
        i++;
    return i;
}

void Head::Clear() { while (First()) First()->Out(); }

Link* Head::LINK() { return 0; }

void Link::Out() {
    if (SUC) { SUC->PRED = PRED; PRED->SUC = SUC; PRED = SUC = 0; }
}

void Link::Into(Head *H) { Precede(H); }

void Link::Precede(Linkage *L) {
    Out();
    if (L && L->SUC)
      { SUC = L; PRED = L->PRED; L->PRED = PRED->SUC = this; }
}

void Link::Follow(Linkage *L) { if (L) Precede(L->SUC); else Out();
}

Link* Link::LINK() { return this; }
```

E. Source code of the simulation library

Header file: `simulation.h`

```
#ifndef Simulation
#define Simulation Sequencing

#include "coroutine.h"
#include "simset.h"
#include "random.h"

class Process : public Link, public Coroutine {
friend Process *Current();
friend double Time();
friend void Activat(int Reac, Process *X, int Code, double T,
                    Process *Y, int Prio);
friend void Hold(double T);
friend void Passivate();
friend void Wait(Head *Q);
friend void Cancel(Process *P);
friend class Main_Program;
friend class SQS_Process;
public:public:
    virtual void Actions() = 0;
    Process(size_t stack_size = DEFAULT_STACK_SIZE);
    int Idle() const;
    int Terminated() const;
    double EvTime() const;
    Process *NextEv() const;
private:
    void Routine();
    int TERMINATED;
    Process *PRED, *SUC;
    double EVTIME;
};

Process *Main();
Process *Current();
double Time();

void Hold(double T);
void Passivate();
void Wait(Head *Q);
void Cancel(Process *P);

enum Haste {at = 1, delay = 2};
enum Ranking {before = 3, after = 4};
enum Prior {prior = 5};
```

```
void Activate(Process *P);
void Activate(Process *P, Haste H, double T);
void Activate(Process *P, Haste H, double T, Prior Prio);
void Activate(Process *P1, Ranking Rank, Process *P2);

void Reactivate(Process *P);
void Reactivate(Process *P, Haste H, double T);
void Reactivate(Process *P, Haste H, double T, Prior Prio);
void Reactivate(Process *P1, Ranking Rank, Process *P2);

void Accum(double &A, double &B, double &C, double D);

#endif
```

Source file: simulation.cpp

```
#include "simulation.h"
#include <iostream.h>

static void Error(const char *Msg) {
    cerr << "Error: " << Msg << endl;
    exit(0);
}

class SQS_Process : public Process {
public:
    SQS_Process() { EVTIME = -1; PRED = SUC = this; }
    void Actions() {}
    inline static void SCHEDULE(Process *Q, Process *P) {
        Q->PRED = P; Q->SUC = P->SUC;
        P->SUC = Q; Q->SUC->PRED = Q;
    }
    inline static void UNSCHEDULE(Process *P) {
        P->PRED->SUC = P->SUC;
        P->SUC->PRED = P->PRED;
        P->PRED = P->SUC = 0;
    }
} SQS;

class Main_Program : public Process {
public:
    Main_Program() { EVTIME = 0; SQS.SCHEDULE(this,&SQS); }
    void Actions() { while (1) Detach(); }
} MainProgram;

Process::Process(size_t stack_size = DEFAULT_STACK_SIZE) :
                Coroutine(stack_size)
    { TERMINATED = 0; PRED = SUC = 0; }

void Process::Routine() {
    Actions();
    TERMINATED = 1;
    SQS.UNSCHEDULE(this);
    if (SQS.SUC == &SQS)
        Error("SQS is empty");
    ToBeResumed = SQS.SUC;
}
int Process::Idle() const { return SUC == 0; }
int Process::Terminated() const { return TERMINATED; }
double Process::EvTime() const {
    if (SUC == 0)
        Error("No EvTime for Idle Process");
    return EVTIME;
}
```

```
Process* Process::NextEv() const
    { return SUC == &SQS ? 0 : SUC; }
Process *Main() { return &MainProgram; }
Process *Current() { return SQS.SUC; }
double Time() { return SQS.SUC->EVTIME; }

void Hold(double T) {
    Process *Q = SQS.SUC;
    if (T > 0)
        Q->EVTIME += T;
    T = Q->EVTIME;
    if (Q->SUC != &SQS && Q->SUC->EVTIME <= T) {
        SQS.UNSCHEDULE(Q);
        Process *P = SQS.PRED;
        while (P->EVTIME > T)
            P = P->PRED;
        SQS.SCHEDULE(Q,P);
        Resume(SQS.SUC);
    }
}

void Passivate() {
    Process *CURRENT = SQS.SUC;
    SQS.UNSCHEDULE(CURRENT);
    if (SQS.SUC == &SQS)
        Error("SQS is empty");
    Resume(SQS.SUC);
}

void Wait(Head *Q) {
    Process *CURRENT = SQS.SUC;
    CURRENT->Into(Q);
    SQS.UNSCHEDULE(CURRENT);
    if (SQS.SUC == &SQS)
        Error("SQS is empty");
    Resume(SQS.SUC);
}

void Cancel(Process *P) {
    if (!P || !P->SUC)
        return;
    Process *CURRENT = SQS.SUC;
    SQS.UNSCHEDULE(P);
    if (SQS.SUC != CURRENT)
        return;
    if (SQS.SUC == &SQS)
        Error("SQS is empty");
    Resume(SQS.SUC);
}
```

```
enum {direct = 0};

void Activat(int Reac, Process *X, int Code,
             double T, Process *Y, int Prio) {
    if (!X || X->TERMINATED || (!Reac && X->SUC))
        return;
    Process *CURRENT = SQS.SUC, *P = 0;
    double NOW = CURRENT->EVTIME;
    switch(Code) {
    case direct:
        if (X == CURRENT)
            return;
        T = NOW; P = &SQS;
        break;
    case delay:
        T += NOW;
    case at:
        if (T <= NOW) {
            if (Prio && X == CURRENT)
                return;
            T = NOW;
        }
        break;
    case before:
    case after:
        if (!Y || !Y->SUC) {
            SQS.UNSCHEDULE(X);
            if (SQS.SUC == &SQS)
                Error("SQS is empty");
            return;
        }
        if (X == Y)
            return;
        T = Y->EVTIME;
        P = Code == before ? Y->PRED : Y;
    }
    if (X->SUC)
        SQS.UNSCHEDULE(X);
    if (!P) {
        for (P = SQS.PRED; P->EVTIME > T; P = P->PRED)
            ;
        if (Prio)
            while (P->EVTIME == T)
                P = P->PRED;
    }
    X->EVTIME = T;
    SQS.SCHEDULE(X,P);
    if (SQS.SUC != CURRENT)
        Resume(SQS.SUC);
}
```

```
void Activate(Process *P)
    { Activat(0,P,direct,0,0,0); }
void Activate(Process *P, Haste H, double T)
    { Activat(0,P,H,T,0,0); }
void Activate(Process *P, Haste H, double T, Prior Pri)
    { Activat(0,P,H,T,0,Pri); }
void Activate(Process *P1, Ranking Rank, Process *P2)
    { Activat(0,P1,Rank,0,P2,0); }

void Reactivate(Process *P)
    { Activat(1,P,direct,0,0,0); }
void Reactivate(Process *P, Haste H, double T)
    { Activat(1,P,H,T,0,0); }
void Reactivate(Process *P, Haste H, double T, Prior Pri)
    { Activat(1,P,H,T,0,Pri); }
void Reactivate(Process *P1, Ranking Rank, Process *P2)
    { Activat(1,P1,Rank,0,P2,0); }

void Accum(double &A, double &B, double &C, double D) {
    A += C*(Time() - B); B = Time(); C += D;
}
```

F. The random drawing library

Each of the functions in this library performs a random drawing of some kind. Their semantics is the same as in SIMULA.

The last parameter to the functions, `U`, is an integer variable specifying a pseudo-random number stream (seed).

```
int Draw(double A, long &U);
```

The value is 1 with the probability A, 0 with probability 1-A. It is always 1 if $A \geq 1$, and always 0 if $A \leq 0$.

```
long Randint(long A, long B, long &U);
```

The value is one of the integers `A`, `A`+1, …, `B`-1, `B` with equal probability. If $B < A$, the call constitutes an error.

```
double Uniform(double A, double B, long &U);
```

The value is uniformly distributed in the interval $A \leq x < B$. If $B < A$, the call constitutes an error.

```
double Normal(double A, double B, long &U);
```

The value is normally distributed with mean `A` and standard deviation `B`.

```
double Negexp(double A, long &U);
```

The value is a drawing form the negative exponential distribution with mean 1/`A`. If `A` is non-positive, a runtime error occurs.

```
long Poisson(double A, long &U);
```

The value is a drawing form the Poisson distribution with parameter `A`.

```
double Erlang(double A, double B, long &U);
```

The value is a drawing form the Erlang distribution with mean 1/`A` and standard deviation $1/(A * \sqrt{B})$. Both `A` and `B` must be positive.

```
long Discrete(double A[], long N, long &U);
```

The one-dimensional array `A` of `N` elements of type `double`, augmented by the element 1 to the right, is interpreted as a step function of the subscript, defining a discrete (cumulative) distribution function.

The function value satisfies

$$0 \leq \texttt{Discrete(A, U)} \leq \texttt{N}$$

It is defined as the smallest `i` such that `A[i]` > `r`, where `r` is a random number in the interval [0;1] and `A[N]` = 1.

```
double Linear(double A[], double B[], long N, long &U);
```

The value is a drawing from a (cumulative) distribution function f, which is obtained by linear interpolation in a non-equidistant table defined by `A` and `B`, such that `A`[i] = f(`B`[i]).

It is assumed that `A` and `B` are one-dimensional arrays of the same length, `N`, that the first and last elements of `A` are equal to 0 and 1, respectively, and that `A`[i] ≥ `A`[j] and `B`[i] > `B`[j] for i > j.

```
long Histd(double A[], long N, long &U);
```

The value is an integer in the range [0;`N`-1] where `N` is the number of elements in the one-dimensional array `A`. The latter is interpreted as a histogram defining the relative frequencies of the values.

Header file: `random.h`

```
#ifndef RANDOM_H
#define RANDOM_H

int Draw(double A, long &U);

long Randint(long A, long B, long &U);

double Uniform(double A, double B, long &U);

double Normal(double A, double B, long &U);

double Negexp(double A, long &U);

long Poisson(double A, long &U);

double Erlang(double A, double B, long &U);

long Discrete(double A[], long N, long &U);

double Linear(double A[], double B[], long N, long &U);

long Histd(double A[], long N, long &U);

#endif
```

Source file: random.cpp

```
#include "random.h"
#include <iostream.h>
#include <limits.h>
#include <math.h>

#define RAN 1220703125
#define MAXU INT_MAX
#define NextU (((unsigned long) (U = (U*RAN)|1))>>1)
#define Random ((NextU + 0.5) / (MAXU + 1.0))

static void Error(const char *Message) {
    cerr << "Error: " << Message << endl;
    exit(0);
}

int Draw(double A, long &U) {
    return Random < A;
}

long Randint(long A, long B, long &U) {
    if (B < A)
        Error("Randint: Second parameter is lower than first
parameter");
    long u = NextU;
    double R;
    u = (long) (R = u*(B - A + 1.0)/(MAXU + 1.0) + A);
    return u > R ? u-1 : u;
}

double Uniform(double A, double B, long &U) {
    if (B < A)
        Error("Uniform: Second parameter is lower than first
parameter");
    return Random*(B-A) + A;
}

#define p0 (-0.322232431088)
#define p1 (-1)
#define p2 (-0.342242088547)
#define p3 (-0.0204231210245)
#define p4 (-0.0000453642210148)
#define q0 0.099348462606
#define q1 0.588581570495
#define q2 0.531103462366
#define q3 0.10353775285
#define q4 0.0038560700634
```

```
double Normal(double A, double B, long &U) {
    double y, x, p, R = Random;
    p = R > 0.5 ? 1.0 - R : R;
    y = sqrt (-log (p * p));
    x = y + ((((y * p4 + p3) * y + p2) * y + p1) * y + p0)/
            ((((y * q4 + q3) * y + q2) * y + q1) * y + q0);
    if (R < 0.5)
        x = -x;
    return B * x + A;
}

double Negexp(double A, long &U) {
    if (A <= 0)
        Error("Negexp: First parameter is lower than zero");
    return -log(Random)/A;
}

long Poisson(double A, long &U) {
    double Limit = exp(-A), Prod = NextU;
    long n;
    for (n = 0; Prod >= Limit; n++)
        Prod *= Random;
    return n;
}

double Erlang(double A, double B, long &U) {
    if (A <= 0)
        Error("Erlang: First parameter is not greater than
zero");
    if (B <= 0)
        Error("Erlang: Second parameter is not greater than zero");
    long Bi = (long) B, Ci;
    if (Bi == B)
        Bi--;
    double Sum = 0;
    for (Ci = 1; Ci <= Bi; Ci++)
        Sum += log(Random);
    return (-(Sum + (B - (Ci-1))*log(Random))/(A*B));
}

long Discrete(long A[], long N, long &U) {
    double Basic = Random;
    long i;
    for (i = 0; i < N; i++)
        if (A[i] > Basic)
            break;
    return i;
}
```

```
double Linear(double A[], long B[], long N, long &U) {
    if (A[0] != 0.0 || A[N-1] != 1.0)
        Error("Linear: Illegal value in first array");
    double Basic = Random;
    long i;
    for (i = 1; i < N; i++)
        if (A[i] >= Basic)
            break;
    double D = A[i] - A[i-1];
    if (D == 0.0)
        return B[i-1];
    return B[i-1] + (B[i]-B[i-1])*(Basic-A[i-1])/D;
}

long Histd(double A[], long N, long &U) {
    double Sum = 0.0;
    long i;
    for (i = 0; i < N; i++)
        Sum += A[i];
    double Weight = Random * Sum;
    Sum = 0.0;
    for (i = 0; i < N - 1; i++) {
        Sum += A[i];
        if (Sum >= Weight)
            break;
    }
    return i;
}
```

G. Source code of the car wash simulation program

```
#include "simulation.h"
#include <iostream.h>

Head *TeaRoom, *WaitingLine;
double ThroughTime, SimPeriod, P;
long NoOfCustomers, MaxLength, N, U;

class Car : public Process {
    void Actions() {
        double EntryTime = Time();
        Into(WaitingLine);
        long QLength = WaitingLine->Cardinal();
        if (MaxLength < QLength)
            MaxLength = QLength;
        if (!TeaRoom->Empty())
            Activate((Process*) TeaRoom->First());
        Passivate();
        NoOfCustomers++;
        ThroughTime += Time() - EntryTime;
    }
};

class CarWasher : public Process {
    void Actions() {
        for (;;) {
            Out();
            while (!WaitingLine->Empty()) {
                Car *Served = (Car*) WaitingLine->First();
                Served->Out();
                Hold(10);
                Activate(Served);
                delete Served;
            }
            Wait(TeaRoom);
        }
    }
};

class CarGen : public Process {
    void Actions() {
        while (Time() <= SimPeriod) {
            Activate(new Car);
            Hold(Negexp(1/P,U));
        }
    }
};
```

```
void Report() {
    cout << N << " Car washer simulation\n";
    cout << "No.of cars through the system = "
         << NoOfCustomers << endl;
    cout << "Av.elapsed time = " << ThroughTime/NoOfCustomers
         << endl;
    cout << "Maximum queue length = " << MaxLength << endl;
}

void CarWash() {
    P = 11; N = 2; SimPeriod = 200; U = 5;
    TeaRoom = new Head;
    WaitingLine = new Head;
    for (int i = 1; i <= N; i++)
        (new CarWasher)->Into(TeaRoom);
    Activate(new CarGen);
    Hold(SimPeriod + 10000000);
    Report();
}

int main() Simulation(CarWash())
```

H. A rudimentary implementation of CBack

Header file: `CBack.h`

```
#ifndef Backtracking
#define Backtracking Sequencing

#include "coroutine.h"

unsigned long Choice(long);
void Backtrack();

extern void (*Fiasco)();

#endif
```

Source file: `CBack.cpp`

```
#include "CBack.h"
#include <stdlib.h>
#include <iostream.h>
#include <setjmp.h>
void (*Fiasco) () = 0;

static void Error(const char *Msg) {
    cerr << "Error: " << Msg << endl;
    exit(0);
}

class State : public Coroutine {
public:
    unsigned long LastChoice;
    State *Previous;
private:
    void Routine() {};
};

static State *TopState = 0, *Previous;

unsigned long Choice(long N) {
    if (N <= 0) Backtrack();
    if (N == 1) return 1;
    Previous = TopState;
    TopState = new State;
    if (!TopState)
        Error("No more space for choice\n");
    TopState->Previous = Previous;
    TopState->LastChoice = 0;
    TopState->StoreStack();
    setjmp(TopState->Environment);
    if (++TopState->LastChoice < N)
        return TopState->LastChoice;
    Previous = TopState->Previous;
    delete TopState;
    TopState = Previous;
    return N;
}

void Backtrack() {
    if (!TopState) {
        if (Fiasco)
            Fiasco();
        exit(0);
    }
    TopState->RestoreStack();
}
```

I. A complete implementation of CBack (with coroutine sequencing)

Header file: `Coroutine.h`

```
#ifndef Sequencing
#define Sequencing(S) {char Dummy; StackBottom = &Dummy; S;}
#include <stddef.h>
#include <setjmp.h>
extern char *StackBottom;
extern void (*CleanUp) ();

class Coroutine {
friend void Resume(Coroutine *);
friend void Call(Coroutine *);
friend void Detach();
friend unsigned long Choice(long);
friend void Backtrack();
friend unsigned long NextChoice();
friend class Process;
friend class State;
friend class Notification;
friend void Clean();
protected:
    Coroutine();
    ~Coroutine();
    virtual void Routine() = 0;
private:
    void Enter();
    void StoreStack();
    void RestoreStack();
    char *StackBuffer, *Low, *High;
    size_t BufferSize;
    jmp_buf Environment;
    Coroutine *Caller, *Callee;
    static Coroutine *ToBeResumed;
    State *TopState;
    unsigned long LastChoice, Alternatives;
    long Merit;
};

void Resume(Coroutine *);
void Call(Coroutine *);
void Detach();
Coroutine *CurrentCoroutine();
Coroutine *MainCoroutine();

#endif
```

Header file: `CBack.h`

```
#ifndef Backtracking
#define Backtracking Sequencing

#include "coroutine.h"
#include <stddef.h>

#define Notify(V) NotifyStorage(&V, sizeof(V))
#define Nmalloc(Size) NotifyStorage(malloc(Size), Size)
#define Ncalloc(N, Size) NotifyStorage(calloc(N,Size),
(N)*Size)
#define Nrealloc(P, Size)\
        (RemoveNotification(P),\
         NotifyStorage(realloc(P, Size), Size))
#define Nfree(P) (RemoveNotification(P), free(P))
#define ClearAll() (ClearChoices(), ClearNotifications())

unsigned long Choice(long);
void Backtrack();
unsigned long NextChoice(void);
void Cut(void);
void ClearChoices(void);

void *NotifyStorage(void *Base, size_t Size);
void RemoveNotification(void *Base);
void ClearNotifications(void);

extern void (*Fiasco)();
extern long Merit;
extern int BreadthFirst;

#endif
```

Source file: `Coroutine.cpp`

```
#define Synchronize // {jmp_buf E; if (!setjmp(E))
longjmp(E,1);}

#include "coroutine.h"
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

char *StackBottom;

void NoClean() {}
void (*CleanUp) () = NoClean;

class State;
class Notification;

State *TopState;
size_t NotifiedSpace;
Notification *FirstNotification, *N;
unsigned long LastChoice, Alternatives;
long Merit;

#define Terminated(C) (!(C)->StackBuffer && (C)->BufferSize)
static Coroutine *Current = 0, *Next;

static void Error(const char *Message) {
    cerr << "Error: " << Message << endl;
    exit(0);
}

Coroutine *Coroutine::ToBeResumed = 0;

static class MainCoroutine : public Coroutine {
public:
    MainCoroutine() { Current = this; }
    void Routine() {}
} Main;
```

```
Coroutine::Coroutine() {
    char X;
    if (StackBottom)
        if (&X < StackBottom ?
           &X <= (char*) this && (char*) this <= StackBottom :
            &X >= (char*) this && (char*) this >= StackBottom)
            Error("Attempt to allocate a Coroutine on the stack");
    StackBuffer = 0; Low = High = 0; BufferSize = 0;
    Callee = Caller = 0; TopState = 0;
    LastChoice = Alternatives = 0;
    Merit = 0;
}

Coroutine::~Coroutine() {
    CleanUp();
    delete StackBuffer;
    StackBuffer = 0;
}

inline void Coroutine::RestoreStack() {
    Synchronize;
    char X;
    if (&X >= Low && &X <= High)
        RestoreStack();
    memcpy(Low, StackBuffer, High - Low);
    longjmp(Environment, 1);
}

inline void Coroutine::StoreStack() {
    if (!Low) {
        if (!StackBottom)
            Error("StackBottom is not initialized");
        Low = High = StackBottom;
    }
    char X;
    if (&X > StackBottom)
        High = &X;
    else
        Low = &X;
    if (High - Low > BufferSize) {
        delete StackBuffer;
        BufferSize = High - Low;
        if (!(StackBuffer = new char[BufferSize]))
            Error("No more space available");
    }
    Synchronize;
    memcpy(StackBuffer, Low, High - Low);
}
```

```
inline void Coroutine::Enter() {
    if (!Terminated(Current)) {
        Current->StoreStack();
        if (setjmp(Current->Environment))
            return;
    }
    Current->TopState = ::TopState;
    Current->LastChoice = ::LastChoice;
    Current->Alternatives = ::Alternatives;
    Current->Merit = ::Merit;
    ::TopState = TopState;
    ::LastChoice = LastChoice;
    ::Alternatives = Alternatives;
    ::Merit = Merit;
    Current = this;
    if (!StackBuffer) {
        Routine();
        CleanUp();
        delete Current->StackBuffer;
        Current->StackBuffer = 0;
        if (ToBeResumed) {
            Next = ToBeResumed;
            ToBeResumed = 0;
            Resume(Next);
        }
        Detach();
    }
    RestoreStack();
}

void Resume(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Resume a non-existing Coroutine");
    if (Next == Current)
        return;
    if (Terminated(Next))
        Error("Attempt to Resume a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Resume an attached Coroutine");
    while (Next->Callee)
        Next = Next->Callee;
    Next->Enter();
}
```

```
void Call(Coroutine *Next) {
    if (!Next)
        Error("Attempt to Call a non-existing Coroutine");
    if (Terminated(Next))
        Error("Attempt to Call a terminated Coroutine");
    if (Next->Caller)
        Error("Attempt to Call an attached Coroutine");
    Current->Callee = Next;
    Next->Caller = Current;
    while (Next->Callee)
        Next = Next->Callee;
    if (Next == Current)
        Error("Attempt to Call an operating Coroutine");
    Next->Enter();
}

void Detach() {
    Next = Current->Caller;
    if (Next)
        Current->Caller = Next->Callee = 0;
    else {
        Next = &Main;
        while (Next->Callee)
            Next = Next->Callee;
    }
    Next->Enter();
}

Coroutine *CurrentCoroutine() { return Current; }

Coroutine *MainCoroutine() { return &Main; }
```

Source file: CBack.cpp

```
#include "CBack.h"
#include <iostream.h>
#include <setjmp.h>

void (*Fiasco) () = 0;
int BreadthFirst = 0;

static void Error(const char *Message)
  { cerr << "Error: " << Message << endl; exit(0); }

class State; class Notification;

extern State *TopState, *FirstFree = 0;
extern unsigned long LastChoice, Alternatives;
extern size_t NotifiedSpace;
extern Notification *FirstNotification, *N;

void Clean() {
    State *OldTopState = TopState;
    TopState = CurrentCoroutine()->TopState;
    ClearChoices();
    if (CurrentCoroutine() != MainCoroutine())
        TopState = OldTopState;
    CurrentCoroutine()->TopState = 0;
}

class State : public Coroutine {
    void Routine() {};
public:
    State() { CleanUp = Clean; }
    State *Link(State *B) {
        B->Next = Son;
        B->Previous = this;
        if (Son) Son->Previous = B;
        Son = B;
        return this;
    }
    State *Merge(State *B) {
        if (!BreadthFirst)
           return Merit >= B->Merit ? Link(B) : B->Link(this);
        return Merit > B->Merit ? Link(B) : B->Link(this);
    }
    void Insert() {
        Previous = Next = Son = 0;
        ::TopState = !::TopState ? this : Merge(::TopState);
    }
    State *Previous, *Next, *Son;
};
```

```
class Notification {
public:
    void *Base;
    size_t Size;
    Notification *Next;
};

inline void DeleteMax() {
    State *Max = TopState, *Prev, *A, *B, *C;
    if (!Max)
        return;
    Prev = TopState = 0;
    for (A = Max->Son; A && (B = A->Next); A = C) {
        C = B->Next;
        A->Next = B->Next = A->Previous = B->Previous = 0;
        TopState = B->Merge(A);
        TopState->Previous = Prev;
        Prev = TopState;
    }
    if (A) {
        A->Previous = Prev;
        TopState = A;
    }
    if (TopState) {
        for (A = TopState->Previous; A; A = B) {
            B = A->Previous;
            TopState = A->Merge(TopState);
        }
        TopState->Previous = 0;
    }
    Max->Next = FirstFree;
    FirstFree = Max;
}

void PopState() { DeleteMax(); }
```

```
unsigned long Choice(long N) {
    if (N <= 0) Backtrack();
    if (N == 1 && (!TopState || TopState->Merit <= Merit))
        return (LastChoice = Alternatives = 1);
    State *NewState;
    if (FirstFree)  {
        NewState = FirstFree;
        FirstFree = NewState->Next;
    }
    else if (!(NewState = new State))
        Error("No more space for Choice\n");
    NewState->LastChoice = NewState->Alternatives = 0;
    NewState->Merit = Merit;
    NewState->Previous = NewState->Next = NewState->Son = 0;
    static Notification *Ntf;
    static char *B;
    for (Ntf = FirstNotification,
         B = (char *) NotifiedSpace; Ntf;
         B += Ntf->Size, Ntf = Ntf->Next)
        memcpy(B, Ntf->Base, Ntf->Size);
    NewState->StoreStack();
    setjmp(NewState->Environment);
    if (!NewState->Alternatives) {
        NewState->Alternatives = N;
        NewState->Insert();
        TopState->RestoreStack();
    }
    else {
        for (Ntf = FirstNotification,
             B = (char*) NotifiedSpace; Ntf;
             B += Ntf->Size, Ntf = Ntf->Next)
            memcpy(Ntf->Base, B, Ntf->Size);
    }
    Alternatives = TopState->Alternatives;
    Merit = TopState->Merit;
    LastChoice = ++TopState->LastChoice;
    if (LastChoice == Alternatives)
        PopState();
    return LastChoice;
}

void Backtrack() {
    if (!TopState) {
        if (Fiasco)
            Fiasco();
        Detach();
    }
    TopState->RestoreStack();
}
```

```
unsigned long NextChoice() {
    if (++LastChoice > Alternatives)
        Backtrack();
    if (LastChoice == Alternatives)
        PopState();
    else
        TopState->LastChoice = LastChoice;
    return LastChoice;
}

void Cut() {
    if (LastChoice < Alternatives)
        PopState();
    Backtrack();
}

void *NotifyStorage(void *Base, size_t Size) {
    if (TopState)
        Error("Notification (unfinished Choice calls)");
    for (Notification *N = FirstNotification; N; N = N->Next)
         if (N->Base == Base)
             return 0;
    N = new Notification;
    if (!N)
        Error("No more space for notification");
    N->Base = Base; N->Size = Size;
    NotifiedSpace += Size;
    N->Next = FirstNotification;
    FirstNotification = N;
    CleanUp = Clean;
    return Base;
}

void RemoveNotification(void *Base) {
    if (TopState)
       Error("RemoveNotification (unfinished Choice calls)");
    for (Notification *N = FirstNotification, *Prev = 0;
         N;
         Prev = N, N = N->Next) {
        if (N->Base == Base) {
            NotifiedSpace -= N->Size;
            if (!Prev)
                FirstNotification = N->Next;
            else
                Prev->Next = N->Next;
            delete N;
            return;
        }
    }
}
```

```
void ClearChoices() {
    while (TopState)
        PopState();
    LastChoice = Alternatives = 0;
}

void ClearNotifications() {
    while (FirstNotification)
        RemoveNotification(FirstNotification->Base);
}
```